

MATRIX Client-Server C# based TCP/IP Framework

Corresponds to Version 1.2

Change History

Date	Change
25.09.2021	added info about version
19.07.2021	Initial doc

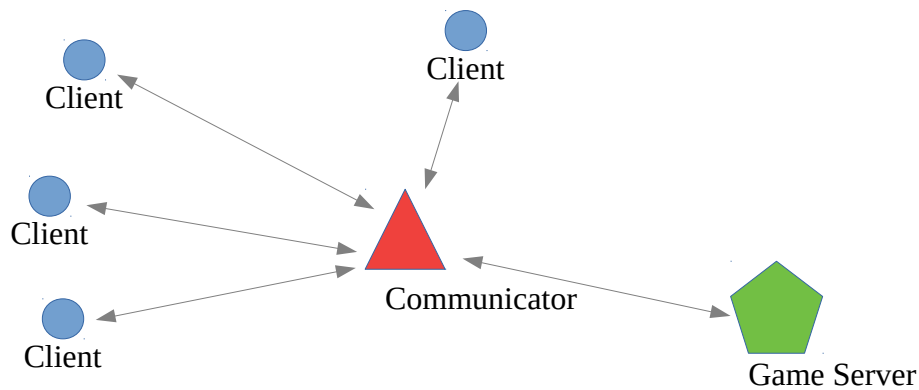
Table of Contents

Introduction.....	4
The Package.....	5
Quick Programming Guide.....	6
MATRIX API.....	12
API Methods.....	13
ForceSaveConfig.....	13
GetConfigItem.....	14
GetMasterAddress.....	15
GetSystemInfo.....	16
RegisterMethod.....	18
Send.....	20
SendToMe.....	21
SetAvailableHosts.....	23
SetHostAliases.....	24
SetComponentState.....	25
SetConfigItem.....	26
Start.....	27
Stop.....	28
Subscribe.....	29
SynchRequest.....	31
Trace.....	33
Unsubscribe.....	34
WaitForComponent.....	35
WaitForComponentInit.....	36
Property List.....	37
Data Types.....	38
CPropertyList (constructor).....	40
AddProperty.....	41
AddPropertyValue.....	42
AddBranch.....	43
FindProperty.....	44
FindPropertyV.....	45
GetProperty.....	46
DeleteProperty.....	47
GetEnum.....	48
Count.....	49
Constructing, sending and receiving Messages.....	50
Using MATRIX in Unity.....	51
Setting up the Environment.....	52
Initializing the MATRIX in your game.....	53
Using SynchRequest.....	56
Message subscriptions.....	60
Using MATRIX in Visual Studio.....	61
Communicator.....	62
Test Components.....	64

Configuration.....66
 Setting up Communication.....68
 MATRIX Component Settings.....68
 Communicator Settings.....69
 Example of the communication settings.....70
 Setting up Trace.....71

Introduction

MATRIX is a simple way to create multi-player games based on Client-Server technology. It consists of a Communicator which is supposed to run on a remote server and a couple of DLLs to be used by client programs (Unity based user interface, implementing all necessary visual effects) and by a game server (server-side application, implementing the game logic). The DLLs perform all necessary data exchange between the clients and the game server via Communicator:



In real environment the Game Server and Communicator usually run on the same remote host, when the Client applications run on player's PCs connecting to the Communicator using TCP/IP. It's possible to run all the components (Clients, Communicator, Game Server) on the individual hosts or run all of them on the same host (what is useful for developing/debugging).

All the connections between the components and communicator are handled automatically "under the hood". The components automatically re-connect to the communicator if connection has been lost by whatever reason.

The data exchange between the Clients and the Game Server is performed by sending/receiving messages. Messages can be sent by Clients and Server. Receiving messages is arranged using subscription mechanism. The MATRIX also has a convenient Request system, based on messaging: a Client can send synchronous requests to the Server and get responses back.

A number of clients capable to connect the Communicator at the same time is limited only by a number of threads available on the host machine for a process – it can be several thousands depending on the hardware. If that's not enough, you can run another communicator on a separate host. This makes the system scalable. Currently it's not possible to run more than one communicator on the same host, but this may change soon.

Existing implementation has no internal encoding or password protection. Originally it's been designed to work in local networks. It can be used for games safely – who wants to hack games? Anyway, optional encoding/password protection are planned for the future releases.

Potentially, the framework can be used to create business distributed systems (it's where it's been created and used initially).

The Package

The package contains 2 DLLs located in your *Assets/MatrixClientServer* folder. The “**CSMatrix.dll**” is a C++ based API implementing all the necessary functionality to work with the Communicator. You won’t work with this DLL directly, instead you’ll use “**zStdCSLib.dll**” what is a C# wrapper over the exported functions of “**CSMatrix.dll**”.

The assembly “**zStdCSLib.dll**” has been build on Visual Studio 2017. It’s been tested and successfully used with Unity 2018.3.0f2. Later versions of Unity and Visual Studio should not have any problems with the MATRIX DLLs.

Existing implementation supports Windows only (Windows 7 and above). The Communicator for Linux (Ubuntu/Mint) exists (you can access it for test purposes on host 139.162.234.7), but there is no interface DLLs made yet (it is planned to do in the nearest future).

The Communicator program **dcnNode.exe**, the test applications and example projects can be downloaded [here](#).

Unzip "Applications.zip" and "ExampleProjects.zip" in a convenient location. Folder "Applications" will contain two sub folders "Communicator" (with dcnNode.exe) and "TestComponents".

Folder "ExampleProjects" contains several C# projects you can look at and build. To build the example projects you'll need to add or fix references to "**zStdCSLib.dll**" in each of them. Each project also has a "post build event" copying “**CSMatrix.dll**” into output folder. If you'd like to run the built binaries you'll need to correct paths in these events to the location of “**CSMatrix.dll**” on your PC. Basically, every MATRIX based application must have both DLLs in the same folder where the executable is.

The MATRIX framework is targeted for C# programming. Potentially, the “**CSMatrix.dll**” can be used by C++ code via accessing the exported functions directly (for those who’d like to write a game server in C++), but this is not a subject of this document. I can add a convenient wrapper for C++ if there is a number of requests to do so. Also, it's possible to program on C++ directly to write applications for Linux and windows, but it's not a subject of this document either.

Quick Programming Guide

Let's have a look at two simple components `TestServer` and `TestClient` in *ExampleProjects* folder. They implement a very simple client-server application showing the main features of the `MATRIX` package.

A game server is supposed to run on a remote host and be accessible by multiple clients (player's applications). The server is supposed to have only one running instance, when a client can have thousands of them. The server usually performs a game logic and send messages to clients (players) about the current state of the game (information about game object positions, their statuses and so on). Also, the server executes different requests from clients.

The `TestServer` does very simple things: it can process a couple of requests and send a specific message. Let's have a look at the code:

```
class Program {
    static void Main(string[] args)
    {
        // Initialize the matrix (note: only one instance of the server is allowed)
        zx.matrix.CMatrixSubsystem.Start(
            "TestServer", "1.1", zx.matrix.TComponentRunType.RUN_TYPE_SINGLE);

        // Register returning server name
        zx.matrix.CMatrixSubsystem.RegisterMethod("GetMyName", Method_GetMyName);

        // Register method sending test messages
        zx.matrix.CMatrixSubsystem.RegisterMethod("SendMessages", Method_SendMessages);

        // This is called when the component has connected to the node
        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CEventConnected(),
            (msg, addr) =>
            {
                // Set signal "initialized"
                zx.matrix.CMatrixSubsystem.SetComponentState(zx.matrix.TComponentState.STATE_INITIALIZED);
                Console.WriteLine("The Server is ready to work.");
            },
            zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);
        // This is called when the component has disconnected from the node
        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CEventDisconnected(),
            (msg, addr) => Console.WriteLine("The Server is waiting for connection..."),
            zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);
    }
}
```

The first line initializes the `MATRIX` system by calling "Start" method. Here you have to provide a distinctive server name (here it's "TestServer"), application version (optional parameter, here is "1.1") and the application run type. If you don't provide the application version then by default a "system" version will be used (what is most probably a version of your Unity!). Because we require only one instance of the server, we pass "zx.matrix.TComponentRunType.RUN_TYPE_SINGLE". Note: after calling "Start" the `MATRIX` will take care under the hood about all the connectivity tasks with the communicator.

In the next two lines our server registers two methods: "GetMyName" and "SendMessages". These are requests which can be called by any client. We'll look at the methods later.

Then, the server code makes subscriptions to existing "events", provided by `MATRIX`. The event handlers are called when the server is connected or disconnected to the communicator. You don't need to implement these events, but they are convenient to trigger some logic. In our example, when the server connects to the communicator it sets its state to "STATE_INITIALIZED" and writes in console that's it's "ready". When the server is disconnected from the communicator (it may happen

if you have stopped the communicator or there is a network problem), it simply writes a message in console.

Note, that state “STATE_INITIALIZED” can be observed by client applications (this will be shown below). The client app may inform a player about the game server status.

Also, successful connection to the communicator does not mean that the server is ready to work. This is simple in our example. In real life, the server may load massive data from the database or/and do some start up logic. The game server should set its state to “STATE_INITIALIZED” when it’s really ready to work!

When the TestServer has started it just waits for requests from clients. The server can process two requests:

“GetMyName” - simply returns a name back to the requester:

```
//-----
// This dummy method just returns a name of the server
// in string output parameter "MyName"
static bool Method_GetMyName(
    zx.pl.CPropertyList inParameters,
    zx.pl.CPropertyList outParameters,
    zx.CError outError)
{
    outParameters.AddPropertyValue("MyName", "I am a Test Server");
    return true;
}
```

Mind the syntax of the method handler. It must be a function returning a boolean and accepting 3 parameters:

- **inParameters** – list of input parameters
- **outParameters** – list of output parameters
- **outError** – error descriptor

In this case the method does not expect any input parameters. It puts output value “I am a Test Server” into parameter “MyName” and returns success (true).

Another method “SendMessages” is more complicated:

```
//-----
// This dummy method starts sending TestMessage N times.
// A number of times is specified in a "long" parameter "Number"
static bool Method_SendMessages(
    zx.pl.CPropertyList inParameters,
    zx.pl.CPropertyList outParameters,
    zx.CError outError)
{
    // Get a number of messages to send
    var number = inParameters.GetProperty("Number").DC.GetLongValue();
    if (number <= 0 || number > 1000)
    {
        outError.Error(string.Format(
            "Invalid number of messages {0}. It has to be a number in range from 1 to 1000", number));
        return false;
    }

    // Send messages of type "test".
    // Each message contains one string field with info
    var msg = new zx.matrix.CStdMessage("TestMsg", "Test");
    var fldInfo = msg.Root.AddProperty(zx.pl.CDataContainer.TypeString, "Info");
    for (var i = 0; i < number; ++i)
    {
        fldInfo.DC.SetStringValue(string.Format("This is message {0} from {1}...", i + 1, number));
        zx.matrix.CMatrixSubsystem.Send(msg);
    } // for
}
```

```

    return true;
}
...

```

It retrieves a number of messages for sending from input parameter “Number”. If this parameter is out of bounds, it populates the error descriptor and returns ‘false’ indicating the failure.

If specified by the caller number of messages is fine, it creates a test message with name “TestMsg” and category “Test”, adds a string field “Info” to this message, and then sends it the requested number of times (setting field “Info” with information about message index).

Note, the TestServer “doesn’t know” who it is sending the messages to – it sends them to any subscriber of the message “TestMsg/Test”, so, potentially, this message can reach many destinations.

Our example TestServer never stops. To stop it, you can close the console window.

