

Programming Guide for Genuine Channels v2.5



Genrix

Genuine solutions for
successful development

This guide contains information about how to configure, use, customize and deploy a solution based on Genuine Channels. You are expected to know enough about the .NET Remoting technology in order to use Genuine Channels.

1. INTRODUCTION	5
2. TRANSPORT CONTEXT	8
2.1. Transport Context Hierarchy	10
2.2. Connection Manager	11
2.3. Known Hosts	11
2.4. Host Information	13
2.5. Parameter Provider	13
2.6. Direct Exchange Manager	15
2.6.1. Sending	16
2.6.2. Receiving	20
2.7. Synchronous and asynchronous transport approaches	22
2.8. Three Connection Patterns	24
2.8.1. Persistent connection	25
2.8.2. Named connection	26
2.8.3. Invocation connection	26
3. SECURITY	27
3.1. Key Providers	28
3.2. Security Session	30
3.2.1. Connection Level Security Session	31
3.2.2. Invocation Level Security Session	32
3.3. Security Session Parameters	34
3.4. Basic Security Provider	38
3.5. Known Symmetric Security Provider	38
3.6. Self-Establishing Symmetric Security Provider	39
3.7. SSPI Security Providers	39
3.8. Zero Proof Authorization Security Providers	41
3.9. Chaining mode	43
4. EVENTS IN DISTRIBUTED ENVIRONMENT	45
4.1. Broadcast Engine in practice	46
4.2. Filtering recipients	49

4.3.	Dispatcher parameters	55
4.4.	IP Multicasting	56
4.4.1.	Client Side	58
4.4.2.	Server Side	59
5.	CHANNELS	60
5.1.	GTCP	60
5.1.1.	Configuration via a configuration file	60
5.1.2.	Programmatic configuration	62
5.1.3.	GTCP Channel Parameters	64
5.1.4.	GTCP Transport Context Parameters	65
5.1.5.	Events	67
5.2.	GHTTP	69
5.2.1.	HTTP handler	69
5.2.2.	Configuration via a configuration file	70
5.2.3.	Programmatic configuration	71
5.2.4.	GHTTP Channel Parameters	72
5.2.5.	GHTTP Transport Context Parameters	73
5.2.6.	Events	75
5.2.7.	HTTP authentication	76
5.2.8.	If HTTP channel does not work	76
5.3.	GUdp	77
5.3.1.	Configuration via a configuration file	78
5.3.2.	Programmatic configuration	78
5.3.3.	GUdp Channel Parameters	78
5.3.4.	GUdp Transport Context Parameters	79
5.3.5.	Events	80
5.3.6.	IP Multicasting	81
5.3.7.	Configuration via a configuration file	81
5.3.8.	Programmatic configuration	82
5.4.	Shared Memory	85
5.4.1.	Configuration via a configuration file	85
5.4.2.	Programmatic configuration	86
5.4.3.	Shared Memory Channel Parameters	87
5.4.4.	Shared Memory Transport Context Parameters	87
5.4.5.	Events	88
5.5.	GXHTTP Channel	89
5.5.1.	Configuration via a configuration file	89
5.5.2.	Programmatic configuration	90
5.5.3.	GXHTTP Channel Parameters	91
5.5.4.	GHTTP Transport Context Parameters	92
5.5.5.	Events	93
6.	CHANNEL EVENTS	95
7.	QUEUING	97
8.	LOGGING	98
9.	CLIENT SESSION	100
10.	THREAD POOLING STRATEGIES	102

11.	CHANNEL PREFIX -----	104
12.	COMPRESSION -----	105
13.	DEPLOYMENT -----	106
14.	ERROR MESSAGES -----	107
15.	CHAT SAMPLE -----	112

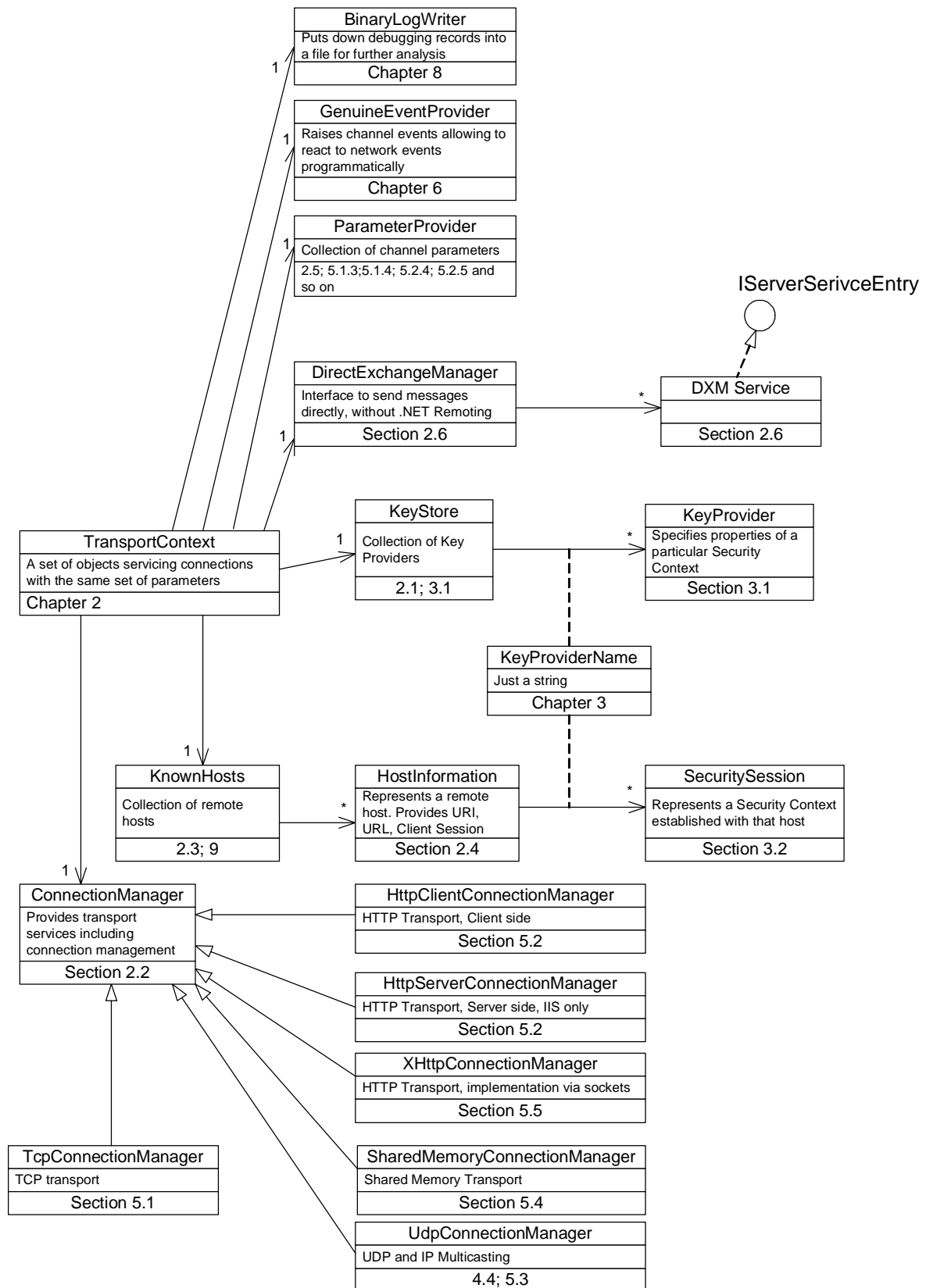
1. Introduction

Initially, Genuine Channels were designed as a substitution for the native channels distributed with Microsoft .NET Framework. In the course of time new features and capabilities were added and Genuine Channels were completely redesigned. Now the Genuine Channels Product is, first of all, a high-end transport environment accessible through .NET Remoting and Direct eXchange Manager (DXM) API. .NET Remoting allows you to invoke remote services as if they were usual objects. It hides serialization and service management beneath a small set of handy methods. The DXM interface is a socket-like interface providing maximum performance when you need it. The DXM interface also provides a compact and convenient API, but it leaves serialization stuff to you.

When you use Genuine Channels, you will use .NET Remoting and Transport Context's objects. This document does not describe .NET Remoting, please refer to MSDN and Beginner's Guide instead. This document describes Transport Context's objects and their settings.

Channels are objects providing transport services using a certain network protocol. You need to construct only one **channel** per network protocol. Actually, each Genuine Channels' channel is a small wrapper over a Transport Context. Therefore, we can use the terms *.NET Remoting Channel* and *Transport Context* as synonyms. However, in this document we will emphasize the features of the Transport Context and, therefore, we will use *Transport Context* instead of *channel*.

The Transport Context is a set of objects representing transport connections with the same set of parameters. You can have several Transport Contexts in the same appdomain. The diagram below illustrates the most important objects of the Transport Context and refers to the sections where those objects are described.



Besides Transport Contexts and .NET Remoting channels, we will use other professional terms.

By **Client**, we usually mean the calling application domain, which initiates an invocation and receives the response.

The **Server** is the processing application domain receiving an invocation, handling it and returning the response back to the **Client**.

Usually, a host sends and receives invocations in both directions; in this case, it combines the **Client** and **Server** behavior during its lifetime.

By **Client Application**, we mean an application domain that opens a **connection** to the **server application**. Note that the words **Host** and **Application Domain** are usually synonyms because any object located outside of the current Application Domain is accessible only through Remoting.

By **server application**, we mean an application domain that accepts connections established by a **client application**.

By **MBR Object**, we mean an instance of a class inherited from the MarshalByRefObject class.

The words **URI** and **URL** are usually used to specify a destination or an object address. The Host URI is the unique identifier of the client or server application domain. It remains the same while the application domain is alive. As soon as the application is restarted, the Host URI is changed. The example of a URI: `_gtcp://538741a411394c00b40e084fe24306d7`. The object URI is a unique identifier of an MBR Object connected to .NET Remoting.

The **URL** is a string specifying the network location of a **server application**. For example, `gtcp://myDomainName:8737`.

The **Transparent Proxy** is an object that virtually exposes functionality implemented by an **MBR Object** located in a **server application**. **Transparent Proxies** are constructed and provided automatically by .NET Remoting. The **Real proxy** is an object intercepting all invocations of the corresponding **Transparent Proxy** and directing them into the client channel sink.

The **Security Key Provider** is an object describing the set of rules and algorithms providing some encryption, integrity checking, authentication, impersonation and delegation services. The **Security Session** is an object that manages a particular security algorithm between two specific hosts. A **Security Key Provider** and its **Security Sessions** are always associated with a string called **Security Key Provider Name** or **Security Session Name** respectively.

Serialization is the process of converting an *object* into a form that can be transmitted over a network. **Deserialization** is the reverse process of constructing the *object* from the received data.

A remote application domain is usually represented by an instance of the HostInformation class containing the **URI**, **URL**, **Client Session**, **Security Sessions**, and **Transport Context** of the remote host.

2. Transport Context

Each Genuine Channel uses a Transport Context. The Transport Context is a set of entities serving connections with the same network parameters. You can have any number of Transport Contexts with different settings and parameters.

You can use Transport Context services via .NET Remoting and .NET Remoting channels. In this case you should construct or declare any Genuine Channels that will construct the proper Transport Context automatically. You can think of a Transport Context as a channel with some additional functionality.