Ideally, the game server should quit gracefully. You can implement a specific method “StopServer” and call it, say, from a game master admin tool. Such a method may look so:

```

static bool Method_StopServer(
    zx.pl.CPropertyList inParameters,
    zx.pl.CPropertyList outParameters,
    zx.CError outError)
{
    zx.matrix.CMatrixSubsystem.Stop();
    return true;
}

```

Let’s have a look at the TestClient application which interacts with the TestServer.

```

class Program {
    const int NumberOfRequestedMessages = 500;
    static int NumberOfReceivedMessages = 0;
    static System.Threading.ManualResetEvent AllMessagesReceived =
        new System.Threading.ManualResetEvent(false);

    static void Main(string[] args)
    {
        // Initialize the matrix
        zx.matrix.CMatrixSubsystem.Start("TestClient", "1.1");

        // Subscribe for test messages sent by the server
        // This subscription is "not blocking": it guarantees that MsgHandler_TestMsg will never
        // be called in race condition
        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CStdMessage("TestMsg", "Test"),
            new zx.matrix.ProcessHandle(MsgHandler_TestMsg));

        // Connection events examples
        // This is called when the component has connected to the node
        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CEventConnected(),
            (msg, addr) =>
                Console.WriteLine(
                    "CONNECTED HAS TO " + zx.matrix.CMatrixSubsystem.GetMasterAddress().GetFullName(),
                    zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);
        // This is called when the component has disconnected from the node
        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CEventDisconnected(),
            (msg, addr) => Console.WriteLine("DISCONNECTED"),
            zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);

        // Wait for server initialization (with literally unlimited timeout)
        Console.WriteLine("Waiting for TestServer initialization..");
        zx.matrix.CMatrixSubsystem.WaitForComponentInit("TestServer", uint.MaxValue);

        // Ask server name
        Console.WriteLine("Asking server name..");
    }
}

```



```

zx.pl.CPropertyList outParams = null;
var outError = new zx.CError();
if (zx.matrix.CMatrixSubsystem.SynchRequest(
    aComponentName: "TestServer",
    aMethodName: "GetMyName",
    aInParameters: new zx.pl.CBranch(),
    outParameters: ref outParams,
    outError: ref outError,
    aPriority: zx.matrix.TPriority.PRIORITY_HIGHEST))
{
    // Executed ok
    Console.WriteLine(
        "Server name is: " + outParams.GetProperty("MyName").DC.GetStringValue());
} // if
else
{
    // Call ended with error
    Console.WriteLine("ERROR: " + outError.ErrorMessage);
    outError.ClearError(); // to re-use the same error object below
}

// Invoke server method "SendMessages"

// Do it with invalid parameters (just for example)
Console.WriteLine("Calling 'SendMessages' with bad input...");
var inParams = new zx.pl.CBranch();
inParams.AddPropertyValue("Number", 100000); // requested too many messages
zx.matrix.CMatrixSubsystem.SynchRequest(
    aComponentName: "TestServer",
    aMethodName: "SendMessages",
    aInParameters: inParams,
    outParameters: ref outParams,
    outError: ref outError,
    aPriority: zx.matrix.TPriority.PRIORITY_HIGHEST);
Console.WriteLine("ERROR: " + outError.ErrorMessage);

// Now, call the method correctly
Console.WriteLine("Calling 'SendMessages' with good input...");
// request sending messages in parameter 'Number'. This parameter was added above in inParams
inParams.GetProperty("Number").DC.SetLongValue(NumberOfRequestedMessages);
if (zx.matrix.CMatrixSubsystem.SynchRequest(
    aComponentName: "TestServer",
    aMethodName: "SendMessages",
    aInParameters: inParams,
    outParameters: ref outParams,
    outError: ref outError,
    aPriority: zx.matrix.TPriority.PRIORITY_HIGHEST))
{
    // Request sent successfully: wait till we receive all messages
    Console.WriteLine("Waiting for receiving the messages...");
    AllMessagesReceived.WaitOne();
    Console.WriteLine("All messages have been received. Mission complete, quitting...");
}
else
{
    Console.WriteLine("ERROR: " + outError.ErrorMessage);
    Console.WriteLine("Disappointed, quitting...");
}

// Inform anybody who's interested about finishing
var msgInfo = new zx.matrix.CStdMessage("TestMsgFinishInfo", "Test");
msgInfo.Root.AddPropertyValue("NumberOfReceivedMessages", NumberOfReceivedMessages);
zx.matrix.CMatrixSubsystem.Send(msgInfo);

zx.matrix.CMatrixSubsystem.Stop();
}
...

```

One can see that the first line in Main() is MATRIX initialization “zx.matrix.CmatrixSubsystem.Start("TestClient", "1.1)”. By default method “Start” uses “zx.matrix.TcomponentRunType.RUN_TYPE_MULTI” - that is what we need, we want to be able starting multiple clients.

Then the TestClient subscribes for messages sent by the TestServer, setting as a handler method “MsgHandler_TestMsg”:

```
...
    zx.matrix.CMatrixSubsystem.Subscribe(
        new zx.matrix.CStdMessage("TestMsg", "Test"),
        new zx.matrix.ProcessHandle(MsgHandler_TestMsg));
...

```

Then follow two event subscription commands “zx.matrix.CMatrixSubsystem.Subscribe(new zx.matrix.CeventConnected...” and “zx.matrix.CMatrixSubsystem.Subscribe(new zx.matrix.CEventDisconnected...””. They just trace the facts of connection/disconnection to the communicator.

After that the client consider itself “initialised” and waits for the TestServer to be “initialised” with unlimited timeout: “zx.matrix.CMatrixSubsystem.WaitForComponentInit("TestServer", uint.MaxValue);”. The MATRIX will report to the client application when a component with name “TestServer” will have “STATE_INITIALIZED”. As soon as it happens, the TestClient calls server’s method “GetMyName” sending a synchronous request:

```
...
    if (zx.matrix.CMatrixSubsystem.SynchRequest(
        aComponentName: "TestServer",
        aMethodName: "GetMyName",
        aInParameters: new zx.pl.CBranch(),
        outParameters: ref outParams,
        outError: ref outError,
        aPriority: zx.matrix.Tpriority.PRIORITY_HIGHEST)
...

```

A synchronous request blocks program execution till response has come or timeout is reached. By default method “zx.matrix.CmatrixSubsystem.SynchRequest” has unlimited timeout. The method returns "true" if successful response received or "false" (with outError set) if request has failed by whatever reason (timeout, network error, error generated by the server...). In a case of success makes sense to check outParameters if any return values are expected. In our case we expect a “server name” returned in outParameter “MyName” - the code simply prints it out:

```
...
    // Executed ok
    Console.WriteLine(
        "Server name is: " + outParams.GetProperty("MyName").DC.GetStringValue());
...

```

After asking a server name the TestClient requests the server to send a number of messages. It makes two calls: the first one should fail, because a number of messages is too big. This demonstrates how error is processed and returned by the server. The second call does it right asking to send 500 messages (this number is stored in constant “NumberOfRequestedMessages”), what is accepted by server:

```
...
    inParams.GetProperty("Number").DC.SetLongValue(NumberOfRequestedMessages);
    if (zx.matrix.CMatrixSubsystem.SynchRequest(
        aComponentName: "TestServer",
        aMethodName: "SendMessages",
        aInParameters: inParams,
        outParameters: ref outParams,
        outError: ref outError,
        aPriority: zx.matrix.TPriority.PRIORITY_HIGHEST))
    {
        // Request sent successfully: wait till we receive all messages
        Console.WriteLine("Waiting for receiving the messages...");
        AllMessagesReceived.WaitOne();
        Console.WriteLine("All messages have been received. Mission complete, quitting...");
    }
...

```

Note, if the call is successful, the client waits for signalled event “AllMessagesReceived”, set by method “MsgHandler_TestMsg” - subscription handler for the “TestMsg/Test”.

Let’s have a look at this handler:

```
...
// Message handler for test Message
private static void MsgHandler_TestMsg(zx.matrix.CMessage aMsg, zx.matrix.CAddress aAddr)
{
    // We use "shared resource" NumberOfReceivedMessages here, but no need to care about
    // locking, because the subscription was done as "non blocking"
    ++NumberOfReceivedMessages;

    // Decerialize the message
    var msg = zx.matrix.CStdMessage.Deserialize(aMsg);
    Console.WriteLine(
        string.Format(
            "=> Received message {0}/{1} from {2}. Info: '{3}'",
            NumberOfReceivedMessages, NumberOfRequestedMessages,
            aAddr.GetFullName(),
            msg.GetFullType(),
            msg.Root.GetProperty("Info").DC.GetStringValue());
    if (NumberOfReceivedMessages >= NumberOfRequestedMessages)
    {
        // All messages received: flag the event
        AllMessagesReceived.Set();
    }
}
...

```

The method counts a number of received messages by incrementing “NumberOfReceivedMessages”. Note, the received message must be explicitly de-serialised from an abstract “zx.matrix.Cmessage” class:

```
...
var msg = zx.matrix.CStdMessage.Deserialize(aMsg);
...

```

Only after de-serialization variable “msg” will have all fields sent by the server. The code simply prints the info field out and checks a number of received messages reaches the number of requested messages. As soon as it happens, the event “AllMessagesReceived” is triggered.

After that, the TestClient considers its mission as “done” and quits:

```
...
// Inform anybody who's interested about finishing
var msgInfo = new zx.matrix.CStdMessage("TestMsgFinishInfo", "Test");
msgInfo.Root.AddPropertyValue("NumberOfReceivedMessages", NumberOfReceivedMessages);
zx.matrix.CMatrixSubsystem.Send(msgInfo);

zx.matrix.CMatrixSubsystem.Stop();
...

```

But before quitting it sends another message to any subscriber (not presented in the examples) about successfully fulfilled job (in reality, TestServer/TestClient participate in one of the automated tests of the MATRIX system and this “TestMsgFinishInfo” message is caught by the test program).

That's it. In these examples the TestClient demonstrates abilities to request the TestServer, and makes it to fulfil some actions (like sending messages); the TestServer implements the requests and sends messages to TestClient(s). Running many TestClients is possible, but it will break their logic (in a way the logic is written). This is only a brief example of how the coding looks.

MATRIX API

You have access to the MATRIX functionality via the API provided by “**zStdCSLib.dll**”.

API Methods

All the API functions are static public methods of class “CMatrixSubsystem”. Here are all the API methods in alphabetic order.

ForceSaveConfig

Signature:

```
public static void ForceSaveConfig() {...}
```

Description:

Forces saving changes made by "SetCfgItem" calls into the configuration file. When you call "SetCfgItem" the data are not saved into the configuration file immediately. The MATRIX framework just notes the changes been made and does the save when you gracefully quit the application. You may want to save the changes straight away to avoid loosing your changed data due to unexpected application crash or PC reboot.

Example:

```
...
// Set up default configuration
var hostAliases = zx.matrix.CMatrixSubsystem.GetCfgItem(CfgItemPath_DCNNostAliases);
if (string.IsNullOrEmpty(hostAliases))
{
    zx.matrix.CMatrixSubsystem.SetCfgItem(CfgItemPath_DCNAvailableHosts, DefaultAvailableHosts);
    zx.matrix.CMatrixSubsystem.SetCfgItem(CfgItemPath_DCNNostAliases, DefaultHostsAliases);
    zx.matrix.CMatrixSubsystem.ForceSaveConfig();
}
...
```

GetConfigItem

Signature:

```
public static unsafe string GetCfgItem(string aPath) {...}
```

Parameters:

- **aPath** (string): a path to configuration item (like "DCN.AvailableHosts") in the config file

Description:

This function allows you to retrieve a specific configuration item from the configuration file. If the configuration item does not exist, the function returns empty string.

Example:

```
...  
var availableHosts = zx.matrix.CMatrixSubsystem.GetCfgItem("DCN.AvailableHosts");  
...
```

GetMasterAddress

Signature:

```
public static Address GetMasterAddress() {...}
```

Description:

This method returns an address the component it is connected to. It's a communicator address.

"CAddress" is a class with 3 fields:

```
public class CAddress : CSerialObject{
    public int Port = 0;           // Port number
    public string HostName = ""; // Host name
    public string SessionID = ""; // session id
}
```

Example:

```
private void WhenConnectedEvent(zx.matrix.CMessage aMsg, zx.matrix.CAddress aAddr)
{
    statusBarPanelConnection.Text = "Connected to " +
        CMatrixSubsystem.GetMasterAddress().GetFullName();
    statusBarPanelConnection.Icon =
        Icon.FromHandle(((Bitmap)imageListMain.Images[IMAGE_LAMP_ON]).GetHicon());
}
```

GetSystemInfo

Signature:

```
public static CPLSystemInfo GetSystemInfo() {...}
```

Description:

Returns a property containing the system information.

The system information contains data about all the components, connected to the communicator and messages these components are subscribed to. This may be interesting for specific admin components like monitors (see *ExampleProjects/dcnMonitor*).

"CPLSystemInfo" is a container with two branches "Components" and "Messages":

```
public class CPLSystemInfo : zx.pl.CPropertyList {
    // Field names
    public const string FldComponents = "Components";
    public const string FldMessages = "Messages";
    ...
    // Extractors
    public zx.pl.CPropertyList Components
    {
        get { return FindPropertyV(FldComponents); }
    }
    public zx.pl.CPropertyList Messages
    {
        get { return FindPropertyV(FldMessages); }
    }
    ...
}
```

Branch "Components" holds information about all the components currently connected to the communicator. Each item stored in "Components" is a property list of type "zx.matrix.CPLComponentDesc" with following fields:

```
public class CPLComponentDesc : zx.pl.CPropertyList {
    // Field names
    public const string FldSessionID = "SessionID";
    public const string FldType = "Type";
    public const string FldName = "Name";
    public const string FldRank = "Rank";
    public const string FldRunType = "RunType";
    public const string FldStartTime = "StartTime";
    public const string FldState = "State";
    public const string FldPort = "Port";
    public const string FldHostName = "HostName";
    public const string FldParameters = "Parameters"; // Additional parameters
    ...
}
```

The best way to understand these fields is to look at *ExampleProjects/dcnMonitor* project and see how they are used there.

Branch "Messages" contains information about the messages subscribed by different components. This sort of data is quite specific and I omit the details in this document.

Example:

Here is an implementation of "WaitForComponent" (see below) MATRIX method, using "GetSystemInfo":

```
public static bool WaitForComponent(
    string aComponentName, zx.matrix.TComponentState aState, uint aTimeOutMs)
{
    const int _time_period = 500;
    for (int t = 0; !IsStopping() && t <= aTimeOutMs; t += _time_period)
    {
        CPLComponentDesc desc = GetSystemInfo().FindComponent(aComponentName);
    }
}
```



```
    if (desc == null && aState == zx.matrix.TComponentState.STATE_STOPPED ||
        desc != null && desc.State == (int)aState)
        return true;
    System.Threading.Thread.Sleep(_time_period);
} // while

// Timeout
return false;
}
```

RegisterMethod

Signature:

```
public static void RegisterMethod(
    string aMethodName, ProcessOMMethodHandle aHandler) {...}
```

Parameters:

- **aMethodName** (string): method name
- **aHandler** (ProcessOMMethodHandle): methods handler

Description:

This function is used to register a named method which can be called remotely from another application via network. In the most cases the methods are implemented on the server side and they are called by the clients.

ProcessOMMethodHandle is a delegate type where you are supposed to implement the method's logic:

```
public delegate bool ProcessOMMethodHandle(
    zx.pl.CPropertyList inParameters, zx.pl.CPropertyList outParameters, zx.CError outError);
```

"inParameters" is a container of input parameters. "outParameters" is a container of what the method returns to the caller. The method should return "true" in success and "false" in error case. In the error case "outError" is expected to be populated by error details.

Examples:

Working examples of method implementation could be found in TestServer project (see *ExampleProjects/TestServer*). Registration:

```
...
// Register returning server name
zx.matrix.CMatrixSubsystem.RegisterMethod("GetMyName", Method_GetMyName);

// Register method sending test messages
zx.matrix.CMatrixSubsystem.RegisterMethod("SendMessages", Method_SendMessages);
...
```

...and implementation:

```
...
// This dummy method starts sending TestMessage N times.
// A number of times is specified in a "long" parameter "Number"
static bool Method_SendMessages(
    zx.pl.CPropertyList inParameters,
    zx.pl.CPropertyList outParameters,
    zx.CError outError)
{
    // Get a number of messages to send
    var number = inParameters.GetProperty("Number").DC.GetLongValue();
    if (number <= 0 || number > 1000)
    {
        outError.Error(string.Format(
            "Invalid number of messages {0}. It has to be a number in range from 1 to 1000", number));
        return false;
    }

    // Send messages of type "test".
    // Each message contains one string field with info
    var msg = new zx.matrix.CStdMessage("TestMsg", "Test");
    var fldInfo = msg.Root.AddProperty(zx.pl.CDataContainer.TypeString, "Info");
    for (var i = 0; i < number; ++i)
    {
        fldInfo.DC.SetStringValue(string.Format("This is message {0} from {1}...", i + 1, number));
        zx.matrix.CMatrixSubsystem.Send(msg);
    }
}
```

```

    } // for
    return true;
}
...

```

One can see that this method checks expected parameters (here just one parameter "Number"), returns error if the parameter is incorrect, does some work (sends messages) and returns "true" if the parameter is correct. It returns no output parameters to the caller.

This simple method returns to the caller one output parameter "MyName" and always succeeds:

```

// This dummy method just returns a name of the server
// in string output parameter "MyName"
static bool Method_GetMyName(
    zx.pl.CPropertyList inParameters,
    zx.pl.CPropertyList outParameters,
    zx.CError outError)
{
    outParameters.AddPropertyValue("MyName", "I am a Test Server");
    return true;
}

```

As you may notice a set of input and output parameters entirely depends on the implementation of the methods and a way how they are called. Methods implementations should check the input expected parameters and their values. The caller side should provide expected parameters for the called methods. There is no internal checks for "not expected" parameters (at least in the present implementation of the MATRIX). You may add your own if necessary.

Methods also can be implemented as lambdas at the registration point like:

```

...
zx.matrix.CMatrixSubsystem.RegisterMethod(
    "GetMyName",
    (inParameters, outParameters, outError) =>
    {
        outParameters.AddPropertyValue("MyName", "I am a Test Server");
        return true;
    });
...

```

Send

Signature:

```
public static bool Send(CMessage aMsg, ESendType aSendType = ESendType.Normal)
{...}
```

Parameters:

- **aMsg** (CMessage): message
- **aSendType** (ESendType): type of sending

Description:

Sends a message to any recipients subscribed to it. "aMsg" should be a message object derived from "zx.matrix.CMessage".

"ESendType" is an enum:

```
public enum ESendType { ToMe, Normal };
```

"ESendType.ToMe" means that the message is sent internally – it will not leave the application. Internal messages are useful to notify other parts of the application about something. They are also called "events".

"ESendType.Normal" - is a default sending mode when the message goes via network to all the components subscribed to it.

Examples:

From TestClient project (see *ExampleProjects/TestClient*):

```
...
// Inform anybody who's interested about finishing
var msgInfo = new zx.matrix.CStdMessage("TestMsgFinishInfo", "Test");
msgInfo.Root.AddPropertyValue("NumberOfReceivedMessages", NumberOfReceivedMessages);
zx.matrix.CMatrixSubsystem.Send(msgInfo);
...
```

From TestSender project (see *ExampleProjects/TestSender*):

```
...
// Create a message
zx.matrix.CStdMessage msg = new zx.matrix.CStdMessage("test");
for (int i = 1; i <= n && mIsSendingEvent.WaitOne(0, true); i++)
{
    zx.pl.CPropertyList p = msg.Root.AddProperty(zx.pl.CDataContainer.TypeString, "text");
    p.DC.SetStringValue(i.ToString() + " " + txtMsg.Text);
    CMatrixSubsystem.Send(msg);
    labelNumberOfSent.Text = "Sent " + i.ToString() + " from " + n.ToString();
} // for
...
```

SendToMe

Signature:

```
public static bool SendToMe(CMessage aMsg) {...}
```

Parameters:

- **aMsg** (CMessage): message

Description:

Sends message internally (basically it's a Send call with aSendType == "EsendType.ToMe").

Example:

(Taken from a unit test. It also demonstrates creation of a complex message)

```
[Test]
public void Test_SendingReceivingMessage_WithAllTypesFields()
{
    // --ARRANGE

    // Prepare message
    var msg = new zx.matrix.CStdMessage("TestMsg", "Test");
    var fldBranch= msg.Root.AddProperty(zx.pl.CDataContainer.TypeBranch,
"branch");
    var brFld1 = fldBranch.AddPropertyValue("brFld1", "Hello");
    var brFld2 = fldBranch.AddProperty(zx.pl.CDataContainer.TypeUID, "brFld2");
brFld2.DC.SetStringValue(System.Guid.NewGuid().ToString());
    var fldLong = msg.Root.AddProperty(zx.pl.CDataContainer.TypeLong, "long");
fldLong.DC.SetLongValue(12345);
    var fldInt64 = msg.Root.AddProperty(zx.pl.CDataContainer.TypeInt64, "int64");
fldInt64.DC.SetInt64Value(3889292999);
    var fldBool = msg.Root.AddProperty(zx.pl.CDataContainer.TypeBool, "bool");
fldBool.DC.SetBoolValue(true);
    var fldDouble = msg.Root.AddProperty(zx.pl.CDataContainer.TypeDouble,
"double");
fldDouble.DC.SetDoubleValue(3.141592);
    var fldString = msg.Root.AddProperty(zx.pl.CDataContainer.TypeString,
"string");
fldString.DC.SetStringValue("damn microsoft");
    var fldDate = msg.Root.AddProperty(zx.pl.CDataContainer.TypeDate, "date");
fldDate.DC.SetDateTimeValue(DateTime.UtcNow); // must be in UTC time for
proper deserialisation
    var fldUid = msg.Root.AddProperty(zx.pl.CDataContainer.TypeUID, "uid");
fldUid.DC.SetStringValue(System.Guid.NewGuid().ToString());

    // Subscribe for the message
    var receivedEvent = new System.Threading.ManualResetEvent(false);
    CMessage receivedMessage = null;
    CMatrixSubsystem.Subscribe(
        msg,
        (message, addr) =>
        {
            receivedMessage = message;
            receivedEvent.Set();
        });

    // --ACT

    // Send message to itself
    CMatrixSubsystem.SendToMe(msg);

    // Waiting for receive
```

```

Assert.IsTrue(receivedEvent.WaitOne(5000));

// Deserialize the message
var chkMsg = zx.matrix.CStdMessage.Deserialize(receivedMessage);

// --CHECK

var chkBranch = chkMsg.Root.FindProperty("branch");
Assert.NotNull(chkBranch);
var fld = chkBranch.FindProperty("brFld1");
Assert.NotNull(fld);
Assert.AreEqual(brFld1.DC.GetStringValue(), fld.DC.GetStringValue());
fld = chkBranch.FindProperty("brFld2");
Assert.NotNull(fld);
Assert.AreEqual(brFld2.DC.GetStringValue(), fld.DC.GetStringValue());

fld = chkMsg.Root.FindProperty("long");
Assert.NotNull(fld);
Assert.AreEqual(fldLong.DC.GetLongValue(), fld.DC.GetLongValue());

fld = chkMsg.Root.FindProperty("int64");
Assert.NotNull(fld);
Assert.AreEqual(fldInt64.DC.GetInt64Value(), fld.DC.GetInt64Value());

fld = chkMsg.Root.FindProperty("bool");
Assert.NotNull(fld);
Assert.AreEqual(fldBool.DC.GetBoolValue(), fld.DC.GetBoolValue());

fld = chkMsg.Root.FindProperty("double");
Assert.NotNull(fld);
Assert.AreEqual(fldDouble.DC.GetDoubleValue(), fld.DC.GetDoubleValue());

fld = chkMsg.Root.FindProperty("string");
Assert.NotNull(fld);
Assert.AreEqual(fldString.DC.GetStringValue(), fld.DC.GetStringValue());

fld = chkMsg.Root.FindProperty("date");
Assert.NotNull(fld);
//fld.DC.GetDateTimeValue().Ticks
// Due to serialization some ms are lost, so it's Ok to be not equal
var origTime = fldDate.DC.GetDateTimeValue().ToLocalTime(); // local times are
compared
var receivedTime = fld.DC.GetDateTimeValue();
Assert.AreNotEqual(origTime, receivedTime);
// But till milliseconds everything is fine
Assert.AreEqual(origTime.Year, receivedTime.Year);
Assert.AreEqual(origTime.Month, receivedTime.Month);
Assert.AreEqual(origTime.Day, receivedTime.Day);
Assert.AreEqual(origTime.Hour, receivedTime.Hour);
Assert.AreEqual(origTime.Minute, receivedTime.Minute);
Assert.AreEqual(origTime.Second, receivedTime.Second);
Assert.AreEqual(origTime.Millisecond, receivedTime.Millisecond);

fld = chkMsg.Root.FindProperty("uid");
Assert.NotNull(fld);
Assert.AreEqual(fldUid.DC.GetStringValue(), fld.DC.GetStringValue());
}
}
...

```

SetAvailableHosts

Signature:

```
public static void SetAvailableHosts(string aHosts) {...}
```

Parameters:

- **aHosts** (string): comma separated list of available hosts like "localhost,139.162.223.8"

Description:

Call "SetAvailableHosts" if you'd like to configure available hosts programmatically (rather than manually changing the configuration file) from your application.

More details about available hosts one can find in Configuration topic.

Example:

(Taken from TestReceiver project – see *ExampleProjects/TestReceiver*)

```
private void btnConnectionApply_Click(object sender, System.EventArgs e)
{
    // Here I lock the button to avoid of friquent clicking
    btnConnectionApply.Enabled = false;
    CMatrixSubsystem.SetAvailableHosts(tbAvailableHosts.Text);
    System.Threading.Thread.Sleep(2000);
    btnConnectionApply.Enabled = true;
}
```

SetHostAliases

Signature:

```
public static void SetHostAliases(string aHostsAliases) {...}
```

Parameters:

- **aHostAliases** (string): comma separated list of host aliases like "ubuntu:139.162.233.2,MyRemoteHost:139.162.212.3"

Description:

Call "SetHostAliases" if you'd like to configure the host aliases programmatically (rather than manually changing the configuration file) from your application.

More details about available hosts one can find in Configuration topic.

Example:

```
...  
    zx.matrix.CMatrixSubsystem.SetCfgItem(CfgItemPath_DCNNostAliases, DefaultHostsAliases);  
...
```


SetComponentState

Signature:

```
public static void SetComponentState(TComponentState aState) {...}
```

Parameters:

- **aState** (zx.matrix.TComponentState): a component state

Description:

Forcibly changes a state of your component to "aState".

The MATRIX framework automatically changes the state of your component to "STATE_STOPPED" or "STATE_STARTED". However, you might want to set the state to "STATE_INITIALIZED" when the component is ready to work or to "STATE_ERROR", if the component is in error state and can't normally operate.

Only these states are "officially" recognised by the MATRIX framework:

```
STATE_ERROR      = -1,
STATE_STOPPED    = 0,
STATE_STARTED    = 1, // is set automatically, when created
STATE_IDLE       = 2, // is set, when a component is in idle state
STATE_INITIALIZED = 222 // must be set by user application
```

Any other integers are possible, but will not be recognised by other components, like "dcnMonitor", for example.