1. Every Transport Context implements the *Belikov.GenuineChannels.TransportContext.ITransportContext* interface.
2. The Transport Context does not rely on .NET Remoting functionality.
3. All services provided by the Transport Context implemented as MBR objects. You can leverage the Transport Context functionality from any appdomain and any remote host.
4. You can create any Genuine Channel and bind it to a particular Transport Context, no matter whether the Transport Context is in the current appdomain or not.
5. You can bind several channels (in different appdomains) to the same Transport Context. In this case every channel will be associated with some specific transport user name (just a string). The Transport Context will dispatch incoming requests to different channels according to the transport user name specified by the client.
6. You can use Transport Context services directly, not via .NET Remoting.

The primary task of the Transport Context is

1. To simplify the development. All Transport Contexts provide the same services, no matter what channel or transport you choose.
2. To provide convenient access to underlying data structures (see Transport Context Hierarchy).

To create a Transport Context without creating a channel, you can prepare a set of parameters and call one of the *TransportContextServices.CreateDefaultXXXContext* methods:

```
[C#]
IDictionary transportProperties = new Hashtable();

// common settings
transportProperties["MaxContentSize"] = "100000";
transportProperties["MaxQueuedItems"] = "1000";
transportProperties["MaxTotalSize"] = "10000000";
transportProperties["NoSizeChecking"] = "true";
transportProperties["Compression"] = "false";
transportProperties["SyncResponses"] = "true";
transportProperties["ConnectTimeout"] = "30000";

// security settings
transportProperties["HoldThreadDuringSecuritySessionEstablishing"] = "true";

// tcp transport settings
transportProperties["TcpReconnectionTries"] = "10";
transportProperties["TcpSleepBetweenReconnections"] = "3000";
transportProperties["TcpMaxTimeSpanToReconnect"] = "45000";
transportProperties["TcpPersistentConnectionSendPingAfterInactivity"] = "30000";
transportProperties["TcpClosePersistentConnectionAfterInactivity"] = "60000";
transportProperties["TcpSeparateReceivingThreadForPersistentConnections"] = "true";

// create the transport context
this._jTransportContext = TransportContextServices.CreateDefaultTcpContext(transportProperties, null);
```


[VB.NET]

```
Dim transportProperties As IDictionary = New Hashtable

" common settings
transportProperties("MaxContentSize") = "100000"
transportProperties("MaxQueuedItems") = "1000"
transportProperties("MaxTotalSize") = "10000000"
transportProperties("NoSizeChecking") = "true"
transportProperties("Compression") = "false"
transportProperties("SyncResponses") = "true"
transportProperties("ConnectTimeout") = "30000"

" security settings      transportProperties("HoldThreadDuringSecuritySessionEstablishing") = "true"

" tcp transport settings
transportProperties("TcpReconnectionTries") = "10"
transportProperties("TcpSleepBetweenReconnections") = "3000"
transportProperties("TcpMaxTimeSpanToReconnect") = "45000"
transportProperties("TcpPersistentConnectionSendPingAfterInactivity") = "30000"
transportProperties("TcpClosePersistentConnectionAfterInactivity") = "60000"
transportProperties("TcpSeparateReceivingThreadForPersistentConnections") = "true"

" create the transport context
Me._ITransportContext = TransportContextServices.CreateDefaultTcpContext(transportProperties, Nothing)
```

The second parameter, which is a null reference in the sample code, specifies the default transport user (a .NET Remoting channel) that will be used to dispatch incoming .NET Remoting invocations.

There are three ways to obtain an existing Transport Context.

1. All Genuine Channels implement the ***Belikov.GenuineChannels.TransportContext.ITransportContextProvider*** interface allowing you to fetch the Transport Context to which the channel was bound.

[C#]

```
ITransportContextProvider ITransportContextProvider = (ITransportContextProvider) this.IConnectionProvider.Channel;
ITransportContext iTransportContext = ITransportContextProvider.ITransportContext;
```

[VB.NET]

```
Dim ITransportContextProvider As ITransportContextProvider = _
    DirectCast(Me.IConnectionProvider.Channel, ITransportContextProvider)
Dim ITransportContext As ITransportContext = ITransportContextProvider.ITransportContext()
```

2. ***ITransportContext*** is provided as a value of the ***HostInformation.ITransportContext*** property during all events.

[C#]

```
public static void GenuineChannelsEventHandler(object sender,
                                           GenuineEventArgs e)
{
    ITransportContext iTransportContext = e.HostInformation.ITransportContext;
```

[VB.NET]

```
Public Shared Sub GenuineChannelsEventHandler(ByVal sender As Object, ByVal e As GenuineEventArgs)
    Dim ITransportContext As ITransportContext = e.HostInformation.ITransportContext
```

3. You can fetch the Transport Context of any MBR object created locally or obtained from the remote host via the ***GenuineUtility.FetchChannelUriFromMbr*** call:

[C#]

```
using Belikov.GenuineChannels;
using Belikov.GenuineChannels.Security;
```

```

using Belikov.GenuineChannels.Connection;
using Belikov.GenuineChannels.TransportContext;

string uri;
ITransportContext iTransportContext;
GenuineUtility.FetchChannelUriFromMbr( anyMBRObject, out uri, out iTransportContext);

[VB.NET]
Imports Belikov.GenuineChannels
Imports Belikov.GenuineChannels.Security
Imports Belikov.GenuineChannels.Connection
Imports Belikov.GenuineChannels.TransportContext

Dim uri As String
Dim iTransportContext As ITransportContext
GenuineUtility.FetchChannelUriFromMbr(anyMBRObject, uri, iTransportContext);

```

2.1. Transport Context Hierarchy

Entity name	Entity description
IKeyStore	A collection of Security Key Providers used in a particular Transport Context. You can add and delete Key Providers at run time. Besides Security Key Providers registered at a particular Transport Context, there are global Key Providers registered at SecuritySessionServices. Those Key Providers can be used in any Transport Context. See the Security chapter below.
Connection Manager	The Connection Manager represents a specific transport and manages all connections in the Transport Context. All connections have the same parameters. The Connection Manager provides methods to start and stop listening, send messages, close connections, provides traffic counters.
IIncomingStreamHandler	Manages a set of handlers waiting for the invocation response. The only useful feature you can find here is dispatching an exception to all handlers waiting for something from a specific remote host.
Direct Exchange Manager	If you do not want to use .NET Remoting, you will probably work with the Direct Exchange Manager. It provides a fast and efficient way to send and receive raw streams via the Connection Manager without dealing with internal Genuine Channels agreements and behavior.
KnownHosts	Provides the list of all known hosts. Each host is represented as an instance of the HostInformation class, which maintains information related to a particular remote host, such as Security Sessions (you can shut down any Security Session manually), remote host information (Uri, Url, Physical address), the Client Session associated with the remote host. You can renew host's lifetime, close all connections and release all resources related to a particular remote host.
HostIdentifier	This is the main unique identifier of the Transport Context. It is usually used for building the URI of the local host.
IParameterProvider	Represents a set of absolutely all parameters and settings associated with the Transport Context. The parameter values are not cached and any change in parameters will be applied immediately.
IGenuineEventProvider	Provides the event fired for each Transport Context event. In addition, it provides a way to fire a channel event, if you wish.

IEventLogger	An instance of the binary logger that records debugging information with respect to the Transport Context developments and warnings. You can get or set the logger and write a log entry.
SecuritySessionAttributes	The Security Session used by default in the Transport Context.

2.2. Connection Manager

Connection Manager provides transport services built over a particular network or OS means. Since release 2.5.0, we have TCP, HTTP server, HTTP client, GXHTTP, UDP and Shared Memory connection managers. HTTP server Connection Manager works only inside IIS as an ASP.NET application.

You can use Connection Manager to start and/or stop listening to particular local end points:

```
[C#]
tcpConnectionManager.StartListening("gtcp://0.0.0.0:8788");
// ...
tcpConnectionManager.StopListening("gtcp://0.0.0.0:8788");

[VB.NET]
tcpConnectionManager.StartListening("gtcp://0.0.0.0:8788")
" ...
tcpConnectionManager.StopListening("gtcp://0.0.0.0:8788")
```

ConnectionManager.ReleaseConnections is a very powerful method closing all connections that come under specified connection parameters.

```
[C#]
// This call will shut down absolutely all opened connections.
// But it does not release resources associated with the remote host
// and it does not eliminate established Security Sessions.
ConnectionManager.ReleaseConnections(null, GenuineConnectionType.All, new ApplicationException("Manual shutting down.));

// This call will shut down all connections to the specific remote host by its URL.
HostInformation remote = iTransportContext.KnownHosts.Get(serverUrl);
ConnectionManager.ReleaseConnections(remote, GenuineConnectionType.All,
    new ApplicationException("Manual shutting down.));

[VB.NET]
" This call will shut down absolutely all opened connections.
" But it does not release resources associated with the remote host
" and it does not eliminate established Security Sessions.
ConnectionManager.ReleaseConnections(Nothing, GenuineConnectionType.All,
    New ApplicationException("Manual shutting down.))

" This call will shut down all connections to the specific remote host by its URL
Dim remote As HostInformation = iTransportContext.KnownHosts.Get(serverUrl)
ConnectionManager.ReleaseConnections(remote, GenuineConnectionType.All,
    New ApplicationException("Manual shutting down.))
```

2.3. Known Hosts

KnownHosts is a collection containing all known (available) remote hosts. Each remote host is an instance of the *HostInformation* class.

You can **get information about some specific host by its URI**:

```
[C#]
remote = iTransportContext.KnownHosts[GenuineUtility.CurrentRemoteUri];

[VB.NET]
remote = iTransportContext.KnownHosts(GenuineUtility.CurrentRemoteUri)
```

If you want to **get information about a host by its URL**, always call the GenuineUtility.Parse

method to build the correct URL of the remote host:

```
[C#]
// Correct
string ignored;
string url = GenuineUtility.Parse(url, out ignored);
HostInformation hostInformation = iTransportContext.KnownHosts.Get(url);

// WRONG!!!
remote = iTransportContext.KnownHosts.Get("gtcp://127.0.0.1:8737");

[VB.NET]
" Correct
Dim ignored As String
Dim url As String = GenuineUtility.Parse(url, ignored)
Dim hostInformation As HostInformation = iTransportContext.KnownHosts.Get(url)

' WRONG!!!
remote = iTransportContext.KnownHosts.Get("gtcp://127.0.0.1:8737")
```

To get the full list of all known hosts call the `KnownHosts.GetKnownHosts` method (every call leads to a lookup through the hashtable, so do not use it too often):

```
[C#]
ArrayList hosts = iTransportContext.KnownHosts.GetKnownHosts();

[VB.NET]
Dim hosts As ArrayList = iTransportContext.KnownHosts.GetKnownHosts()
```

To close all connections and release all resources associated with a specific remote host, call the `ReleaseHostResources` method:

```
[C#]
// For example, you have an MBR object implementing the IStringExchange
// interface.
// The object has been obtained from the remote host or created locally
// via the Activator.GetObject method.
string uri;
ITransportContext iTransportContext;

// get Transport Context and remote host's URI.
GenuineUtility.FetchChannelUriFromMbr((MarshalByRefObject) iStringExchange, out uri, out iTransportContext);
HostInformation remoteHost = iTransportContext.KnownHosts.Get(uri);

// And make Transport Context to close all opened connections and
// release all resources associated with this remote host.
// The specified exception will be delivered to all callers dependent
// on this remote host and to all event subscribers.
iTransportContext.KnownHosts.ReleaseHostResources(remoteHost, GenuineExceptions.Get_Receive_ConnectionClosed());

[VB.NET]
" For example, you have an MBR object implementing the IStringExchange
" interface.
" The object has been obtained from the remote host or created locally
" via the Activator.GetObject method.
Dim uri As String
Dim ITransportContext As ITransportContext

" get Transport Context and remote host's URI.
GenuineUtility.FetchChannelUriFromMbr(DirectCast(iStringExchange, MarshalByRefObject), uri, ITransportContext)
Dim remoteHost As HostInformation = ITransportContext.KnownHosts.Get(uri)

" And make Transport Context to close all opened connections and
" release all resources associated with this remote host.
" The specified exception will be delivered to all callers dependent
" on this remote host and to all event subscribers.
ITransportContext.KnownHosts.ReleaseHostResources(remoteHost, GenuineExceptions.Get_Receive_ConnectionClosed())
```

2.4. Host Information

The *HostInformation* class keeps all information regarding a specific remote host.