Example:

This component is happy to set itself to "initialized" when it's connected to the communicator:

```
...
// This is called when the component has connected to the node
zx.matrix.CMatrixSubsystem.Subscribe(
    new zx.matrix.CEventConnected(),
    (msg, addr) =>
    {
        // Set signal "initialized"
        zx.matrix.CMatrixSubsystem.SetComponentState(zx.matrix.TComponentState.STATE_INITIALIZED);
        Console.WriteLine("The Server is ready to work.");
    },
    zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);
...

```

SetConfigItem

Signature:

```
public static void SetCfgItem(string aPath, string aNewValue) {...}
```

Parameters:

- **aPath** (string): a path to configuration item (like "DCN.AvailableHosts") in the configuration file
- **aNewValue** (string): new value of the configuration item.

Description:

This function allows you programmatically change a specific configuration item in the configuration file.

If the configuration item does not exist, it will be added to the configuration with the new value.

Example:

```
...  
    // Set up default configuration  
    var hostAliases = zx.matrix.CMatrixSubsystem.GetCfgItem(CfgItemPath_DCNNostAliases);  
    if (string.IsNullOrEmpty(hostAliases))  
    {  
        zx.matrix.CMatrixSubsystem.SetCfgItem(CfgItemPath_DCNAvailableHosts, DefaultAvailableHosts);  
        zx.matrix.CMatrixSubsystem.SetCfgItem(CfgItemPath_DCNNostAliases, DefaultHostsAliases);  
        zx.matrix.CMatrixSubsystem.ForceSaveConfig();  
    }  
...
```

Start

Signature:

```
public static void Start(string aComponentName, TComponentRunType aRunType) {...}
public static void Start(string aComponentName) {...}
```

Parameters:

- **aComponentName** (string): a name of the component
- **aRunType** (TComponentRunType): a component run type

Description:

Call Start to initialise the MATRIX in your application. Provide a sane not empty component name. Passing empty string will cause assigning to the component a name “`__ANONYMOUS_COMPONENT__`”, what is definitely not what you want.

TComponentRunType is an enum:

```
// Supported component run types
public enum TComponentRunType {
    RUN_TYPE_MULTI = 0, // only one instance of a component with a specific name can be running
    RUN_TYPE_SINGLE = 1 // any number of instances is allowed (default)
}; // TComponentRunType
```

Use RUN_TYPE_MULTI for the client applications (your game interface) which supposed to have any number of instances and RUN_TYPE_SINGLE for the game server.

The Start call initializes the MATRIX component and automatically connects/reconnects it to the communicator when it becomes available. Call Start() before calling any other API methods. Only one Start() call can be done in your application – attempts to call Start() again are ignored by internal logic.

Examples:

```
...
// Initialize the matrix (note: only one instance of the server is allowed)
zx.matrix.CMatrixSubsystem.Start("TestServer", zx.matrix.TcomponentRunType.RUN_TYPE_SINGLE);
...
```

Stop

Signature:

```
public static void Stop() {...}
```

Description:

Stop() gracefully stops the MATRIX component (disconnects it from the communicator, shuts down all internal threads and so on). Call Stop before quitting your application.

After calling Stop() you won't be able to start MATRIX by calling Start() again (at least it's not intended and behaviour in such case is not guaranteed).

Subscribe

Signature:

```
public static System.Guid Subscribe(
    CMessage aMsg, ProcessHandle aHandle,
    ESubscriptionType aSubscriptionType = ESubscriptionType.Message,
    bool aIsBlocking = false) {...}
```

Parameters:

- **aMsg** (CMessage): message to subscribe
- **aHandle** (ProcessHandle): message handler (called when message received)
- **aSubscriptionType** (EsubscriptionType): type of subscription
- **aIsBlocking** (bool): is message handler blocking or not

Description:

Method Subscribe is used to catch the messages of specific type and process them by a specified message handler. A message class you are subscribing for is specified by parameter “aMsg”. It must be derived from “zx.matrix.CMessage” class.

“aHandle” is a delegate of type “ProcessHandle”:

```
public delegate void ProcessHandle(CMessage aMsg, CAddress aSender);
```

You may well use lambda here.

Subscription type is an enum:

```
public enum ESubscriptionType { Event, Message };
```

Use “ESubscriptionType.Event” to subscribe for internal MATRIX events and “ESubscriptionType.Message” to subscribe for external messages (sent by other components via network).

Parameter "aIsBlocking" may affect performance of the message processing. Method "Subscribe" expects "the worse" and processes every single incoming message in a separate thread. It's been done to avoid possible blocking the message flow by a bad behaving handler which may block (on a mutex or event waiting for example) and wait for something. This also means, that potentially all message handlers are called concurrently and therefore any shared data must be protected (by using the same mutex or so...). If you know that your message handler is simple and never blocks, subscribe with "aIsBlocking = false". This will guarantee, that message handlers of this type will be called using one after another. You also don't need to protect shared data in such handler. Using "aIsBlocking" as "false" is a sort of a contract between you and the MATRIX: you "promise" that your code will be fast and non blocking. If you cheat, your system may stuck in the internal message processing queue.

Method subscribe returns a Guid – unique id of the subscribed message. This id can be used later to explicitly unsubscribe from the message, if necessary.

Examples:

Events subscriptions (blocking by default):

```
...
    // Subscribe for system information event
    Subscribe(
        new zx.matrix.CEventSysInfo(),
        new ProcessHandle(Handler_EventSysInfoChanged), ESubscriptionType.Event);
    // Method call result event
    Subscribe(
        new zx.matrix.CEventMethodCallResult(),
        new ProcessHandle(Handler_EventMethodCallResult), ESubscriptionType.Event);
...
```

Non blocking message subscription:

```
...
    CMatrixSubsystem.Subscribe(
        new zx.matrix.CStdMessage("test"),
        new ProcessHandle(MsgHandler_Std),
        CMatrixSubsystem.ESubscriptionType.Message, false);
...
```

Blocking message subscription using lambda:

```
...
    // Subscribe for the message
    var receivedEvent = new System.Threading.ManualResetEvent(false);
    CMessage receivedMessage = null;
    CMatrixSubsystem.Subscribe(
        msg,
        (message, addr) =>
        {
            receivedMessage = message;
            receivedEvent.Set();
        });
...
```

SynchRequest

Signature:

```
public static bool SynchRequest(
    string aComponentName,
    string aMethodName,
    zx.pl.CPropertyList aInParameters,
    ref zx.pl.CPropertyList outParameters,
    ref zx.CError outError,
    string aObjectPath = "",
    uint aTimeoutMs = DefaultTimeoutMs,
    TPriority aPriority = zx.matrix.TPriority.PRIORITY_NORMAL) {...}
```

Parameters:

- **aComponentName** (string): a name of a MATRIX component (assigned when calling Start method)
- **aMethodName** (string): a name of a method to be called
- **aInParameters** (zx.pl.CPropertyList): container of input parameters
- **outParameters** (zx.pl.CPropertyList): container of output parameters
- **outError** (zx.CError): result error descriptor
- **aObjectPath** (string): path to the object method owner inside the component. Empty string "" means that the component itself is a holder of the method
- **aTimeoutMs** (uint): timeout in milliseconds. Default value is DefaultTimeoutMs=30000 (30 seconds)
- **aPriority** (TPriority): priority of the request

Description:

SynchRequest calls a method of another MATRIX component (registered by RegisterMethod) via network. The program execution halts till a response has come back or timed out.

"aObjectPath" is necessary if the method belongs to a sub-object of the calling component. For example, a component "TraceCollector" may have internal object "Configuration" which has method "SetItem". In this case, a call would look so:

```
zx.matrix.CMatrixSubsystem.SynchRequest(
    aComponentName: "TraceCollector",
    aMethodName: "SetItem",
    aObjectPath: "Configuration"
    ...
```

"TPriority" is an enum:

```
public enum TPriority {
    PRIORITY_LOWEST = 0,
    PRIORITY_BELOW_NORMAL = 127,
    PRIORITY_NORMAL = 128,
    PRIORITY_ABOVE_NORMAL = 129,
    PRIORITY_HIGHEST = 255
}; // TPriority
```

Using priority > "PRIORITY_NORMAL" ensures the request will avoid any internal queueing and will be executed faster.

Example:

From TestClient project (see *ExampleProjects/TestClient*):

```
...
zx.pl.CPropertyList outParams = null;
var outError = new zx.CError();
if (zx.matrix.CMatrixSubsystem.SynchRequest(
    aComponentName: "TestServer",
    aMethodName: "GetMyName",
    aInParameters: new zx.pl.CBranch(),
    outParameters: ref outParams,
    outError: ref outError,
    aPriority: zx.matrix.TPriority.PRIORITY_HIGHEST))
{
    // Executed ok
    Console.WriteLine(
        "Server name is: " + outParams.GetProperty("MyName").DC.GetStringValue());
} // if
else
{
    // Call ended with error
    Console.WriteLine("ERROR: " + outError.ErrorMessage);
    outError.ClearError(); // to re-use the same error object below
}
...

```


Trace

Signature:

```
public static void Trace(string aMsg) {...}
```

Parameters:

- **aMsg** (string): *a string to be written in a trace file*

Description:

Writes a string into a trace file (see Trace for details)

Example:

```
...  
CMatrixSubsystem.Trace("+++ +++ NUMBER OF PROCESSING HANDLERS=" + mProcessorsNum.ToString());  
...
```

Unsubscribe

Signature:

```
public static void Unsubscribe(System.Guid aSubscriptionID) {...}
```

Parameters:

- **aSubscriptionID** (System.Guid): id of a message subscription to unsubscribe

Description:

Call Unsubscribe to explicitly unsubscribe a specific subscription. If such subscription exists, the internal message processor will remove the subscription. If subscription doesn't exist, nothing will happen.

Normally you don't need to unsubscribe the messages during your application live time. What may happen is you created a subscription with a handler which is going to be destroyed (you close a window, for example, which processes a message). In this case before the handler is destroyed you must unsubscribe the handler associated with the message, otherwise, if message comes, the internal message processor will call a destroyed handler with very bad consequences...

Examples:

This example is taken from dcnMonitor example project:

```
private void FormComponents_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    // Unsubscribe from the messages
    CMatrixSubsystem.Unsubscribe(mId_CMsgSystemInfo);
}
```

Here, a form of components, subscribed to system info message is closing, hence – a correspondent handler should be removed.

WaitForComponent

Signature:

```
public static bool WaitForComponent(
    string aComponentName, zx.matrix.TComponentState aState, uint aTimeOutMs)
{...}
```

Parameters:

- **aComponentName** (string): a name of a MATRIX component (assigned when calling Start method)
- **aState** (zx.matrix.TComponentState): a component state
- **aTimeOutMs** (uint): timeout in milliseconds

Description:

This function blocks further execution till a specified by name component has a state "aState", or a time out occurs. "zx.matrix.TComponentState" is an enum:

```
public enum TComponentState {
    STATE_ERROR      = -1,
    STATE_STOPPED    = 0,
    STATE_STARTED    = 1, // is set automatically, when created
    STATE_IDLE       = 2, // is set, when a component is in idle state
    STATE_INITIALIZED = 222 // must be set by user application
}; // TComponentState
```

Example:

Here we are waiting for "STATE_INITIALIZED" from another more popular method:

```
public static bool WaitForComponentInit(string aComponentName, uint aTimeOutMs)
{
    return WaitForComponent(
        aComponentName, zx.matrix.TComponentState.STATE_INITIALIZED, aTimeOutMs);
}
```

WaitForComponentInit

Signature:

```
public static bool WaitForComponentInit(string aComponentName, uint aTimeOutMs)
{...}
```

Parameters:

- ***aComponentName*** (string): a name of a MATRIX component (assigned when calling Start method)
- ***aTimeOutMs*** (uint): timeout in milliseconds

Description:

This is a specialization of "WaitForComponent" waiting for state "STATE_INITIALIZED".

Example:

```
...
// wait for server initialization (with literally unlimited timeout)
Console.WriteLine("Waiting for TestServer initialization...");
zx.matrix.CMatrixSubsystem.WaitForComponentInit("TestServer", uint.MaxValue);
...
```

Property List

Property list is a universal container widely used in MATRIX to transfer structured data. It's used in CStdMessage, in the requests input/output parameters and so on.

A property list is a tree-like container, where each element ("property") has name and a Data Container (actually a value) of a specific type, and also it may have a list of other properties.

MATRIX "knows" how to serialize/de-serialize the property lists in order to transfer it via the network.

The class contains the following important members:

```
public class CPropertyList {
    // Name
    public string Name = "";
    // Data container
    public CDataContainer DC;
    // List of properties
    private SortedList mList = new SortedList();
    ...
}
```

"*Name*" - is a property list name. It may contain Latin letters, digits, underscores.

"*DC*" - is a data container – holds a value of the property list. The DC must have one of the types listed below (see Data Types).

"*mList*" (private member shown for reference) - is a list of member properties. Each property in the list has a unique name, not necessarily equal to the name of the property itself (See "AddProperty" methods)

Data Types

Only the following data types are supported by the MATRIX:

```
...
// Basic data types (C++ property lists compatible)
public const string TypeVoid = "void";
public const string TypeBranch = "branch"; // for branches only
public const string TypeLong = "long";
public const string TypeInt64 = "int64";
public const string TypeBool = "bool";
public const string TypeDouble = "double";
public const string TypeString = "string";
public const string TypeDate = "date";
public const string TypeUID = "uid";
...
```

It's possible, but not recommended to invent your own types. At least it's not currently explained in this document how to do it properly.

TypeVoid

An "empty" type with no data. Sometimes property lists of such a type are useful as "flag" fields or empty results.

TypeBranch

"Branches" are designed to hold other properties. They have no values.

TypeLong

Holds a signed 32 bit integer

TypeInt64

Holds a signed 64 bit integer

TypeBool

Holds a boolean value (true or false)

TypeDouble

Holds a signed double value

TypeString

Holds a string value of any length

TypeDate

Holds "System.DateTime" value

TypeUID

Holds "System.Guid" value

Although, only "TypeBranch" is recommended to hold other property lists, it is not forbidden to add properties to any type, say "TypeLong".

All these types are specified in the "CDataContainer" class – a property lists value holder.

The abstract "CDataContainer" class looks so:

```
public class CDataContainer : CSerialObject {
...
    //-----
    // interface to basic types
    //-----
    // Returns data type
    public virtual new string GetType()
    {
        return TypeVoid;
    }

    public virtual int GetLongValue()
    {
        return 0;
    }
    public virtual void SetLongValue(int aValue)
    {}

    public virtual uint GetULongValue()
    {
        return (uint)GetLongValue();
    }
    public virtual void SetULongValue(uint aValue)
    {}

    public virtual long GetInt64Value()
    {
        return (long)GetLongValue();
    }
    public virtual void SetInt64Value(long aValue)
    {}

    public virtual double GetDoubleValue()
    {
        return 0;
    }
    public virtual void SetDoubleValue(double aValue)
    {}

    public virtual bool GetBoolValue()
    {
        return false;
    }
    public virtual void SetBoolValue(bool aValue)
    {}

    public virtual string GetStringValue()
    {
        return "";
    }
    public virtual void SetStringValue(string aValue)
    {}

    public virtual System.DateTime GetDateTimeValue()
    {
        return System.DateTime.Now;
    }
    public virtual void SetDateTimeValue(System.DateTime aValue)
    {}

    public virtual void SetValue(object aValue)
    {
        SetStringValue(aValue.ToString());
    }
...
}
```

It has getters and setters for "basic" value types. The class itself contains "empty" value and represents "TypeVoid". The data containers for all supported types are derived from "CDataContainer" overriding the virtual methods to deal with specific type values.

CPropertyList Methods

CPropertyList (constructor)

Signature:

```
public CPropertyList(string aType, string aName) {...}
```

Parameters:

- **aType** (string): a type of the property (must be one of the listed above types)
- **aName** (string): property name

Description:

Creates a property list of specific type and name. Not registered type will cause an exception.

Example:

Taken from CStdMessage code: creates a root of the message

```
...  
// Root property  
public zx.pl.CPropertyList Root =  
    new zx.pl.CPropertyList(zx.pl.CDataContainer.TypeVoid, "Root");  
...
```


AddProperty

Signature:

```
public CPropertyList AddProperty(string aType, string aName) {...}
public CPropertyList AddProperty(string aName, CPropertyList aPl) {...}
public CPropertyList AddProperty(CPropertyList aPl) {...}
```

Parameters:

- **aType** (string): a type of the property (must be one of the listed above types)
- **aName** (string): property name
- **aPl** (CPropertyList): another property

Description:

Adds another property to the property list. Version "AddProperty(string aType, string aName)" creates a new property of type "aType" and adds it. Version "AddProperty(string aName, CPropertyList aPl)" adds existing property under name "aName". Note: "aName" can be different from "aPl.GetName()". Version "AddProperty(CPropertyList aPl)" adds existing property using "aPl.GetName()".

If the property list already contains a property with the specified name, the existing property is removed first, then the new one is added.

If property name is empty (""), an auto-generated name will be provided by method "GetUniqueName()".

Examples:

Adding a property of "TypeUID" and assigning a new generated GUID:

```
...
    var fldUid = msg.Root.AddProperty(zx.pl.CDataContainer.TypeUID, "uid");
    fldUid.DC.SetStringValue(System.Guid.NewGuid().ToString());
...
```

Adding existing property:

```
...
    var prop = pl.AddProperty(zx.pl.CDataContainer.TypeLong, "val");
    msg.Root.AddProperty(prop); // same as msg.Root.AddProperty("val", prop)
...
```

AddPropertyValue

Signature:

```
public CPropertyList AddPropertyValue(string aName, int aValue) {...}
public CPropertyList AddPropertyValue(string aName, double aValue) {...}
public CPropertyList AddPropertyValue(string aName, bool aValue) {...}
public CPropertyList AddPropertyValue(string aName, string aValue) {...}
public CPropertyList AddPropertyValue(string aName, System.DateTime aValue)
{...}
```

Parameters:

- **aName** (string): property name
- **aValue** (*): property value of different types

Description:

Adds a new property to the property by name and value. A type of added property is defined by one of the overloaded methods. The value is assigned to the added property.

Examples:

```
...
// Prepare input parameters
zx.pl.CPropertyList in_params = new zx.pl.CBranch();
// adds TypeLong property because aCount is 'int'
in_params.AddPropertyValue(PARAM_LOG_Count, aCount);
// adds TypeString property because aLang is 'string'
in_params.AddPropertyValue(PARAM_LOG_Lang, aLang);
...
```

AddBranch

Signature:

```
public CPropertyList AddBranch(string aName) {...}
public CPropertyList AddBranch() {...}
```

Parameters:

- **aName** (string): branch name

Description:

Adds a new property of TypeBranch to the property. Method's implementation tells for itself:

```
public CPropertyList AddBranch(string aName)
{
    return AddProperty(CDataContainer.TypeBranch, aName);
}
```

There is even a helper class "CBranch" derived from "CPropertyList" designed to simplify branches creation:

```
public class CBranch : CPropertyList {
public
    CBranch(string aName) : base(CDataContainer.TypeBranch, aName) {}
public
    CBranch() : base(CDataContainer.TypeBranch, "") {}
}; // CBranch
```

Examples:

```
...
    // Prepare input parameters
    var root = new zx.pl.CBranch(); //
    var sub = root.AddBranch("Sub");
    sub.AddPropertyValue("num", 123);
    return root;
...

```

FindProperty

Signature:

```
public CPropertyList FindProperty(string aName) {...}
```

Parameters:

- ***aName*** (string): branch name

Description:

Returns a property by name. If property with such a name does not exist, returns null.

Examples:

```
...  
var pl = inParams.FindProperty("score");  
if (pl == null)  
    throw new System.Exception("parameter not found");  
...
```

FindPropertyV

Signature:

```
public CPropertyList FindPropertyV(string aName) {...}
```

Parameters:

- ***aName*** (string): branch name

Description:

Returns a property by name. If property with such a name does not exist, returns a property with type void. This method never returns null. It's convenient when you fill up some output data and if some expected fields don't exist, you just put empty data. It saves lot's of code by checking for null in such cases.

Implementation is simple (pretty old c# to keep it compatible with everything – actually written long ago...):

```
public CPropertyList FindPropertyV(string aName)
{
    CPropertyList p = FindProperty(aName);
    return p == null ? new CPropertyList(CDataContainer.TypeVoid, aName) : p;
}
```

Examples:

```
...
    row[dataColumnID] =
        rec.FindPropertyV(zx.matrix.log.CClient.LOGFLD_MessageID).DC.GetLongValue();
    row[dataColumnText] =
        rec.FindPropertyV(zx.matrix.log.CClient.LOGFLD_MessageText).DC.GetStringValue();
...
```

GetProperty

Signature:

```
public CPropertyList GetProperty(string aName) {...}
```

Parameters:

- **aName** (string): branch name

Description:

Returns a property by name. If property with such a name does not exist, throws exception. This method is handy when a property with specific name must exist, and if it isn't, then operation should fail.

Implementation is simple:

```
public CPropertyList GetProperty(string aName)
{
    var pl = FindProperty(aName);
    if (pl == null)
        throw new System.Exception(string.Format("Property '{0}' has been not found", aName));
    return pl;
}
```

Examples:

```
...
static bool Method_SendMessages(
    zx.pl.CPropertyList inParameters,
    zx.pl.CPropertyList outParameters,
    zx.CError outError)
{
    // Get a number of messages to send
    var number = inParameters.GetProperty("Number").DC.GetLongValue();
...
}
```

DeleteProperty

Signature:

```
public void DeleteProperty(string aName) {...}
```

Parameters:

- **aName** (string): branch name

Description:

Removes a property by name. Returns nothing.

Implementation is trivial:

```
public void DeleteProperty(string aName)
{
    mList.Remove(aName);
}
```

Example:

```
public CPropertyList AddProperty(string aName, CPropertyList aPl)
{
    // Remove a property with the same name first
    DeleteProperty(aName);
    // Add it to the list
    mList.Add(aName, aPl);
    return aPl;
}
```

GetEnumerator

Signature:

```
public IDictionaryEnumerator GetEnumerator() {...}
```

Description:

Returns property list enumerator. Useful when it's necessary to loop over all internal properties.

Example:

Taken from ftMonitor (see *ExampleProjects/dcnMonitor*)

```
// Looking for a connected component in the system info.
// Returns a component desc or null, if not found
public CPlComponentDesc FindComponent(string aFldName, string aFldValue)
{
    IDictionaryEnumerator iter = Components.GetEnumerator();
    while (iter.MoveNext())
    {
        CPlComponentDesc component = new CPlComponentDesc((zx.pl.CPropertyList)iter.Value);
        if (component.FindPropertyV(aFldName).DC.GetStringValue() == aFldValue)
            return component;
    } // while
    return null;
}
```


Count

Signature:

```
public int Count() {...}
```

Description:

Returns a number of sub-properties.

Example:

Taken from ftMonitor (see *ExampleProjects/dcnMonitor*)

```
...
    if (CMatrixSubsystem.SynchRequest(
        aComponentName: mManagerName,
        aObjectPath: CFG_MANAGER_OBJECT_NAME,
        aMethodName: METHOD_CheckNames,
        aInParameters: in_params,
        outParameters: ref out_params,
        aTimeOutMs: aTimeOutMs,
        outError: ref outError,
        aPriority: zx.matrix.TPriority.PRIORITY_HIGHEST))
    {
        // Executed Ok
        if (out_params.Count() != 0)
        {
            // Some errors found
            CMatrixSubsystem.Trace("CheckNames: FOUND " + out_params.Count().ToString() + " errors");
        }
    }
...

```

Constructing, sending and receiving Messages

Use "CStdMessage" to construct any data you'd like to send via network.

"CStdMessage" has only one member – property list "Root". All you need to do is to add necessary properties/branches to the Root, and... send the message. The other end will receive the message with all Root content.

For every created "CStdMessage" you should specify message "Type" and "Category". The recipient side should have a subscription to the correspondent "Type" and "Category" to process the message.

Examples:

One application constructs and sends a bunch of messages of type "TestMsg" and category "Test":

```
...
    // Send messages of type "test".
    // Each message contains one string field with info
    var msg = new zx.matrix.CStdMessage("TestMsg", "Test");
    var fldInfo = msg.Root.AddProperty(zx.pl.CDataContainer.TypeString, "Info");
    for (var i = 0; i < number; ++i)
    {
        fldInfo.DC.SetStringValue(string.Format("This is message {0} from {1}...", i + 1, number));
        zx.matrix.CMatrixSubsystem.Send(msg);
    } // for...
```

Another application has a subscription for this message:

```
...
    zx.matrix.CMatrixSubsystem.Subscribe(
        new zx.matrix.CStdMessage("TestMsg", "Test"),
        new zx.matrix.ProcessHandle(MsgHandler_TestMsg));
...

```

Here "MsgHandler_TestMsg" looks so:

```
...
private static void MsgHandler_TestMsg(zx.matrix.CMessage aMsg, zx.matrix.CAddress aAddr)
{
...
    // Decerialize the message
    var msg = zx.matrix.CStdMessage.Deserialize(aMsg);
    Console.WriteLine(
        string.Format(
            "==> Received message {0}/{1} from {2}. Info: '{3}'",
            NumberOfReceivedMessages, NumberOfRequestedMessages,
            aAddr.GetFullName(),
            msg.GetFullType(),
            msg.Root.GetProperty("Info").DC.GetStringValue());
...

```

Note the "Deserialize" call: each message must be explicitly de-serialized as "zx.matrix.CStdMessage" from the abstract "zx.matrix.CMessage" object. Only after that you'll have access to the member "Root" with all data sent by the sender.