1. It implements the *ITransportContextProvider* interface. Therefore, you can always get the Transport Context the remote host is available through.
2. You can get or delete a Security Session by means of the *HostInformation.GetSecuritySession* and *HostInformation.DestroySecuritySession* methods.
3. An instance of the *HostInformation* class contains a Client Session for the remote host. See the Client Session section below.
4. The URI and URL of the remote host are available through the *HostInformation.Uri* and the *HostInformation.Url* properties.
5. The physical address of the remote host is available through the *HostInformation.PhysicalAddress* property. The GTCP Transport Context stores the Socket.LocalEndPoint value in the *HostInformation.LocalPhysicalAddress* property.

For example, if we want to obtain the end point of the remote host, we can use

```
[C#]
string remoteEndPoint = GenuineUtility.FetchHostInformationFromMbr(
    (MarshalByRefObject) anyRemoteMbrObject).PhysicalAddress.ToString();

[VB.NET]
Dim remoteEndPoint as String
remoteEndPoint = GenuineUtility.FetchHostInformationFromMbr( _
    DirectCast(anyRemoteMbrObject, MarshalByRefObject).PhysicalAddress.ToString())
```

2.5. Parameter Provider

The *IParameterProvider* interface provides full access to the collection containing all parameters used by the Transport Context.

Parameters stored in the *IParameterProvider* collection can have different types such as int (Int32), bool (Boolean), TimeSpan, string (String). Consider that Transport Context accepts **only string values during creation**.

```
[C#]
IDictionary transportProperties = new Hashtable();

// it is necessary to use only strings to create initial hashtable
transportProperties["MaxContentSize"] = "100000";
transportProperties["InvocationTimeout"] = "300000";

// create the transport context
ITransportContext = TransportContextServices.CreateDefaultTcpContext(transportProperties, null);

// but then you have to use normal types
ITransportContext.IParameterProvider[GenuineParameter.MaxContentSize] = 300000;
ITransportContext.IParameterProvider[GenuineParameter.InvocationTimeout] = TimeSpan.FromMinutes(1);

[VB.NET]
Dim transportProperties As IDictionary = New Hashtable

" it is necessary to use only strings to create initial hashtable
transportProperties("MaxContentSize") = "100000"
transportProperties("InvocationTimeout") = "300000"

" create the transport context
ITransportContext = TransportContextServices.CreateDefaultTcpContext(transportProperties, Nothing)

" but then you have to use normal types
```

```
iTransportContext.IparameterProvider(GenuineParameter.MaxContentSize) = 300000
iTransportContext.IparameterProvider(GenuineParameter.InvocationTimeout) = TimeSpan.FromMinutes(1)
```

Boolean values must be written as “true” or “false” (case-insensitive) strings, TimeSpan values are always written in milliseconds. The timeout parameter set to “30000” means 30 seconds.

All parameter names are case-insensitive.

The table below contains only those parameters that do not depend on the Connection Manager being used. Connection Manager parameters are described in the corresponding sections.

Message parameters		
Name	Type, default value, unit	Description
MaxContentSize	int 20 000 000 (bytes)	The maximum size of a single message allowed to be sent or received.
MaxQueuedItems	int 100 (messages)	The maximum number of items in the queue. See the Queuing section below for more details.
MaxTotalSize	int 20 000 000 (bytes)	The maximum total size of all messages in the queue. See the Queuing section below for more details.
NoSizeChecking	bool false	Enables or disables message size checking. If message size checking is off, this considerably increases performance for messages containing streams that do not support the Stream.Length property, because no exceptions are thrown and caught. The MaxContentSize and MaxTotalSize queue constraints are ignored if the value of this parameter is false. See the Queuing section below for more details.
Compression	bool, false	Initializes the default Security Session with compression at the Transport Context level.
InvocationTimeout	TimeSpan, 120 000 (milliseconds)	The invocation timeout. An exception will be dispatched to the caller if the response to the message is not received within this time period specified by this value.
SyncResponses	bool, true	Represents a value indicating whether to force the synchronous transport approach for delivering responses.

Common Transport Parameters		
ConnectTimeout	TimeSpan 120 000 (milliseconds)	An exception is dispatched to the caller if no connection to the remote host is established within this time span.
SecuritySessionForPersistentConnections	String null (name)	The name of the key provider to create a Security Session used at the connection level.
SecuritySessionForNamedConnections	String null (name)	The name of the key provider to create a Security Session used at the connection level.

<i>SecuritySessionForInvocationConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.
<i>SecuritySessionForOneWayConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.
<i>ClosePersistentConnectionAfterInactivity</i>	TimeSpan 100 000 (milliseconds)	The period of inactivity to close opened or accepted persistent connections after.
<i>CloseNamedConnectionAfterInactivity</i>	TimeSpan 120 000 (milliseconds)	The period of inactivity to close opened or accepted named connections after.
<i>CloseInvocationConnectionAfterInactivity</i>	TimeSpan 15 000 (milliseconds)	The period of inactivity to close opened or accepted invocation connections after.
<i>CloseOneWayConnectionAfterInactivity</i>	TimeSpan 15 000 (milliseconds)	The period of inactivity to close opened or accepted one-way connections after.
<i>PersistentConnectionSendPingAfterInactivity</i>	TimeSpan 40 000 (milliseconds)	An empty message (6 byte) is sent to the remote host if there are no messages sent to the remote host within this time span.
<i>MaxTimeSpanToReconnect</i>	TimeSpan 180 000 (milliseconds)	The time span before considering a persistent connection to be broken if it is not reestablished.
<i>ReconnectionTries</i>	int, 180, (tries)	The maximum number of reconnection attempts before declaring a connection to be broken.
<i>SleepBetweenReconnections</i>	TimeSpan, 500, (milliseconds)	The time span to wait for after every reconnection failure. Do not set this value to zero due to the following reason. All events generated by Genuine Channels are distributed in separate threads, so you have a chance to receive the event that the connection is reestablished before you receive the event that the connection is being reestablished. As you see, you have 500 milliseconds to process this event.

2.6. Direct Exchange Manager

Direct eXchange Manager (DXM) provides a safe, simple and efficient way to send invocations directly. The word *directly* means that you can avoid using .NET Remoting, sinks, channels, proxies, serialization and deserialization. Note that the entire Genuine Channels stuff, such as Security Sessions, Client Session, message and queue restrictions, timeout constraints and so on, will continue working for you.

Direct eXchange Manager works with streams. In other words, with DXM you can send and receive streams handled by DXM service providers.
You can use both .NET Remoting and Direct Exchange Manager simultaneously in your application without any restrictions.

2.6.1. Sending

It is very easy to work with Direct Exchange Manager. If you want to send something to a remote host, you need only three things: a Transport Context (ITransportContext), a remote host (HostInformation) and content (System.IO.Stream).

You can create the Transport Context or obtain it from objects supporting the ITransportContextProvider interface (any Genuine Channel or HostInformation). In addition, the Transport Context is available during events when it is possible.

If you have only the URL of the remote host, it is a very bad idea to try to get the corresponding HostInformation object from KnownObjects directly like this:

```
[C#]
// never do so!
string url = "gtcp://myRemoteServer:8737/";
HostInformation hostInformation = ITransportContext.KnownHosts.Get(url);

// and these two lines are certainly a bad idea
string url = "ghhttp://myRemoteServer/BackEnd/Service.rem";
HostInformation hostInformation = ITransportContext.KnownHosts.Get(url);

[VB.NET]
" never do so!
Dim url As String = "gtcp://myRemoteServer:8737/"
Dim hostInformation As HostInformation = ITransportContext.KnownHosts.Get(url)

" and these two lines are certainly a bad idea
Dim url As String = "ghhttp://myRemoteServer/BackEnd/Service.rem"
Dim hostInformation As HostInformation = ITransportContext.KnownHosts.Get(url)
```

Instead you should let Genuine Channels get the URL of the remote host:

```
[C#]
// this approach is OK
string ignored;
string url = GenuineUtility.Parse(url, out ignored);
HostInformation hostInformation = ITransportContext.KnownHosts.Get(url);

[VB.NET]
" this approach is OK
Dim ignored As String
Dim url As String = GenuineUtility.Parse(url, ignored)
Dim hostInformation As HostInformation = ITransportContext.KnownHosts.Get(url)
```

If you have an instance of the **HostInformation** class, you can get the Transport Context from it.

Having got both the Transport Context and the HostInformation object, you can send a stream to the remote host. The stream object must be inherited from the **System.IO.Stream** class. If you are not going to send a file (FileStream) or a simple chunk of memory (MemoryStream), then it is a very good idea to use an instance of the **Belikov.GenuineChannels.Messaging.GenuineChunkedStream** class here.

```
[C#]
// You should specify false parameter during construction,
// otherwise Connection Manager will not be able to re-send the content
// after reconnection or a network failure.
GenuineChunkedStream sourceContent = new GenuineChunkedStream(false);
```



```
// serialization with BinaryWriter
// very simple and the fastest approach
BinaryWriter binaryWriter = new BinaryWriter(sourceContent);
binaryWriter.Write(number);
binaryWriter.Write(comment);

// Small hint: if you want to write a size label (Int32), you can use
// the GenuineChunkedStream.WriteInt32AndRememberItsLocation method.
// Either use it directly or via the GenuineChunkedStreamSizeLabel wrapper.
using (new GenuineChunkedStreamSizeLabel(sourceContent))
{
    SerializeDataWithUnknownSizeHere(anArray, sourceContent);
}

// Genuine Chunked stream does not copy the content of the streams being
// added. This is a very efficient solution if you need to send several
// streams in one message
sourceContent.WriteStream(arbitraryStream);

[VB.NET]
" You should specify false parameter during construction,
" otherwise Connection Manager will not be able to re-send the content
" after reconnection or a network failure.
Dim sourceContent As GenuineChunkedStream = new GenuineChunkedStream(false)

" serialization with BinaryWriter
" very simple and the fastest approach
Dim binaryWriter As BinaryWriter = new BinaryWriter(sourceContent)
binaryWriter.Write(number)
binaryWriter.Write(comment)

" Small hint: if you want to write a size label (Int32), you can use
" the GenuineChunkedStream.WriteInt32AndRememberItsLocation method.
" Either use it directly or via the GenuineChunkedStreamSizeLabel wrapper.
Dim genuineChunkedStreamSizeLabel As GenuineChunkedStreamSizeLabel
Try
    genuineChunkedStreamSizeLabel = New GenuineChunkedStreamSizeLabel(sourceContent)
    SerializeDataWithUnknownSizeHere(anArray, sourceContent)
Finally
    genuineChunkedStreamSizeLabel.Dispose()
End Try

" Genuine Chunked stream does not copy the content of the streams being
" added. This is a very efficient solution if you need to send several
" streams in one message
sourceContent.WriteStream(arbitraryStream)
```

As soon as you have the Transport Context, the HostInformation, and the stream, you can send the stream to the remote host. Here you have a choice among synchronous, asynchronous and one-way types of messages.