Using MATRIX in Unity

Initially, the MATRIX framework has been designed to be used in real production environment to communicate different C# based application between the servers and each other. This has worked successfully over the years.

But nothing really stops using it for gaming – it's simple and reliable.

Initializing the MATRIX in your game

The MATRIX framework should be started once (any attempt to start it again won't work and will be ignored). I've found a reliable place to start the MATRIX in a static method with attribute "[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]".

Here is a real example how to initialize the MATRIX in your Unity project:

```

////////////////////////////////////
// Matrix initialisation (once per process)
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
public static void InitializeMatrix()
{
    const string strDisconnected = "DISCONNECTED";
    System.Action setDCNDisconnectedInfo =
        () => stDCNConnectionUpdater.Set(
            "DCN: " + strDisconnected, unx.Tools.FadeColour(Color.red, 1.5f));
    setDCNDisconnectedInfo(); //<-- setting initial status Disconnected
    SetGameServerStatusInfo(strDisconnected, zx.matrix.TComponentState.STATE_ERROR);
    try
    {
        GU.CReportFactory.Get().TraceInfo(
            string.Format("STARTING MATRIX with name '{0}'...", ProductName));
        zx.matrix.CMatrixSubsystem.Start(ProductName);
        GU.CReportFactory.Get().TraceInfo("...Ok");

        // Here I set up default connection setting for the game, so the player could
        // be able to connect to the communicator without any troubles
        // Set up default configuration
        var hostAliases = zx.matrix.CMatrixSubsystem.GetCfgItem(CfgItemPath_DCNNostAliases);
        if (string.IsNullOrEmpty(hostAliases))
        {
            GU.CReportFactory.Get().TraceInfo("Setting up default configuration for MATRIX...");
            zx.matrix.CMatrixSubsystem.SetCfgItem(CfgItemPath_DCNAvailableHosts, DefaultAvailableHosts);
            zx.matrix.CMatrixSubsystem.SetCfgItem(CfgItemPath_DCNNostAliases, DefaultHostsAliases);
            zx.matrix.CMatrixSubsystem.ForceSaveConfig();
            GU.CReportFactory.Get().TraceInfo("...Ok");
        }

        // Here are subscriptions for connected/disconnected events to update
        // the status on the screen
        // Subscribe to connection events
        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CEventConnected(),
            new zx.matrix.CMethodMessageHandler(
                (msg, sender) =>
                {
                    GU.CReportFactory.Get().TraceInfo(string.Format(
                        "+++++++DCN CONNECTED TO '{0}'...",
                        zx.matrix.CMatrixSubsystem.GetMasterAddress().GetFullName()));
                    stDCNConnectionUpdater.Set(
                        string.Format(
                            "DCN: Connected to '{0}'",
                            zx.matrix.CMatrixSubsystem.GetMasterAddress().GetFullName()),
                        unx.Tools.FadeColour(Color.green, 1.5f));
                }
            ),
            aSubscriptionType: zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);

        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CEventDisconnected(),
            new zx.matrix.CMethodMessageHandler(
                (msg, sender) =>
                {
                    GU.CReportFactory.Get().TraceInfo("- - - - - DCN DISCONNECTED");
                    setDCNDisconnectedInfo();
                }
            ),
            aSubscriptionType: zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);

        // Catching "component state changed" event to update its status on the screen
        zx.matrix.CMatrixSubsystem.Subscribe(
            new zx.matrix.CEventComponentComponentStateChanged(),
            new zx.matrix.CMethodMessageHandler(

```

```

(msg, sender) =>
{
    var ev = new zx.matrix.CEventComponentComponentStateChanged(msg);
    ev.Deserialize();
    if (ev.ComponentDesc.Name == GameServerName)
    {
        if (ev.OldState != ev.NewState)
            Debug.Log(string.Format(
                "{0}' state changed from '{1}' to '{2}'",
                GameServerName,
                zx.matrix.SysComponentStateNames.Find(ev.OldState),
                zx.matrix.SysComponentStateNames.Find(ev.NewState)));
        else
            Debug.Log(string.Format(
                "{0}' state is '{1}'",
                GameServerName, zx.matrix.SysComponentStateNames.Find(ev.NewState)));

        SetGameServerStatusInfo(
            zx.matrix.SysComponentStateNames.Find(ev.NewState),
            (zx.matrix.TComponentState)ev.NewState);
    }
    }},
    aSubscriptionType: zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);
} // try
catch (System.Exception e)
{
    GU.CReportFactory.Get().TraceError(
        string.Format("Failed to start MATRIX, reason: {0}", e.Message));
}
}

```

In my Unity app the Communicator Status and Server status are displayed in the bottom left corner of the screen:



"DCN" - is actually "the communicator". It shows that the app is already connected to the communicator in address "ubuntu/1024" (host name/port).

"Galaxy Kings Server" - is my game server. It's not ready yet and it's status shown as "DISCONNECTED".

A few moments later the picture changes to this:



One can see that the MATRIX received a server initialised status and event handler has updated it on the screen.

It's very trivial way to show the status and definitely can be done better.

A good place to disconnect from the MATRIX gracefully when you quit your application is in "OnApplicationQuit" Unity overridden method:

```
private void OnApplicationQuit()
{
    Debug.Log("STOPPING MATRIX...");
    zx.matrix.CMatrixSubsystem.Stop();
}
```

It guarantees that the "Stop" will be called once when you quit.


```

        outParameters: ref outParams,
        outError: ref err,
        aTimeoutMs: 20000));
    } // Action
}, // ActionDesc

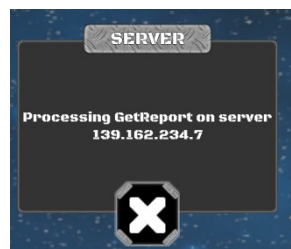
// Waiting
new unx.Tools.CActionDesc()
{
    Action = () =>
    {
        if (msgBox.GetMessage() != message)
            msgBox.ChangeMessage(message);
    },
    HasFinished = () => mre.WaitOne(100, true) // looks as a good delay
},

// Processing result
new unx.Tools.CActionDesc()
{
    Action = () =>
    {
        gk.HideProgressBar();
        if (err.Ok())
        {
            aResponseAction(outParams);
        }
        else
        {
            gk.GetComponent<MessageBox>().ShowError(
                aParent: unx.Tools.FindCanvas().transform,
                aMessage: err.ErrorMessage,
                aOnOk: () => gk.ShowMainMenuIfNoReport());
        } // else (error)
    } // Action
}
},
aTime: new WaitForSeconds(0.05f)); // this delay seem to be working fine
},
onCancel: () => gk.ShowMainMenuIfNoReport());
}

```

"ServerSynchRequest" is a wrapper showing a message box on the screen and running a set of actions in coroutine. What it does is:

- ask a question confirming a request to the game server (ok, in my case the request process can take a long time due to the game specific, so I warn the players before committing to it)
- When the player confirms the choice by clicking "Ok", it shows a message box "Processing whatever on the server":



- In the next action it calls "zx.matrix.CMatrixSubsystem.SynchRequest" wrapped in asynchronous call "zx.Functions.AsynchCall" (see below) which returns a manual reset event "mre".
- Next coroutine action is waiting in a loop till the manual event is set. It will be set by "zx.Functions.AsynchCall" as result of timeout or a successful return of "SynchRequest".


```
        Action = x, Time = aTime, DelayKind = aDelayKind
    }).ToArray(), aTime));
}

// Runs single action
public static void RunCoroutine(
    MonoBehaviour aMono, System.Action aAction,
    YieldInstruction aTime = null,
    EDelayKind aDelayKind = EDelayKind.AfterAction)
{
    RunCoroutine(aMono, new System.Action[] { aAction }, aTime, aDelayKind);
}

// Helper: actually runs actions
private static IEnumerator ExecuteActions(CActionDesc[] aActions, YieldInstruction aTime)
{
    foreach (var actionDesc in aActions)
    {
        // Action execution loop
        while (true)
        {
            if (actionDesc.DelayKind == EDelayKind.BeforeAction)
                yield return actionDesc.Time ?? aTime;
            actionDesc.Action();
            if (actionDesc.DelayKind == EDelayKind.AfterAction)
                yield return actionDesc.Time ?? aTime;
            if (actionDesc.HasFinished == null || actionDesc.HasFinished())
                break; // current action has finished
        } // while
    } // foreach
} // ExecuteActions()
```

Message subscriptions

I don't have examples for message subscriptions done in Unity yet (apart from subscriptions to the events) like here:

```

...
zx.matrix.CMatrixSubsystem.Subscribe(
    new zx.matrix.CEventConnected(),
    new zx.matrix.CMethodMessageHandler(
        (msg, sender) =>
        {
            GU.CReportFactory.Get().TraceInfo(string.Format(
                "+++++++DCN CONNECTED TO '{0}'...",
                zx.matrix.CMatrixSubsystem.GetMasterAddress().GetFullName()));
            stDCNConnectionUpdater.Set(
                string.Format(
                    "DCN: Connected to '{0}'",
                    zx.matrix.CMatrixSubsystem.GetMasterAddress().GetFullName()),
                unx.Tools.FadeColour(Color.green, 1.5f));
        }
    ),
    aSubscriptionType: zx.matrix.CMatrixSubsystem.ESubscriptionType.Event);
...

```

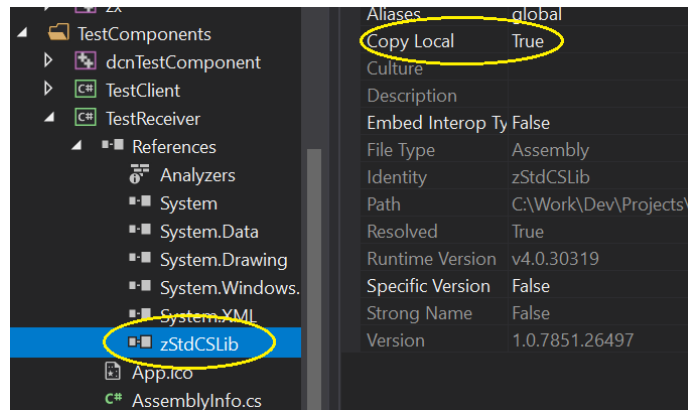
It should be pretty much the same, only subscription type should be "zx.matrix.CMatrixSubsystem.ESubscriptionType.Message".

There is an important thing to mention however: the event handlers passed to the MATRIX will be called not in the Unity main execution thread, therefore it is not possible to manipulate the game objects directly from these handlers. To do something with the game objects, the message handlers should possibly launch coroutines or trigger some other events.

Real examples of message handling are expected in the next release of the MATRIX.

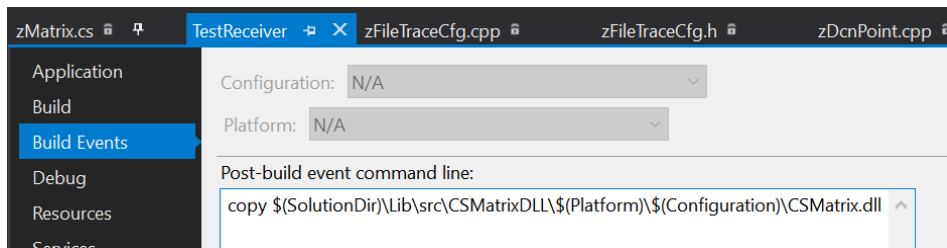
Using MATRIX in Visual Studio

To use MATRIX in your Visual Studio project you need to add a reference to "zStdCSLib.dll":

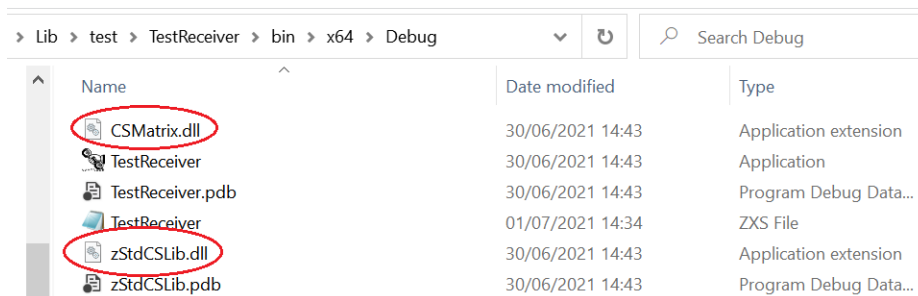


Set "Copy Local" property to "true", so your assembly will have the dll in the same location.

It's also a good idea to set up a post build event in the project settings to automatically copy the "CSMatrix.dll" into the binary folder, like here:



Here I make sure, that "CSMatrix.dll" is always copied into destination folder. After a successful build the destination folder will look like here:



...both dlls are here and you can simply start the application (TestReceiver here).

Communicator

The Communicator program **dcnNode.exe** is located in *Applications/Communicator* folder.