In case of a synchronous invocation, you send a stream to a specific remote service and receive a stream in reply:

```
[C#]
Stream aResult =
    iTransportContext.DirectExchangeManager.SendSync(hostInformation, serviceName, sourceContent);

[VB.NET]
Dim aResult As Stream = _
    iTransportContext.DirectExchangeManager.SendSync(hostInformation, serviceName, sourceContent)
```

In case of an asynchronous invocation, you send a stream to a specific service of the remote host and handle the response either in the callback specified by the provided delegate (*Belikov.GenuineChannels.DirectExchange.StreamResponseEventHandler*), or in the instance of the class implementing the *Belikov.GenuineChannels.DirectExchange.IStreamResponseHandler* interface.

```
[C#]
// initiating the call
```

```
iTransportContext.DirectExchangeManager.SendAsync(hostInformation,
serviceName, sourceContent, new StreamResponseEventHandler(this.OnFinish),
invocationInformation);

// or
iTransportContext.DirectExchangeManager.SendAsync(hostInformation,
serviceName, sourceContent, IStreamResponseHandler, invocationInformation);
```

[VB.NET]

```
" initiating the call
iTransportContext.DirectExchangeManager.SendAsync(hostInformation,_
serviceName, sourceContent, New StreamResponseEventHandler(Me.OnFinish),_
invocationInformation)

" or
iTransportContext.DirectExchangeManager.SendAsync(hostInformation,_
serviceName, sourceContent, IStreamResponseHandler, invocationInformation)
```

Please note that you can provide an arbitrary object during an asynchronous invocation. Direct eXchange Manager will pass this object back to the response handler.

Handlers being called via delegates receive references to either an instance of the class inherited from the System.IO.Stream class or the instance of the Exception class. It must analyze the type of the received object at run time and make the proper decision on it:

[C#]

```
// the response handler called via delegate
public void OnFinish(object response, HostInformation remoteHost, object invocationInformation)
{
    try
    {
        // Here you have access to Client Session via remoteHost parameter
        // You have the invocationInformation specified during
        // initiating of the asynchronous invocation.

        Exception exception = response as Exception;
        if (exception != null)
            throw exception;

        Stream stream = response as Stream;
        ParseResponse(stream);
    }
    catch(Exception ex)
    {
        Console.WriteLine("Exception: {0}; Stack: {1}.", ex.Message, ex.StackTrace);
    }
}
```

[VB.NET]

```
" the response handler called via delegate
Public Sub OnFinish(ByVal response As Object, ByVal remoteHost As _
HostInformation, ByVal invocationInformation As Object)
Try
    " Here you have access to Client Session via remoteHost parameter
    " You have the invocationInformation specified during
    " initiating of the asynchronous invocation.
    Dim exception As Exception = DirectCast(response, Exception)
    If Not exception Is Nothing Then
        Throw exception
    End If

    Dim stream As Stream = DirectCast(response, Stream)
    ParseResponse(stream)
Catch ex As Exception
    Console.WriteLine("Exception: {0}; Stack: {1}.", ex.Message, ex.StackTrace)
End Try
End Sub
```

If you prefer to provide an instance of the class instead of the delegate, you need to implement

the `IStreamResponseHandler` interface:

```
[C#]
public class ResponseHandler : IStreamResponseHandler
{
    // Handles the stream coming from the remote host in reply to
    // the sent request.
    public void HandleResponse(Stream response, HostInformation remoteHost, object tag)
    {
        // Here you have access to Client Session via remoteHost parameter
        // You have the invocationInformation specified during
        // initiating of the asynchronous invocation.

        try
        {
            CheckResponse(response);
        }
        catch(Exception ex)
        {
            Console.WriteLine("Exception: {0}; Stack: {1}.", ex.Message, ex.StackTrace);
        }
    }

    // Dispatches the exception to the response processor.
    public void HandleException(Exception exception, HostInformation remoteHost, object invocationInformation)
    {
        // Here you have access to Client Session via remoteHost parameter
        // You have the invocationInformation specified during
        // initiating of the asynchronous invocation.

        Console.WriteLine("Exception: {0}; Stack: {1}.", exception.Message, exception.StackTrace);
    }
}

[VB.NET]
Public Class ResponseHandler
    Implements IStreamResponseHandler

    " Handles the stream coming from the remote host in reply to
    " the sent request.
    Public Sub HandleResponse(ByVal response As Stream, ByVal remoteHost As HostInformation, ByVal tag As Object)
        " Here you have access to Client Session via remoteHost parameter
        " You have the invocationInformation specified during
        " initiating of the asynchronous invocation.

        Try
            CheckResponse(response)
        Catch ex As Exception
            Console.WriteLine("Exception: {0}; Stack: {1}.", ex.Message, ex.StackTrace)
        End Try
    End Sub

    " Dispatches the exception to the response processor.
    Public Sub HandleException(ByVal exception As Exception, ByVal remoteHost As HostInformation, _
        ByVal invocationInformation As Object)
        " Here you have access to Client Session via remoteHost parameter
        " You have the invocationInformation specified during
        " initiating of the asynchronous invocation.
        Console.WriteLine("Exception: {0}; Stack: {1}.", exception.Message, exception.StackTrace)
    End Sub
End Class
```

To send a one-way invocation, use the `DirectExchangeManager.SendOneWay` member.

```
[C#]
iTransportContext.DirectExchangeManager.SendOneWay(hostInformation, serviceName, sourceContent);

[VB.NET]
iTransportContext.DirectExchangeManager.SendOneWay(hostInformation, serviceName, sourceContent)
```

You will not receive a response and you will not get an exception if something is wrong.

2.6.2. Receiving

We will use the term *DXM Service* for all objects supporting the *Belikov.GenuineChannels.DirectExchange.IServerServiceEntry* interface. In order to catch incoming streams sent via Direct Exchange Manager, you need to associate a DXM Service with a string treated as the name of the service. If you want them DXM Service to be available only within a particular Transport Context, you need to register it at *ITransportContext.DirectExchangeManager*. It is possible to register a global DXM Service that will be available through any Transport Context. Use the static *Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.RegisterGlobalServerService* method to do it. Services registered at a specific Transport Context override global services associated with the same service name.

[C#]

```
// Register the DXM Service that will be available through any Transport Context.
Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.RegisterGlobalServerService (
    "/Test/Service1", new DirectExchangeService());
```

```
// Register the DXM Service that will be available only through the specified Transport Context.
// This service overrides a DXM Service registered above.
iTransportContext.DirectExchangeManager.RegisterServerService (
    "/Test/Service1", new DirectExchangeService());
```

[VB.NET]

```
' Register the DXM Service that will be available through any Transport Context.
Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.RegisterGlobalServerService (
    "/Test/Service1", New DirectExchangeService())

' Register the DXM Service that will be available only through the specified Transport Context.
' This service overrides a DXM Service registered above.
iTransportContext.DirectExchangeManager.RegisterServerService ("/Test/Service1", New DirectExchangeService())
```

To unregister a service, call *DirectExchangeManager.UnregisterServerService* or *Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.UnregisterGlobalServerService* respectively:

[C#]

```
Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.UnregisterGlobalServerService ("/Test/Service1");
```

// or

```
iTransportContext.DirectExchangeManager.UnregisterServerService ("/Test/Service1");
```

[VB.NET]

```
Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.UnregisterGlobalServerService ("/Test/Service1")
```

' or

```
iTransportContext.DirectExchangeManager.UnregisterServerService ("/Test/Service1")
```

To get the list of all registered services, call the *DirectExchangeManager.GetListOfGlobalRegisteredServices* or *ITransportContext.DirectExchangeManager.GetListOfRegisteredServices* method:

[C#]

```
string[] globalServices = Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.GetListOfGlobalRegisteredServices();
```

// or

```
string[] localServices = iTransportContext.DirectExchangeManager.GetListOfRegisteredServices();
```

[VB.NET]

```
Dim globalServices() As String =
```

```

    Belikov.GenuineChannels.DirectExchange.DirectExchangeManager.GetListOfGlobalRegisteredServices()
' or
Dim localServices() As String = iTransportContext.DirectExchangeManager.GetListOfRegisteredServices()

```

To implement a stream consumer, you should consider the following. First, you must close or read the entire stream (to the very end) as soon as possible. Sometimes the reading will be performed directly from the transport provider (TCP connection, HTTP response, memory share and so on), so if you do not close or read the entire message quickly, you will hold the underlying connection and all processes dependent on it. In addition, BufferPool's buffers used for caching will be released immediately. This will result in increased memory consumption and worse performance.

Your handler should not throw any exceptions, so, please, deal with them inside. If you do not catch an exception, the request processor will try to send it to the remote host in order to dispatch it to the caller. As a result, you should be sure that the exception is serializable.

Generally, the provided stream does not support seeking and the Stream.Length property. In some cases, Stream.Read will return only the currently available data without filling up the provided buffer completely. In other words, if you provide a buffer and request 2500 bytes, it can return 147 bytes during the first Stream.Read invocation, 1046 bytes during the second invocation and so on.

In all other respects, the implementation is usually straightforward: you get a stream and return the stream (or the Stream.Null value):

```

[C#]
public class DirectExchangeService : IServerServiceEntry
{
    // Processes incoming requests. Must close the provided stream
    // immediately after using. May not throw any exceptions.
    public Stream HandleMessage(Stream stream, HostInformation sender)
    {
        try
        {
            try
            {
                // TODO: parse the stream here

                BinaryReader binaryReader = new BinaryReader(stream);
                int number = binaryReader.ReadInt32();
                string str = binaryReader.ReadString();
                Console.WriteLine("{0} - {1}.", number, str);
            }
            finally
            {
                stream.Close();
            }

            // TODO: prepare the response

            GenuineChunkedStream outputStream =
                new GenuineChunkedStream(false);
            BinaryWriter binaryWriter = new BinaryWriter(outputStream);
            binaryWriter.Write((int) 3737);
            binaryWriter.Write("The response");
            return outputStream;
        }
        catch(Exception ex)
        {
            // TODO: write it to the log here
        }

        return Stream.Null;
    }
}

```

```

[VB.NET]
Public Class DirectExchangeService
    Implements IServerServiceEntry
    " Processes incoming requests. Must close the provided stream
    " immediately after using. May not throw any exceptions.
    Public Function HandleMessage(ByVal stream As Stream, _
        ByVal sender As HostInformation) As Stream
        Try
            Try
                " TODO: parse the stream here

                Dim binaryReader As BinaryReader = New BinaryReader(stream)
                Dim number As Integer = binaryReader.ReadInt32()
                Dim str As String = binaryReader.ReadString()
                Console.WriteLine("{0} - {1}.", number, Str)
            Finally
                stream.Close()
            End Try

            " TODO: prepare the response
            Dim outputStream As GenuineChunkedStream = _
                New GenuineChunkedStream(False)
            Dim binaryWriter As BinaryWriter = New BinaryWriter(outputStream)
            binaryWriter.Write(Convert.ToInt16(3737))
            binaryWriter.Write("The response")
            Return outputStream
        Catch ex As Exception
            " TODO: write it to the log here
        End Try

        Return Stream.Null
    End Function
End Class

```

If you return a null reference instead of the *Stream.Null* value, the *Stream.Null* value will be sent as a result.