You can simply run it as a console application:

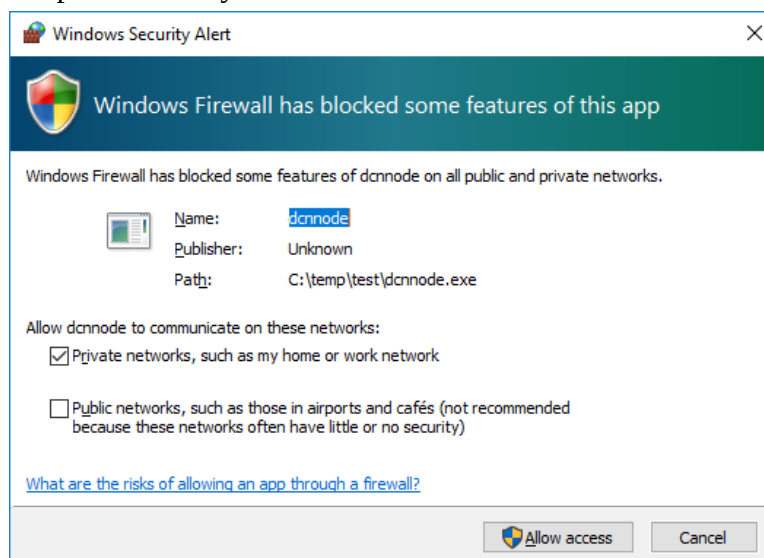
```

E:\Dev\MatrixClientServer\Bin\EXE\dcnNode.exe
17:15:11.400|i|#0:1|
TRACE CONFIGURATION
=====
TRACE PERIOD HAS CHANGED: 24 --> 24 (hours)
17:15:11.401|i|#0:1|
TRACE CONFIGURATION
=====
FILE TIME LIMIT HAS CHANGED: 60 --> 60 (minutes)
17:15:11.401|i|#0:1|
TRACE CONFIGURATION
=====
FILE LINES LIMIT HAS CHANGED: -1 --> 2147483647
17:15:11.409|i|#0:1|The list of hosts aliases is empty
17:15:11.411|i|(0-0/0)|//|#999|WF-00200|The service has started in CONSOLE mode
17:15:11.411|i|To stop, press 'q'...
17:15:13.504|E|(0-0/0)|//|#0:1|-c->DESKTOP-243DN0E/1025...|CComponent::AddConnection: !!! C
annot connect to 'DESKTOP-243DN0E/1025' for connection 0
17:15:14.609|E|(0-0/0)|//|#0:1|-c->DESKTOP-243DN0E/1026...|CComponent::AddConnection: !!! C
annot connect to 'DESKTOP-243DN0E/1026' for connection 0
17:15:15.718|E|(0-0/0)|//|#0:1|-c->DESKTOP-243DN0E/1027...|CComponent::AddConnection: !!! C

```

It's waiting for incoming connections and periodically checking other communicators running on the same host. Press 'q' to stop the console run gracefully.

Your Fire Wall may complain this way:



Just allow access and carry on.

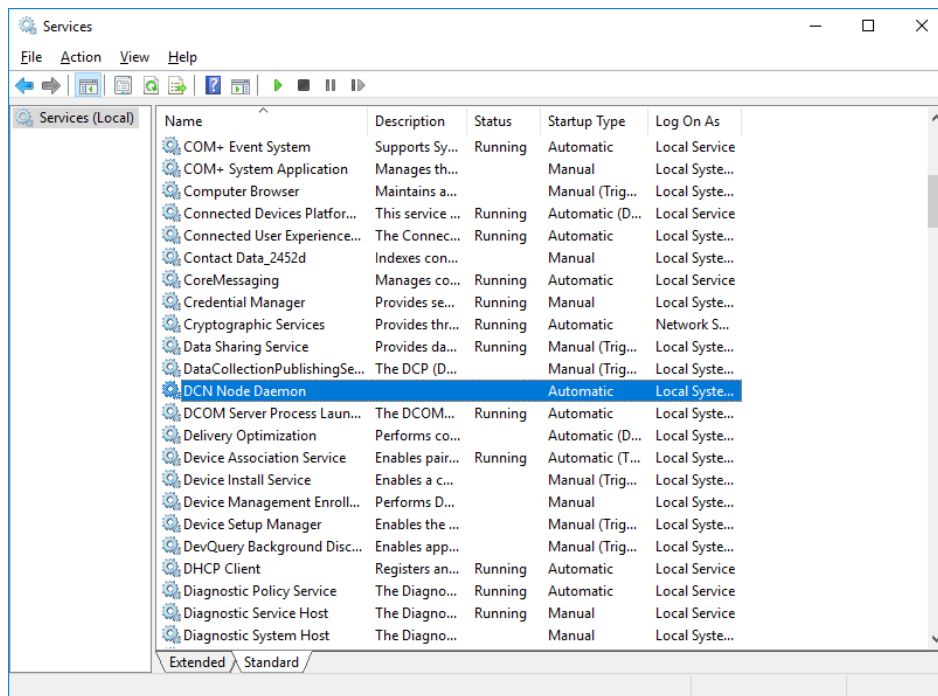
On the real environment it makes sense to install the communicator as a Windows Service to make it start automatically along with the host. To display command line options run “dcnNode -h”:

```
E:\Dev\MatrixClientServer\Bin\EXE>dcnNode.exe -h
Usage: dcnNode
-I [service name (default=all)] Service to install
-U [service name (default=all)] Service to uninstall
-V Version info
-P [service name:] [<param_name=param_value>, <param_name=param_value>...] Display/set
paramers
-PFORCE works only with -p: forces creatring cfg item if does not exist
-CFGEXP <filename[,destbranch[,srcbranch]] Exports configuration from <srcbranch> in a text
file <filename> in file's branch <destbranch>>
-CFGIMP <filename[,destbranch[,srcbranch]] Imports configuration from <srcbranch> from a
text file <filename> in configuration branch <destbranch>>
-H Help
```

To install a windows service simply type in command prompt “dcnNode -i” (make sure you are running the command prompt as Administrator):

```
C:\Temp\Test>dcnNode.exe -i
DCN Node Daemon has been installed sucessfully
```

In the Services you’ll see:



Now, you can start/stop the communicator using this terrible app. Because it is set “Automatic” the service will start every time you reboot the host.

To uninstall the communicator service do the following:

```
C:\Temp\Test>dcnNode.exe -u
DCN Node Daemon has been removed sucessfully
```

Test Components

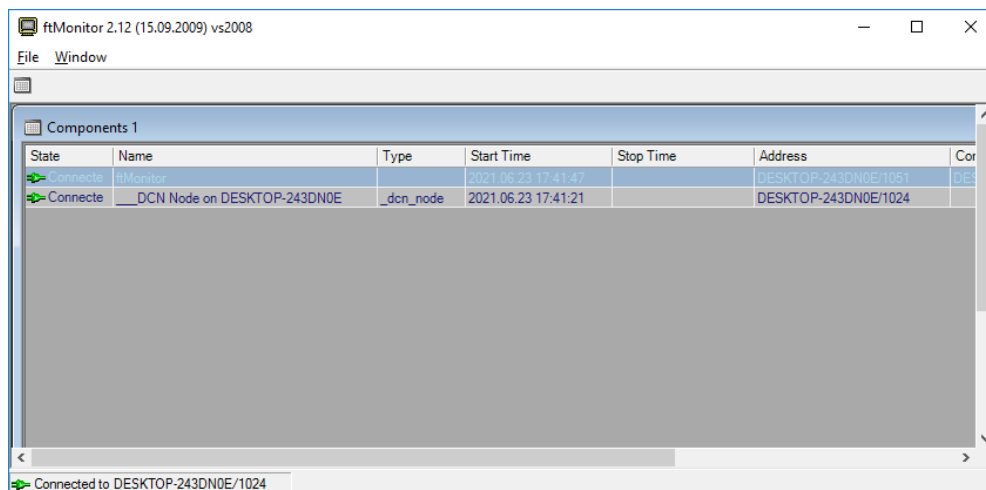
In the folder *Applications/TestComponents* you can find a number of test components which can be used to check the communicator is working correctly.

Run the communicator (dcnNode.exe – either as a console or as a service – doesn't matter).

Run Monitor.exe:

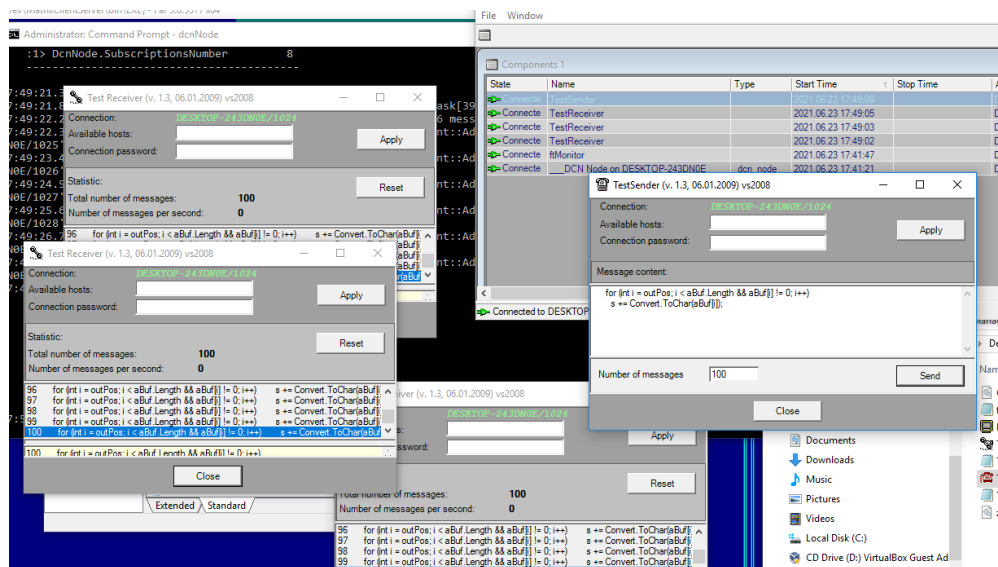


Monitor is a simple test program which shows all the components running in the current environment. On the bottom left one can see a connection status with the communicator running on the "localhost". Click "List of Components button":



One can see two components are currently running on the system: the Monitor itself (ftMonitor) and the communicator (__DCN Node...).

In the *TestComponents* folder there are also two simple applications TestSender and Test Receiver. Run several instances of each:



Here I run 3 TestReceiver's and one TestSender. You can click button 'Send' on TestSender and see all receivers have received 100 messages. You can also see that the monitor displays all the running components.

The test components are useful to check connectivity of your real client-server environment. When you have a communicator and game server running on a remote host, you may run the Monitor, Sender and Receiver on your client host, and see how good the message flow is.

The source code of all the test components can be found in the *ExampleProjects* folder for reference.

For testing purposes there is a communicator running on the Linode machine with IP address "139.162.234.7". You can try to configure your test components to communicate via this remote communicator. The communication settings should be like these:

```
...
section DCN
{
  item "AvailableHosts",      "localhost,139.162.234.7";
  item "ForceInitialized",   "false";
  item "HostsAliases",       "ubuntu:139.162.234.7";
  item "IsBroadcastAvailable", "false";
  item "PortsForComponents", "1051-1150";
  item "PortsForNodes",     "1024-1030";
  item "ScanNetHostsTimerMin", "60";
  item "ScanPortsRate",     "15";
} // DCN
...
```

There is "localhost" in "AvailableHosts" too just to make the component to connect local host first, and if not found – go to the linode host.

Do not use the test host for your games – it's available not on permanent basis, and only as an example! This address may change.

BTW, Linode may be a good choice to host your game server remotely (see <https://www.linode.com/pricing/>). I am fine with \$10 a month option to host a linux (ubuntu) machine. Initially, I hosted my game on my home computer, but found not good running it in 24/7 mode...

Configuration

Any MATRIX component has a configuration file located in the same folder where the executable is. The file has a name in following format "<ComponentName>.zxs", where "<ComponentName>" is based on the name of the component given by "zx.matrix.CMatrixSubsystem.Start" call. For example "zx.matrix.CMatrixSubsystem.Start("My Component")" will generate "My_Component.zxs" configuration file.