It is necessary to emphasize here that all Genuine Channels features and restrictions are always applied. For example, if you use an SSPI Security Session, the stream consumer will be invoked in the impersonated security context.

2.7. Synchronous and asynchronous transport approaches

First, do not mix up synchronous and asynchronous transport approaches up with synchronous and asynchronous invocations. In spite of the fact that Connection Manager always tries to use the synchronous transport approach for sending synchronous invocations and the asynchronous approach for asynchronous invocations, generally they are independent things.

The synchronous transport approach uses a thread to send an invocation to the remote host. This leads to a faster exchange but the thread is held while the message is being delivered.

The asynchronous transport approach uses completion ports for sending and receiving messages. This slows down the exchange between two specific hosts, but the server can serve more clients and does not have to have a thread per client.

Short summary

Condition	The Synchronous approach	The Asynchronous approach
Performance	Noticeably faster than the asynchronous approach.	Slower than the synchronous approach.

Scalability	Consumes a thread while sending an invocation. The thread is released automatically when sending is completed.	Uses completion ports and the asynchronous I/O API to initiate sending.
Underlying API	Uses ordinary means to implement synchronous sending.	Based on completion ports.

1. Synchronous sending does not take a thread per client. It uses the provided thread (where the invocation is initiated) only to send the current invocation to the remote host.
2. Asynchronous sending does not use any threads at all. In this mode, Genuine Channels initiate sending a message with the help of the asynchronous I/O API.
3. All timeout and message checks are always performed correctly, no matter what approach you use.
4. You can specify a different approach for every invocation you initiate (for every message being sent).

Genuine Channels always use the synchronous approach for sending synchronous messages. It is faster and the number of threads does not matter here because you will wait for the response anyway in order to continue processing.

Genuine Channels always use the asynchronous approach for sending asynchronous messages. The Broadcast Engine always sends invocations asynchronously.

Methods of sending

Connection Manager	Synchronous invocations	Asynchronous invocations
GTCP	Are sent synchronously.	Are sent asynchronously.
GXHTTP	Are sent synchronously.	Are sent asynchronously.
GHTTP	Are always sent asynchronously because the underlying API does not implement the synchronous I/O.	Are sent asynchronously.
GUDP	Are sent synchronously.	Are always sent synchronously because sending depends neither on the state of the network nor on the availability of the remote host.
Shared Memory	Are sent synchronously.	Are always sent synchronously because the asynchronous approach is inadmissibly slow.

You can enforce a specific method of sending with the help of Security Session Parameters. The following snippet enforces the synchronous method of sending for absolutely all invocations made via Genuine Channels.

```
[C#]
SecuritySessionServices.SetCurrentSecurityContext ( new SecuritySessionParameters (
    SecuritySessionServices.DefaultContext, SecuritySessionAttributes.ForceSync, TimeSpan.MinValue,
    GenuineConnectionType.Persistent, null, TimeSpan.FromMinutes(5) ) );
```

```
// invocations ...
```

```
[VB.NET]
```

```
SecuritySessionServices.SetCurrentSecurityContext ( New SecuritySessionParameters ( _  
    SecuritySessionServices.DefaultContext, SecuritySessionAttributes.ForceSync, TimeSpan.MinValue, _  
    GenuineConnectionType.Persistent, Nothing, TimeSpan.FromMinutes(5) ) )
```

The method of receiving

Connection Manager	The method of receiving
GTCP	Uses the combined synchronous/asynchronous approach. Does not consume a thread per client or a thread per Channel (Transport Context).
GXHTTP	Uses the combined synchronous/asynchronous approach. Does not consume a thread per client or a thread per Channel (Transport Context).
GHTTP	Both GHTTP client and GHTTP server Connection Managers uses the pure asynchronous approaches. Does not consume a thread per client or thread per Channel (Transport Context).
GUDP	Uses the pure synchronous approach. Consumes a thread per Channel (or Transport Context).
Shared Memory	Uses the pure synchronous approach. Consumes a thread per client (Remote Host).

2.8. Three Connection Patterns

This part of Genuine Channels is still experimental and only the Persistent pattern is supported by all channels. However, there are three patterns that will be supported: **Persistent**, **Named**, and **Invocation**. The table below contains comparative characteristics of every connection pattern.

Feature	Persistent	Named	Invocation
Connection establishing	There is only one connection between two peers opened by the client and accepted by the server.	Each connection has a unique connection name. The client is responsible for specifying the connection name during every call.	A separate connection is opened for each invocation. The connection can be re-used for other invocations after the previous invocation is completed.
Bi-directional (the server can invoke client's objects and fire events through this connection)	Yes	Yes. The server must specify the name of the connection.	No

Serialization	Every connection has a queue of messages. Connection Manager performs serialization into the intermediate container.	Every connection has a queue of messages. Connection Manager performs serialization into the intermediate container.	No queue is used. Serialization and deserialization performed directly to/from the transport stream.
Network failure	Results in reestablishing the connection. All invocations made during the reestablishing are stored in the queue.	Results in releasing all resources allocated for this connection and dispatching the exception to all callers dependent on this connection.	Results in releasing all resources allocated for this connection and dispatching the exception to the caller.
NLB	It's a bad idea to use this pattern with NLB. There will be no load balancing and every reconnection may result in the "Server has been restarted" Exception.	Perfectly fits the transaction scenario. The client can initiate the operation and the server can inform the client about the operation progress and/or request some additional information.	Each invocation will be directed to the available server according to the general idea of NLB.
Ping	An empty message (6 bytes) is sent after the specified time of inactivity. The connection is considered to be broken if there are no successfully received messages within the specified time span.	An empty message (6 bytes) is sent after the specified time of inactivity. The connection is considered to be broken if there are no successfully received messages within the specified time span.	No pings are sent. The remote host must respond to the sent request within the specified time span. After the invocation is completed, the established connection may remain alive for the specified time span.
Support in release 2.5.0	GTCP, GHTTP, Shared memory	GTCP	GHTTP, GUDP

2.8.1. Persistent connection

The persistent connection scheme is the most important network scheme if you do not use NLB. It provides the following benefits in comparison with other patterns:

1. Only one connection is opened between the client and the server (with the exception of GHTTP channel, which opens two keep-alive HTTP connections in order to provide a bidirectional link). Usually, the opening of the connection consumes resources and takes noticeable time.
2. A persistent connection enqueues invocations if it is currently unavailable. This may happen if you send asynchronous requests or if another request is sent in a separate thread.
3. Message queue constraints are applied to every message queue. See the Message Queue

section below for further explanation.

4. Any connection failure causes connection reestablishing. All messages sent during this period are enqueued. If the connection is successfully reestablished, all accumulated messages are sent to the remote host. No messages get lost. If the server is restarted, an exception is dispatched to all callers that initiated an invocation before or while reestablishing.
5. It is a good idea to use a Connection-Level Security Session with persistent connections.

Disadvantages

1. It is a bad idea to open persistent connections to the NLB cluster (even if the client affinity feature is enabled).
2. One solid request (or response) can cause a standstill of other concurrent invocations, because they will have to wait until the request or the response will be sent (or received). So it is recommended to send large chunks of data through separate (for example, via invocation) connections.

2.8.2. Named connection

Named connections are useful if you need to implement the transaction-like behavior. Every connection can represent an operation performed on a specific server node. The server can invoke client's objects, request additional information and send indications of the operation progress.

Advantages

1. Every named connection possesses with a queue and can be used in different threads concurrently.
2. You are supposed to use named connections with NLB.
3. It is a good idea to use a Connection-Level Security Session with named connections.

Disadvantages

1. The connection reestablishing is not supported because you will not be able to connect to the same server node behind NLB (even if the client affinity feature is enabled). Therefore, a connection failure results in breaking the transaction.

2.8.3. Invocation connection

Invocation connections are extremely useful for sending large data and/or utilizing NLB services. Every invocation is supposed to be sent through a separate connection.

Advantages

1. Serialization (deserialization) is performed directly to (from) the transport stream. This results in less memory consumption and better performance.
2. TCP and HTTP connections have a tendency to slow down the exchange through lossy, high bandwidth and high latency links. Opening a new connection for each invocation reduces the consequences of this problem.

Disadvantages

1. A new connection will be probably opened for each invocation unless you change the default connection settings.
2. If a connection fails, you will not be able to receive the response to the invocation.

3. Security

Genuine Channels Security deals with encryption, content integrity, authentication and authorization, security context impersonation and delegation of authority. In order to leverage security functionality, you need to create and associate a Security Key Provider with an arbitrary name. Security Key Providers define security algorithms and settings.

For example, by creating a SES Security Key Provider, you state that you might want to use Self-establishing security algorithms (RSA and Rijndael) somewhere in your application.

```
[C#]
ITransportContextProvider iTransportContextProvider = (ITransportContextProvider) ChannelServices.GetChannel("gtcp");
iTransportContextProvider.ITransportContext.IKeyStore.SetKey("/Chat/SES", new KeyProvider_SelfEstablishingSymmetric());
```

```
[VB.NET]
Dim iTransportContextProvider As ITransportContextProvider = _
    DirectCast (ChannelServices.GetChannel("gtcp"), ITransportContextProvider)
iTransportContextProvider.ITransportContext.IKeyStore.SetKey("/Chat/SES", New KeyProvider_SelfEstablishingSymmetric())
```

Then, you need to create Security Session Parameters that play the role of an invocation context declaring what Security Key Provider with what additional options is used. Additional options include the invocation timeout, the usage of compression, the requested connection pattern, the requested sending behavior and some other things.

In our example, by creating Security Session Parameters, you say that you want to use that created SES Security Key Provider without compression and with the default invocation timeout.

```
[C#]
SecuritySessionParameters securitySessionParameters = new SecuritySessionParameters("/Chat/SES");
iTransportContextProvider.ITransportContext.SecuritySessionParameters = new SecuritySessionParameters("/Chat/SES");
```

```
[VB.NET]
Dim securitySessionParameters As SecuritySessionParameters = New SecuritySessionParameters("/Chat/SES")
iTransportContextProvider.ITransportContext.SecuritySessionParameters = New SecuritySessionParameters("/Chat/SES")
```

Once you create and register a Security Key Provider and have proper Security Session Parameters, you can enforce the usage of Security Session Parameters in one of five execution contexts. Execution context determines for what invocations the specified Security Session Parameters are used.

In our example, we enforce the usage of the Security Session Parameters for all invocations made via a specific Transport Context. We also could enforce it for all invocation made within the local scope in the current thread, for all invocations to a specific remote host, for all invocations made in a particular thread and so on.

```
[C#]
iTransportContextProvider.ITransportContext.SecuritySessionParameters = new SecuritySessionParameters("/Chat/SES");
```

```
[VB.NET]
iTransportContextProvider.ITransportContext.SecuritySessionParameters = New SecuritySessionParameters("/Chat/SES")
```

Security Key Providers define only the general concept of the security behavior. In order to manage this behavior between two specific hosts, Security Sessions are created and established. For example, if you are going to use a Self-Established Security Key Provider between the client and the server, you create and register it once on both client and server sides. Each time a new client is connected, a new Security Session is created, which will be responsible for securing traffic for that client. Therefore, the server will have as many Security Sessions as many clients are connected.

3.1. Key Providers

Key Provider is a factory spawning Security Sessions. You must initialize and register a Key Provider before the corresponding security services are requested.

List of available Key Providers

Key Provider	Description
Basic	<p>Full type: <i>Belikov.GenuineChannels.Security.KeyProvider_Basic.</i></p> <p>This Key Provider is used by default. Does not provide any security features.</p>
Known Symmetric	<p>Full type: <i>Belikov.GenuineChannels.Security.KeyProvider_KnownSymmetric.</i></p> <p>This Key Provider requires a symmetric algorithm to be specified during initialization. This algorithm will be used for encryption and decryption in all spawned Security Sessions.</p>
Self Establishing Symmetric	<p>Full type: <i>Belikov.GenuineChannels.Security.SecuritySession_SelfEstablishingSymmetric</i></p> <p>Security Sessions created by this Key Provider automatically generate 256-bit key for using with the Rijndael Symmetric Algorithm. The key is transmitted with the help of asymmetric encryption (1024-bit RSA is used).</p>
SSPI	<p>Full type: <i>Belikov.GenuineChannels.Security.SSPI.KeyProvider_SspiClient</i> and <i>Belikov.GenuineChannels.Security.SSPI.KeyProvider_SspiServer.</i></p> <p>Security Sessions created by this Key Provider provide authorization, impersonation, delegation, encryption and integrity checking features based on SSPI NTLM and Kerberos protocols.</p>
Zero Proof Authorization	<p>Full type: <i>Belikov.GenuineChannels.Security.ZeroProofAuthorization.KeyProvider_ZpaClient</i> and <i>Belikov.GenuineChannels.Security.ZeroProofAuthorization.KeyProvider_ZpaServer</i></p> <p>Security Sessions created by this Key Provider provide authorization, encryption and integrity checking features based on the Zero Proof Authorization algorithm.</p>

Comparative table

Functionality	Symmetric	Self-Establishing Symmetric	SSPI	Zero Proof Authorization

Encryption	A symmetric algorithm specified while creating the Key Provider.	256-bit Rijndael in the ECB chaining mode.	Supported. See MSDN for details.	256-bit Rijndael in the CBC chaining mode.
Authorization	Not supported.	Not supported.	Based on the Windows security system.	Based on a custom password provider.
Windows security context Impersonation and Delegation	Not supported.	Not supported.	Supported. Technical characteristics depend on the chosen package. See MSDN for details.	Not supported. Though you can always get a remote user's credentials to check the user's permissions.
Integrity Checking	Not supported.	Not supported.	Supported. Technical characteristics depend on the chosen package. See MSDN for details.	A choice between two algorithms based on MAC-3DES-CBC (64 bit) and HMAC-SHA1 (160 bit).
Channel support	All channels	All channels except GUDP.	All channels except GUDP.	All channels except GUDP.
Connection Level Security Sessions	Supported.	Supported.	Supported.	Supported.

Security Key Providers can be registered either locally in a particular Transport Context or globally at *Belikov.GenuineChannels.Security.SecuritySessionServices*. The Transport Context stores the set of Key Providers in the *ITransportContext.IKeyStore* member.

To add or remove a local Key Provider, use the *ITransportContext.IKeyStore.SetKey* object:

```
[C#]
// to register a Self-Establishing Symmetric Key Provider
ITransportContext.IKeyStore.SetKey(keyName, new KeyProvider_SelfEstablishingSymmetric());

// to remove it, provide a null reference instead of the instance
// of the Key Provider
ITransportContext.IKeyStore.SetKey(keyName, null);

[VB.NET]
" to register a Self-Establishing Symmetric Key Provider
ITransportContext.IKeyStore.SetKey(keyName, New KeyProvider_SelfEstablishingSymmetric())

' to remove it, provide Nothing instead of the instance of the Key Provider
ITransportContext.IKeyStore.SetKey(keyName, Nothing)
```

To add or remove a global Key Provider, which will work within all Transport Contexts, use the *Belikov.GenuineChannels.Security.SecuritySessionServices.SetGlobalKey* method:

[C#]

```
// to register a Self-Establishing Symmetric Key Provider
Belikov.GenuineChannels.Security.SecuritySessionServices.SetGlobalKey (
    keyName, new KeyProvider_SelfEstablishingSymmetric());

// to remove it, provide a null reference instead of the instance
// of the Key Provider
Belikov.GenuineChannels.Security.SecuritySessionServices.SetGlobalKey (keyName, null);
```

[VB.NET]

```
' to register a Self-Establishing Symmetric Key Provider
Belikov.GenuineChannels.Security.SecuritySessionServices.SetGlobalKey ( _
    keyName, New KeyProvider_SelfEstablishingSymmetric())

' to remove it, provide Nothing instead of the instance of the Key Provider
Belikov.GenuineChannels.Security.SecuritySessionServices.SetGlobalKey (keyName, Nothing)
```

You can use the *IKeyStore.GetKey* method to get an existing Key Provider or to make sure it exists.

3.2. Security Session

The Security Session is an instance of the class inherited from the *Belikov.GenuineChannels.Security.SecuritySession* class. The Security Session serves only one connection to a specific remote host. If the connection is considered to be broken, the Security Session is destroyed. If the connection is successfully reestablished, the same Security Session is used.

To eliminate a security session, use the *HostInformation.DestroySecuritySession* method.

Security Sessions can work either on the Connection Level or on the Invocation Level. Do not think that an Invocation Level Security Session is established for each invocation! Actually, the difference between Connection Level Security Sessions and Invocation Level Security Sessions is very small. Connection Level Security Sessions produce only encrypted traffic and the receiving host must know what Security Session with what parameters is used. While an Invocation Level Security Session provides its name, therefore the receiving side can take the corresponding Security Session from Transport Context and use it. In any case, a Security Session is established only once per connection lifetime.

Short summary

Feature	Connection Level	Invocation Level
Key Provider creation	A Key Provider must be created and registered before establishing a connection.	A Key Provider can be created and registered at any moment.
Establishing	A Security Session is created and established while the connection is being established.	A Security Session is created and established as soon as you make an invocation requiring the Security Session.
Establishing failure	The connection is immediately closed. No messages may be sent before Connection Level Security Session is successfully established.	Only invocations requiring the specific Security Session are failed. This does not affect other messages that use other Security Sessions.

Other messages	All messages sent through the connection are encrypted by the Connection Level Security Session.	Only invocations made within the appropriate Security Session Parameters are encrypted by the Security Session.
Quantity	Only one Connection Level Security Session can be created per connection.	Any number of Security Session can be created per connection.
Lifetime	A Connection Level Security Session is destroyed when the connection to the remote host considered to be broken or closed.	An Invocation Level Security Session is destroyed only when the remote host is considered to be disconnected. You can always destroy an Invocation Level Security Session manually. See the HostInformation section above.

3.2.1. Connection Level Security Session

A Connection Level Security Session serves a specific connection. All packets sent through such a connection are encrypted by the Connection Level Security Session serving this connection. Generally, this is more securable form of encryption than the use of Invocation Level Security Sessions or any custom means.

To use a Connection Level Security Session, you need to

1. Create and register appropriate Key Providers at both hosts under the same name.

For example, if you use Zero Proof Authorization, you need to create the *KeyProvider_ZpaClient* Key Provider and associate it with the name you choose:

[C#]

CLIENT SIDE

```
using Belikov.GenuineChannels.DotNetRemotingLayer;
using Belikov.GenuineChannels.Security;
using Belikov.GenuineChannels.Security.ZeroProofAuthorization;
using Belikov.GenuineChannels.TransportContext;

// create and register the Key Provider
BasicChannelWithSecurity basicChannelWithSecurity = (BasicChannelWithSecurity) channel;
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/ZPA1",
    new KeyProvider_ZpaClient(flags, login, password));
```

[VB.NET]

CLIENT SIDE

```
Imports Belikov.GenuineChannels.DotNetRemotingLayer
Imports Belikov.GenuineChannels.Security
Imports Belikov.GenuineChannels.Security.ZeroProofAuthorization
Imports Belikov.GenuineChannels.TransportContext

' create and register the Key Provider
Dim basicChannelWithSecurity As BasicChannelWithSecurity = DirectCast(channel, BasicChannelWithSecurity)
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/ZPA1", _
    New KeyProvider_ZpaClient(flags, login, password))
```

On the server side, you need to create an instance of the *KeyProvider_ZpaServer* class and associate it with **the same name**:

[C#]

SERVER SIDE

```
using Belikov.GenuineChannels.DotNetRemotingLayer;
using Belikov.GenuineChannels.Security;
using Belikov.GenuineChannels.Security.ZeroProofAuthorization;
using Belikov.GenuineChannels.TransportContext;

// create and register the Key Provider
BasicChannelWithSecurity basicChannelWithSecurity = (BasicChannelWithSecurity) channel;
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/ZPA1",
    new KeyProvider_ZpaServer(flags, passwordProvider));
```

[VB.NET]

SERVER SIDE

```
Imports Belikov.GenuineChannels.DotNetRemotingLayer
Imports Belikov.GenuineChannels.Security
Imports Belikov.GenuineChannels.Security.ZeroProofAuthorization
Imports Belikov.GenuineChannels.TransportContext

" create and register the Key Provider
Dim basicChannelWithSecurity As BasicChannelWithSecurity = DirectCast (channel, BasicChannelWithSecurity)
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/ZPA1", _
    New KeyProvider_ZpaServer(flags, passwordProvider))
```

- Specify the *GenuineParameter.SecuritySessionForXXXConnections* parameter during Transport Context initialization, or before connection establishing:

[C#]

```
// specifies the name of the Key Provider which will be used to create
// Connection Level Security Sessions for all connections in this
// Transport Context.
ITransportContext.IParameterProvider [GenuineParameter.SecuritySessionForPersistentConnections] = keyName;
```

[VB.NET]

```
" specifies the name of the Key Provider which will be used to create
" Connection Level Security Sessions for all connections in this
" Transport Context.
ITransportContext.IParameterProvider (GenuineParameter.SecuritySessionForPersistentConnections) = keyName
```

A Connection Level Security Session is established immediately after the connection is accepted by a server application. No client or server information is sent through the connection until the Connection Level Security Session is successfully established. For example, if you use a Connection Level SSPI or Zero Proof Authorization Security Session, no messages may be received from (or sent to) the client until it proves its authority. The client application has exactly *GenuineParameter.ConnectTimeout* milliseconds to complete establishing the Connection Level Security Session. If the Connection Level Security Session is not established within the *GenuineParameter.ConnectTimeout* time span, the connection is broken and all resources associated with this connection are released.

To get the Connection Level Security Session while processing an invocation, use *GenuineUtility.CurrentConnectionSecuritySession*.

The Security Context provided by the Connection Level Security Session has the lowest priority. The Security Context established by an Invocation Level Security Session or by HTTP authentication always overrides Security Context defined by the Connection Level Security Session.

3.2.2. Invocation Level Security Session

The only way to specify an Invocation Level Security Session is to enforce the corresponding Security Session Parameters in a specific context. See the Security Session Parameters section

below for more details.