If doesn't exist, the configuration file is created automatically with default settings when the application starts.

```
C:\Temp\DCNTest>dir
Volume in drive C is OS
Volume Serial Number is 9A5D-7A99

Directory of C:\Temp\DCNTest

10/07/2021  17:39  <DIR>          .
10/07/2021  17:39  <DIR>          ..
30/06/2021  15:21             11,754,496  CSMatrix.dll
30/06/2021  15:21             11,155,456  dcnNode.exe
30/06/2021  15:28                957 dcnNode.zxs
30/06/2021  15:21             11,506,176  dcnTestComponent.exe
30/06/2021  15:28                1,115 dcnTestComponent.zxs
30/06/2021  14:43             7,168 TestClient.exe
10/07/2021  17:39                211 TestClient.zxs
30/06/2021  14:43             6,144 TestServer.exe
10/07/2021  17:39                211 TestServer.zxs
30/06/2021  14:43             69,632 zStdCSLib.dll
```

After creation the configuration file may have the following content, for example:

```
////## AUTO GENERATED CONFIGURATION SECTION START >>>
// Ver. *** 1.0.2.4 27.03.2008 ***
// Generation Time: 30.06.2021 15:28:53
section Root
{
  // DCN Connection settings
  section DCN
  {
    item "AvailableHosts",      "localhost";
    item "ForceInitialized",    "false";
    item "HostsAliases",       "";
    item "PortsForComponents",  "1051-1150";
    item "PortsForNodes",      "1024-1030";
    item "ScanNetHostsTimerMin", "60";
    item "ScanPortsRate",      "15";
  } // DCN

  // Trace output settings
  section Trace
  {
    item "LogLockIDs",          "";
    item "TraceFileLinesLimit", "2147483647";
    item "TraceFileTimeLimit",  "60";
    item "TraceInColour",       "false";
    item "TraceLevel",          "1";
    item "TraceLock",           "true";
    item "TraceLockCat",        "";
    item "TracePath",           "c:/Temp/DCNTest/DCN Test";
    item "TracePeriod",         "24";
    item "TraceStructured",     "false";
    item "TraceTimeFormat",     "hh:mi:ss.TTT";
  } // Trace
} // Root
////## AUTO GENERATED CONFIGURATION SECTION END <<<
```

It consists of "Sections" and "Items". The sections can be treated as "folders" and items as "values".

The main section is always "Root". Any section can contain any number of sub-sections and items.

Using MATRIX methods "zx.matrix.CMatrixSubsystem.GetCfgItem/SetCfgItem" one can read/write specific configuration items using a "path" programmatically, like:

```
...    zx.matrix.CMatrixSubsystem.SetCfgItem("DCN.AvailableHosts", "139.162.44.34,localhost");  
...  
...    var tracePath = zx.matrix.CMatrixSubsystem.GetCfgItem("Trace.TracePath");  
...
```

A "path" is constructed using a "dot-notation": "<section>.<subsection>.<item name>". Section "Root" is not mentioned in the path because everything is considered to be inside the Root.

Configuration items can belong directly to the Root. In this case they are addressed without any "sections" like:

```
...    var cfgTitem = zx.matrix.CMatrixSubsystem.GetCfgItem("Item1");  
...
```

The configuration files are "live", i.e. you can change the data while the component is running. The component will catch up with the changes automatically. Be careful when changing the configuration files manually – don't break the syntax! If you do, the component will fail to load the updated file and re-generate a new one with default settings! Also, the components may update the files themselves when they quit – this will overwrite your unsaved changes.

Setting up Communication

MATRIX Component Settings

The communication between the MATRIX components and the Communicator is set in "DCN" section:

```
...
// DCN Connection settings
section DCN
{
  item "AvailableHosts",      "localhost";
  item "ForceInitialized",   "false";
  item "HostsAliases",      "";
  item "PortsForComponents", "1051-1150";
  item "PortsForNodes",     "1024-1030";
  item "ScanNetHostsTimerMin", "60";
  item "ScanPortsRate",     "15";
} // DCN
...
```

- **AvailableHosts** – a list of hosts separated by comma where the MATRIX component will search the Communicator. By default – it is "localhost". Also, only connections coming from the hosts listed here will be accepted. It's possible to use "*" as a wildcard. For example "MyHost*" - all hosts started from "MyHost" will be used. Or "*" - all hosts will be used. But be extremely careful with the "*"! Using wildcards for the MATRIX component may involve too many hosts (actually all hosts found by scanning the network) or no hosts at all! The wildcard is the best suitable for the communicator, because it doesn't know where incoming connections may come from, but, normally, the client applications (like MATRIX components) know where the communicator is. Therefore, be specific – just point a host where the communicator is to avoid long connection delays.
- **ForceInitialized** – if set to "true", the component considers itself as "Initialized" and doesn't try to connect to communicator. By default it's "false", i.e. the component will try to connect to the communicator until the connection is established. Setting "**ForceInitialized**" in "true" makes sense only if the component is supposed to work "offline" and shouldn't waste efforts to find the communicator in the network.
- **HostsAliases** – a list of host aliases, separated by comma. This field is used to resolve the address in the response on connection request coming from the communicator. Example:

```
section DCN
{
  item "AvailableHosts",      "139.162.234.7";
  item "ForceInitialized",   "false";
  item "HostsAliases",      "ubuntu:139.162.234.7";
} // DCN
...
```

The problem is, when a communicator host is specified using an IP address like above, the connection response message will contain a host name, not an IP address. In this particular case, the communicator returns its host name as "ubuntu". "HostAliases" help the framework to resolve host "ubuntu" as "139.162.234.7". This problem arises only when trying to connect the communicator using IP addresses – what is the most common case when the communicator is located somewhere in the WEB. For local networks you can use host names directly and avoid bothering with host aliases.

- **PortsForComponents** – a range of ports allocated for the MATRIX components on your machine. The MATRIX framework doesn't use a single port to specify a connection point, it uses ranges instead. It's more convenient and flexible (a specific port may be busy, in this

case your component will select a free one from the range). Default value is "1051-1150", giving you 100 ports. Unless you have a strong wish to change the port range, just leave it as it is. If you want to use just one port – specify it like "1051". Examples of port ranges: "1051" - just a single port; "1051,1052" - just two ports; "1002-1010,1100-1200, 3455": ports from 1002 to 1010 and from 1100 to 1200 and port 3455.

- **PortsForNodes** – a range of ports used by the Communicator. Your MATRIX component will look for the communicator on the hosts specified by "AvailableHosts" and only on ports, specified by "PortsForNodes". Keep "PortsForNodes" range in synch with the communicator setting "PortsForNodes" (the same name) - see below.
- **ScanNetHostsTimerMin** – specifies a time in minutes to scan all available hosts in the network. It's an expensive and long operation. Default value is "60" what makes one hour. Don't touch this item unless you know what you are doing.
- **ScanPortsRate** – specifies a time in seconds to scan the ports on the available hosts to look for the communicator. Default value is "15"

Communicator Settings

The communicator settings are located in the dcnNode.zxs configuration file directly in the "Root":

```
section Root
{
  item "AvailableHosts",      "localhost";
  item "HostsAliases",       "";
  item "NodeHosts",          "";
  item "PortsForNodes",      "1024-1030";
  item "ScanNetHostsTimerMin", "60";

  // Trace output settings
  ...
}
```

- **AvailableHosts** – a list of hosts separated by comma the communicator accepts the connections from. The syntax is the same as for the MATRIX components. But, for the communicator it makes sense to use just a wildcard "*" to accept connections from any host (unless you know exactly which hosts are used by your players).
- **HostsAliases** – a list of host aliases, separated by comma. This field doesn't make sense for the communicator, leave it blank
- **PortsForNodes** – a range of ports used by the Communicator. The communicator will use the first free port of the range for its connection point. Your MATRIX components should use the same range of ports in their "PortsForNodes" setting.
- **ScanNetHostsTimerMin** – specifies a time in minutes to scan all available hosts in the network. It's an expensive and long operation. Default value is "60" what makes one hour. Don't touch this item unless you know what you are doing.

Example of the communication settings

Normally, your game server will run on the remote internet host. It will be connected by multiple clients from anywhere in the WEB. It makes sense to run the communicator (dcnNode.exe) as a Service on the same machine as the game server.

Let's assume that we use standard port range set by default. In this case the following settings will fork for you:

Communicator:

```
section Root
{
  item "AvailableHosts",      "*"; //<--- let all hosts to connect me
  item "HostsAliases",       "";
  item "NodeHosts",          "";
  item "PortsForNodes",      "1024-1030";
  item "ScanNetHostsTimerMin", "60";
  ...
}
```

Game Server:

```
section Root
{
  // DCN Connection settings
  section DCN
  {
    item "AvailableHosts",      "localhost"; //<--- I am connecting the communicator on localhost
    item "ForceInitialized",    "false";
    item "HostsAliases",       "";
    item "PortsForComponents",  "1051-1150";
    item "PortsForNodes",      "1024-1030";
    item "ScanNetHostsTimerMin", "60";
    item "ScanPortsRate",      "15";
  } // DCN
  ...
}
```

Client application:

```
// DCN Connection settings
section DCN
{
  item "AvailableHosts",      "139.162.234.7"; //<--- IP address of the communicator machine
  item "ForceInitialized",    "false";
  item "HostsAliases",       "ubuntu:139.162.234.7"; //<--- resolved host alias
  item "PortsForComponents",  "1051-1150";
  item "PortsForNodes",      "1024-1030";
  item "ScanNetHostsTimerMin", "60";
  item "ScanPortsRate",      "15";
} // DCN
...
```

If by some reason you find convenient to place the Game Server on a separate host, it's configuration will be exactly the same as for "Client Application", because from the communicator "perspective" - the game server is another "component".

Setting up Trace

All the MATRIX components and the communicator may write diagnostic information into the trace files. The trace settings are located in the section "Trace" in the configuration file (they are the same for the components and communicator):

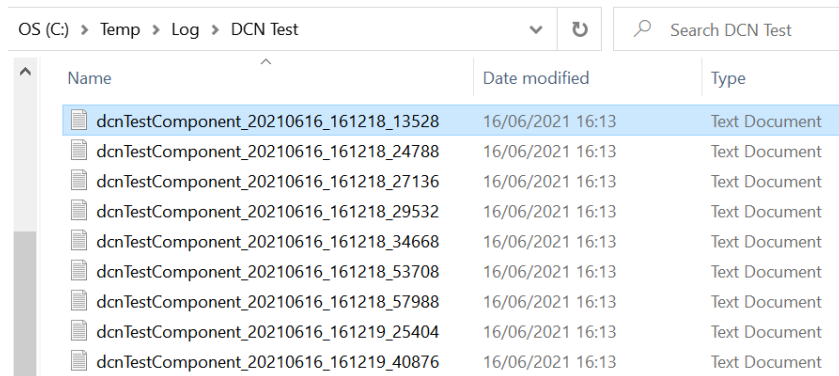
```
...
// Trace output settings
section Trace
{
  item "LogLockIDs",          "";
  item "TraceFileLinesLimit", "2147483647";
  item "TraceFileTimeLimit", "60";
  item "TraceInColour",      "false";
  item "TraceLevel",         "1";
  item "TraceLock",          "true";
  item "TraceLockCat",       "";
  item "TracePath",          "c:/Temp/DCNTest/DCN Test";
  item "TracePeriod",        "24";
  item "TraceStructured",    "false";
  item "TraceTimeFormat",    "hh:mi:ss.TTT";
} // Trace
...
```

- **LogLockIDs** – a list trace IDs to be excluded from trace. Leave it empty
- **TraceFileLinesLimit** – maximum number of lines in the trace file. Default value "2147483647" literally means "unlimited"
- **TraceFileTimeLimit** – maximum time of tracing in one file in minutes before the roll. "60" (one hour) is a default value.
- **TraceInColour** – this parameter works only for console output: if "true", the console trace is done in colour.
- **TraceLevel** – traces the messages up to specified trace level: 0 – no trace, 1 – normal, 2 – detailed, 3 – debug, 4 – detailed debug. Specifying the level larger than 2 may cause too noise trace and affect the performance.
- **TraceLock** – if "true", the file trace is locked (no file trace).
- **TraceLockCat** – locks specified trace categories (leave it untouched)
- **TracePath** – specifies a path where the trace files are created
- **TracePeriod** – specifies a live time of the trace files in hours. Default is "24" (hours). The trace files which are older than "TracePeriod" hours from now are removed automatically.
- **TraceStructured** – changes a view of trace lines if "true" to look a "structured" way.
- **TraceTimeFormat** – specifies a time output for each trace line. Default value is "hh:mi:ss.TTT" (hh – hours, mi – minutes, ss – seconds, TTT – milliseconds). In this format the time stamp looks so "19:12:23.311"

By default the MATRIX component produce no file trace (because parameter "TraceLock" is "true"). Set it to "false" and set "TraceLevel" to "1" or bigger if you'd like to see some trace. You may want to do it if you have connectivity problems and would like to know what's going on.

The trace system creates rolling trace files in the specified directory (parameter "TracePath"). The individual trace file name is constructed from component name, time of creation and process id:

```
<ComponentName>_<Creation DateTime in format YYYYMMDD_hhmiss>_<ProcessId>.log
```



The screenshot shows a Windows File Explorer window with the address bar displaying the path: OS (C:) > Temp > Log > DCN Test. The search bar contains the text "Search DCN Test". The main area displays a list of files with columns for Name, Date modified, and Type. The files are all text documents, and their names follow a pattern: dcnTestComponent_20210616_161218_XXXXXX, where XXXXXX represents a unique identifier. The date and time for all files is 16/06/2021 16:13.

Name	Date modified	Type
dcnTestComponent_20210616_161218_13528	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161218_24788	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161218_27136	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161218_29532	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161218_34668	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161218_53708	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161218_57988	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161219_25404	16/06/2021 16:13	Text Document
dcnTestComponent_20210616_161219_40876	16/06/2021 16:13	Text Document

The individual files live time is specified by parameter "TracePeriod".