To use an Invocation Level Security Session, you need to

1. Create and register appropriate Key Providers at both hosts.

For example, if you use a Self-Establishing Key Provider, you need to create the `KeyProvider_SelfEstablishingSymmetric` Key Provider and associate it with the chosen name:

<p>[C#]</p> <pre>using Belikov.GenuineChannels.DotNetRemotingLayer; using Belikov.GenuineChannels.Security; using Belikov.GenuineChannels.TransportContext; // create and register the Key Provider BasicChannelWithSecurity basicChannelWithSecurity = (BasicChannelWithSecurity) channel; basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/SES1", new KeyProvider_SelfEstablishingSymmetric());</pre> <p>[VB.NET]</p> <pre>Imports Belikov.GenuineChannels.DotNetRemotingLayer Imports Belikov.GenuineChannels.Security Imports Belikov.GenuineChannels.TransportContext " create and register the Key Provider Dim basicChannelWithSecurity As BasicChannelWithSecurity = DirectCast(channel, BasicChannelWithSecurity) basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/SES1", _ New KeyProvider_SelfEstablishingSymmetric())</pre>	<p>CLIENT SIDE</p> <p>CLIENT SIDE</p>
---	---------------------------------------

On the server side, you need also to create an instance of the `KeyProvider_SelfEstablishingSymmetric` class and associate it with **the same name**:

<p>[C#]</p> <pre>using Belikov.GenuineChannels.DotNetRemotingLayer; using Belikov.GenuineChannels.Security; using Belikov.GenuineChannels.TransportContext; // create and register the Key Provider BasicChannelWithSecurity basicChannelWithSecurity = (BasicChannelWithSecurity) channel; basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/SES1", new KeyProvider_SelfEstablishingSymmetric());</pre> <p>[VB.NET]</p> <pre>Imports Belikov.GenuineChannels.DotNetRemotingLayer Imports Belikov.GenuineChannels.Security Imports Belikov.GenuineChannels.TransportContext ' create and register the Key Provider Dim basicChannelWithSecurity As BasicChannelWithSecurity = DirectCast(channel, BasicChannelWithSecurity) basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/TEST/SES1", _ New KeyProvider_SelfEstablishingSymmetric())</pre>	<p>SERVER SIDE</p> <p>SERVER SIDE</p>
---	---------------------------------------

2. Enforce the Security Context with the chosen name (“/TEST/SES1” in the sample above) if you want an invocation to be sent in this Security Context.

<p>[C#]</p> <pre>SecuritySessionServices.SetCurrentSecurityContext (new SecuritySessionParameters ("/TEST/SES1", SecuritySessionAttributes.ForceSync, TimeSpan.MinValue, GenuineConnectionType.Persistent, null, TimeSpan.FromMinutes(5))); // invocations ...</pre> <p>[VB.NET]</p>	
--	--

```
SecuritySessionServices.SetCurrentSecurityContext ( New _
SecuritySessionParameters ( _
    "/TEST/SES1", _
    SecuritySessionAttributes.ForceSync, TimeSpan.MinValue, _
    GenuineConnectionType.Persistent, Nothing, _
    TimeSpan.FromMinutes(5) ) )
```

See the Security Session Parameters section for more details.

An Invocation Level Security Session is created only when you send a message requiring this Security Session. To get the name of the current Security Session on the server side, use the *GenuineUtility.CurrentInvocationSecuritySession* property. You can get an instance of the Security Session via the *HostInformation.GetSecuritySession* method. To eliminate the Security Session, call the *HostInformation.DestroySecuritySession* method.

3.3. Security Session Parameters

There are five contexts where Security Session Parameters can be specified.

The context	Description
Local Scope	<p>The highest priority.</p> <p>Determines the Security Session Parameters for all invocations made in the local scope (in the same thread within brackets):</p> <pre>[C#] using (new SecurityContextKeeper(securitySessionParameters)) { iRemoteMbr.Dolt(aParameter); iTransportContext.DirectExchangeManager.SendSync (hostInformation, serviceName, content); } [VB.NET] Dim securityContextKeeper As SecurityContextKeeper Try securityContextKeeper = New SecurityContextKeeper(SecuritySessionParameters) iRemoteMbr.Dolt(aParameter) iTransportContext.DirectExchangeManager.SendSync(hostInformation, serviceName, content) Finally securityContextKeeper.Dispose() End Try</pre>

Current Thread	<p>Determines the Security Session Parameters being used in the current thread. If you specify the Current Thread Security Session Parameters within a Local Scope, it will override the Local Scope Parameters until the end of the scope, and will be forfeited after the scope.</p> <p>[C#] <pre>SecuritySessionServices.SetCurrentSecurityContext (new SecuritySessionParameters (SecuritySessionServices.DefaultContext, SecuritySessionAttributes.ForceSync, TimeSpan.MinValue, GenuineConnectionType.Persistent, null, TimeSpan.FromMinutes(5))); // invocations ...</pre></p> <p>[VB.NET] <pre>SecuritySessionServices.SetCurrentSecurityContext (New _ SecuritySessionParameters (_ SecuritySessionServices.DefaultContext, _ SecuritySessionAttributes.ForceSync, TimeSpan.MinValue, _ GenuineConnectionType.Persistent, Nothing, _ TimeSpan.FromMinutes(5))) // invocations ...</pre></p> <p>To remove the Current Thread Security Session Parameters, specify a null reference:</p> <p>[C#] <pre>SecuritySessionServices.SetCurrentSecurityContext (null) ;</pre></p> <p>[VB.NET] <pre>SecuritySessionServices.SetCurrentSecurityContext (Nothing)</pre></p>
Remote Host	<p>Determines the Security Session Parameters being used for all invocations sent to a particular remote host. Can be overridden by the Local Scope and Current Thread Security Session Parameters.</p> <p>[C#] <pre>hostInformation.SecuritySessionParameters = securitySessionParameters;</pre></p> <p>[VB.NET] <pre>hostInformation.SecuritySessionParameters = securitySessionParameters</pre></p>
Transport Context Parameters	<p>Determines the Security Session Parameters being used for all invocations sent via a specific Transport Context. Can be overridden by the Local Scope, Current Context and Remote Host Security Session Parameters.</p> <p>[C#] <pre>ITransportContext.SecuritySessionParameters = securitySessionParameters;</pre></p> <p>[VB.NET] <pre>ITransportContext.SecuritySessionParameters = securitySessionParameters</pre></p>
Default Parameters	<p>The lowest priority</p> <p>The Security Session Parameters used by default. You can change them, though it is not recommended.</p> <p>[C#] <pre>SecuritySessionServices.DefaultSecuritySession = securitySessionParameters;</pre></p> <p>[VB.NET] <pre>SecuritySessionServices.DefaultSecuritySession = securitySessionParameters</pre></p>

To get the Security Session Parameters used on the server side, use the

GenuineUtility.CurrentInvocationSecuritySession method.

Security Session Parameters is an instance of the SecuritySessionParameters class. It contains the name of the Key Provider (and the generated Security Session), invocation attributes, the invocation timeout, the type of the requested connection, the name of the requested connection, the connection inactivity timeout, the name of the remote transport user.

List of Security Session Parameters

Entity	Description
The name of the Key Provider.	A Security Session created exclusively for a specific connection by the corresponding Key Provider. Here you specify the name of the Key Provider that will spawn the Security Session.
Invocation attributes	A set of attributes specifying the invocation, transport and context behavior.
The invocation timeout.	The time span within which the response to the invocation must be received. An exception will be dispatched to the caller if the response to the invocation is not received within the time period defined by this value.
The type of the requested connection	Here is the place where you tell Connection Manager to use the Persistent, Named, Invocation or One-Way connection pattern.
The name of the requested connection	The name of the named connection to send the invocation through. Is used on the server side if you want to use the specific connection opened by the remote host.
The connection inactivity timeout	The time span after which a named or invocation connection is to be closed, if there are no messages sent or received to or from the remote host.
The name of the remote transport user	Represents a value indicating the name of the remote transport user. This value is useful if it is necessary to dispatch a .NET Remoting invocation to a particular channel. This option can be used when several channels are bound to the same Transport Context.

The SecuritySessionParameters class has different constructors. The full form of the constructor looks like this:

```
[C#]
new SecuritySessionParameters (
    SecuritySessionServices.DefaultContext.Name, // The Basic Key Provider
    SecuritySessionAttributes.None,           // No context attributes
    TimeSpan.MinValue,                         // To Inherit the value from the GenuineParameter.InvocationParameter
                                           // Transport Context parameter
    GenuineConnectionType.Persistent,         // Persistent connection
    null,                                     // The name of the connection
    TimeSpan.MinValue,                         // To Inherit the value from
                                           // the GenuineParameter.ClosePersistentConnectionAfterInactivity parameter
    string.Empty);                           // No transport user name

[VB.NET]
New SecuritySessionParameters (
    SecuritySessionServices.DefaultContext.Name, ' The Basic Key Provider
    SecuritySessionAttributes.None,             ' No context attributes
    TimeSpan.MinValue,                         ' To Inherit the value from
                                           ' the GenuineParameter.InvocationParameter Transport Context parameter
    GenuineConnectionType.Persistent,         ' Persistent connection
    Nothing,                                   ' The name of the connection
    TimeSpan.MinValue,                         ' To Inherit the value from
                                           ' the GenuineParameter.ClosePersistentConnectionAfterInactivity parameter
    string.Empty)                             ' No transport user name
```

Invocation Attributes

Attribute	Description
<i>SecuritySessionAttributes.None</i>	No attributes are requested.
<i>SecuritySessionAttributes.EnableCompression</i>	Enables the compression of the content being sent.
<i>SecuritySessionAttributes.ForceAsync</i>	Enforces the asynchronous transport approach to be used. By default, the transport approach matches the type of the message.
<i>SecuritySessionAttributes.ForceSync</i>	Enforces the synchronous transport approach to be used. By default, the transport approach matches the type of the message.
<i>SecuritySessionAttributes.UseExistentConnection</i>	The invocation must be sent through the established named connection. An exception is thrown if there is no such connection or if it has been closed. Is used on the client side when the client needs to be sure that all consecutive invocations go through the same named connection.

The following sample invokes the server's method via the established named connection, which has been used to invoke client's method earlier:

```
[C#]
SecuritySessionParameters securitySessionParameters =
    new SecuritySessionParameters (
        // use the default Key Provider
        SecuritySessionServices.DefaultContext.Name, // TODO: specify the name of your Key Provider here
        // async invocation via named connection with enabled compression
        SecuritySessionAttributes.ForceAsync |
        SecuritySessionAttributes.EnableCompression |
        SecuritySessionAttributes.UseExistentConnection,
        // wait for the reply for no more than 15 seconds
        TimeSpan.FromSeconds(15),
        // use named connections
        GenuineConnectionType.Named,
        // with the known name
        GenuineUtility.CurrentInvocationSecuritySession.ConnectionName,
        // inherit the default value
        TimeSpan.MinValue,
        // use the default transport user
        string.Empty);

using (new SecurityContextKeeper(securitySessionParameters))
{
    iRemoteHostCallback.CheckIt(parameter);
}
```

```
[VB.NET]
Dim securitySessionParameters As SecuritySessionParameters =
    New SecuritySessionParameters (
        ' use the default Key Provider
        SecuritySessionServices.DefaultContext.Name,
        ' TODO: specify the name of your Key Provider here
        ' async invocation via named connection with enabled compression
        SecuritySessionAttributes.ForceAsync Or
        SecuritySessionAttributes.EnableCompression Or
        SecuritySessionAttributes.UseExistentConnection,
        ' wait for the reply for no more than 15 seconds
        TimeSpan.FromSeconds(15),
        ' use named connections
        GenuineConnectionType.Named,
        ' with the known name
        GenuineUtility.CurrentInvocationSecuritySession.ConnectionName,
        ' inherit the default value
        TimeSpan.MinValue,
```

```
' use the default transport user
    string.Empty)

Dim securityContextKeeper As SecurityContextKeeper
Try
    securityContextKeeper = New
        SecurityContextKeeper(SecuritySessionParameters)
    iRemoteHostCallback.CheckIt(parameter)
Finally
    securityContextKeeper.Dispose()
End Try
```

3.4. Basic Security Provider

Full type: *Belikov.GenuineChannels.Security.KeyProvider_Basic*.

Security Sessions spawned by this Security Provider do not provide any security functionality. You can use the Basic Security Provider during invocations that do not require any security, or if you want to specify Security Session Parameters (invocation attributes, the type of the used connection, the invocation timeout) without additional security services.

The implementation of the Transport Context creates an instance of the Basic Security Provider and associates it with

Belikov.GenuineChannels.Security.SecuritySessionServices.DefaultContext.Name. You can use the *Belikov.GenuineChannels.Security.SecuritySessionServices.DefaultContextWithCompression* Security Session Parameters to enable data compression.

3.5. Known Symmetric Security Provider

Full type: *Belikov.GenuineChannels.Security.KeyProvider_KnownSymmetric*.

The Known Symmetric Key Provider requires a symmetric algorithm during construction. The implementation is very simple: the provided algorithm is used for encryption and decryption in all Security Sessions spawned by this Key Provider.

```
[C#]
SymmetricAlgorithm symmetricAlgorithm = SymmetricAlgorithm.Create();

// read the key
Stream keyFileStream = File.OpenRead(settings.Attributes["FILE"].Value);
byte[] key = new byte[keyFileStream.Length];
keyFileStream.Read(key, 0, key.Length);
keyFileStream.Close();

// initialize the key
symmetricAlgorithm.Key = key;
symmetricAlgorithm.Mode = CipherMode.ECB;

// Register the Key Provider for using on the invocation level.
// These lines are not necessary if you want to use it only on the
// connection level.
BasicChannelWithSecurity basicChannelWithSecurity = (BasicChannelWithSecurity) channel;
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey ("/Test/Symmetric_encryption",
    new KeyProvider_KnownSymmetric(symmetricAlgorithm) );

// Enable it by default in this Transport Context
basicChannelWithSecurity.ITransportContext.SecuritySessionParameters =
    new SecuritySessionParameters("/Test/Symmetric_encryption");

[VB.NET]
Dim symmetricAlgorithm As SymmetricAlgorithm = SymmetricAlgorithm.Create()

' read the key
Dim keyFileStream As Stream = _
    File.OpenRead(settings.Attributes("FILE").Value)
```

```

Dim key(keyFileStream.Length) As Byte
keyFileStream.Read(key, 0, key.Length)
keyFileStream.Close()

' initialize the key
symmetricAlgorithm.Key = key
symmetricAlgorithm.Mode = CipherMode.ECB

' Register the Key Provider for using on the invocation level.
' These lines are not necessary if you want to use it only on the
' connection level.
Dim basicChannelWithSecurity As BasicChannelWithSecurity = DirectCast(channel, BasicChannelWithSecurity)
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey("/Test/Symmetric_encryption", _
    New KeyProvider_KnownSymmetric(symmetricAlgorithm))

' Enable it by default in this Transport Context
basicChannelWithSecurity.ITransportContext.SecuritySessionParameters = _
    New SecuritySessionParameters("/Test/Symmetric_encryption")

```

This Security Session is useful if you generate or transmit the encryption key on your own.

3.6. Self-Establishing Symmetric Security Provider

Full type: *Belikov.GenuineChannels.Security.SecuritySession_SelfEstablishingSymmetric*.

Each Security Session spawned by this Key Provider generates a unique 256-bit Rijndael key for encryption. The key is unique for every connection. The key is delivered to the other side with the help of 1024-bit RSA encryption.

1. The server sends the public part of an asymmetric key to the client.
2. The client creates a symmetric key, encrypts it with the help of the received public key and sends it back to the server.
3. Both the client and the server use the symmetric key to perform encryption.
4. The implementation has a defense against racing (if both peers start sending a public key, one of the peers will win the race).

3.7. SSPI Security Providers

The full type of the client Key Provider:

Belikov.GenuineChannels.Security.SSPI.KeyProvider_SspiClient.

The full type of the server Key Provider:

Belikov.GenuineChannels.Security.SSPI.KeyProvider_SspiServer.

Security Sessions spawned by the client SSPI Key Provider implement the SSPI client logic.

1. The client SSPI Key Provider is initialized with credentials (login, password, optional domain name, or a null reference to use the security context of the current process):

[C#]

```

credential = new NetworkCredential(this.XmlSettings.Attributes["USER"].Value,
    this.XmlSettings.Attributes["PASSWORD"].Value);
if (this.XmlSettings.Attributes["DOMAIN"] != null)
    credential.Domain = this.XmlSettings.Attributes["DOMAIN"].Value;

BasicChannelWithSecurity basicChannelWithSecurity = (BasicChannelWithSecurity) channel;
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey ( KEYNAME,
    new KeyProvider_SspiClient(flags, package, credential, targetName));

// specify a null reference instead of credential to use the current
// process' security context
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey ( KEYNAME,

```

CLIENT SIDE

```
new KeyProvider_SspiClient(flags, package, null, targetName));
```

[VB.NET]

CLIENT SIDE

```
credential = New NetworkCredential(Me.XmlSettings.Attributes("USER").Value, _
    Me.XmlSettings.Attributes("PASSWORD").Value)
If Not Me.XmlSettings.Attributes("DOMAIN") Is Nothing Then
    credential.Domain = Me.XmlSettings.Attributes("DOMAIN").Value
End If

Dim basicChannelWithSecurity As BasicChannelWithSecurity = DirectCast(channel, BasicChannelWithSecurity)
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey(KEYNAME, _
    New KeyProvider_SspiClient(flags, package, credential, targetName))

" specify a null reference instead of credential to use the current
" process' security context
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey(KEYNAME, _
    New KeyProvider_SspiClient(flags, package, Nothing, targetName))
```

- The server SSPI Key Provider must be initialized with the same settings and associated with the same name:

[C#]

SERVER SIDE

```
BasicChannelWithSecurity basicChannelWithSecurity = (BasicChannelWithSecurity) iConnectionProvider.Channel;
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey ( KEYNAME, new KeyProvider_SspiServer(flags, package));
```

[VB.NET]

SERVER SIDE

```
Dim basicChannelWithSecurity As BasicChannelWithSecurity = _
    DirectCast(iConnectionProvider.Channel, BasicChannelWithSecurity)
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey (KEYNAME, new KeyProvider_SspiServer(flags, package))
```

- Enforce the Key Provider on the Connection Level or Invocation Level. See the corresponding sections above for examples and explanations.

SSPI Security Sessions support client authentication (all versions of Windows including Windows 98 and Windows NT), strong encryption (Windows XP, Windows 2003 or better) and integrity checking (all versions of Windows including Windows 98 and Windows NT). SSPI Server Security Sessions impersonate the security context. Security context delegation is supported only if the Kerberos package is used.

The SSPI Key Provider supports three SSPI packages: Negotiate, NTLM and Kerberos. The recommended package is NTLM.

Supported SSPI features

Feature	Description
Encryption	Enables traffic Encryption. Supported only by Windows XP, Windows 2003 or better.
Signing	Enables content integrity checking.
Impersonation	Enables security context impersonation on the server side.
Delegation	Enables security context delegation on the server side.

If you need to analyze the impersonated context, check roles or impersonate it in another thread, you can use the *SecuritySession_SspiServer.WindowsIdentity* property on the server side while processing a request that came from the client.

[C#]

```
using System.Security.Principal;
using Belikov.GenuineChannels.Security.SSPI;

public void Do(...)
{
```



```
// first, fetch the current SSPI Security Session
SecuritySession_SspiServer securitySession_SspiServer =
    (SecuritySession_SspiServer) GenuineUtility.CurrentRemoteHost.GetSecuritySession (
        GenuineUtility.CurrentInvocationSecuritySessionParameters.Name, null);

// and get Windows Identity
WindowsIdentity windowsIdentity = securitySession_SspiServer.WindowsIdentity;
}

[VB.NET]
Imports System.Security.Principal
Imports Belikov.GenuineChannels.Security.SSPI

Public Sub Do()
    ' first, fetch the current SSPI Security Session
    Dim securitySession_SspiServer As SecuritySession_SspiServer =
        DirectCast(GenuineUtility.CurrentRemoteHost.GetSecuritySession( _
            GenuineUtility.CurrentInvocationSecuritySessionParameters.Name, Nothing), SecuritySession_SspiServer)

    ' and get Windows Identity
    Dim windowsIdentity As WindowsIdentity = securitySession_SspiServer.WindowsIdentity
End Sub
```

SSPI does not send passwords or encryption keys through the connection. It sends only password hashsums and generates encryption keys based on it.

3.8. Zero Proof Authorization Security Providers

The full type of the client Key Provider:

Belikov.GenuineChannels.Security.ZeroProofAuthorization.KeyProvider_ZpaClient.

The full type of the server Key Provider:

Belikov.GenuineChannels.Security.ZeroProofAuthorization.KeyProvider_ZpaServer.

The Zero Proof Authorization Security Provider provides authentication, encryption and content integrity checking services. The client ZPA Key Provider is initialized with a login and a password. The server ZPA Key Provider is initialized with the password provider returning passwords for the requested logins.

The login is a serializable object. The password is a string. None of them can be a null reference or be empty.

The following algorithm is executed by every Security Session.

1. The client or the server initiates establishing a Security Session.
2. The server generates an arbitrary salt sequence with a variable length and sends it to the client. The length of the sequence varies between 128 and 256 bytes.
3. The client receives the sequence and calculates the HMAC-SHA1 (160-bit) keyed hash sum based on the password and the received sequence. The result and the serialized login are sent to the server.
4. The server calculates the same hashsum and compares the calculated value with the received sequence. If both byte sequences are equal, the client is considered to have proved its password knowledge.
5. Both the client and the server generate a 256-bit key and a 128-bit IV vector based on the salt and the password for symmetric encryption (Rijndael encryption) and a 192-bit key for content integrity checking.

As a result, we have a Security Provider that

1. Does not send open passwords through the network. Only salt and salt-based hashsums are sent. The salt is unique for each established connection and Security Session.
2. Performs 256-bit Rijndael symmetric encryption in CBC or EBC chaining modes

- depending on the requested encryption functionality. Keys are unique for every connection and Security Session. Keys are not sent through the network.
3. Performs content integrity checking based on either the MAC-3DES-CBC (64-bit) keyed hash algorithm or based on the HMAC-SHA1 (160-bit) keyed hash algorithm depending on the requested functionality. Keys are not sent through the network.
 4. Provides the client login during every invocation.

Supported options

Feature	Description
ElectronicCodebookEncryption	Enables Rijndael 256-bit encryption in the electronic codebook (ECB, no chaining between blocks) mode.
CipherBlockChainingEncryption	Enables Rijndael 256-bit encryption with the chaining mode of operation of the cipher (CBC). This is the strongest generally available mode of operation. Unless you explicitly have a reason to choose another chaining mode, you probably want to use this type of cipher block chaining.
Mac3DesCbcSigning	Enables the packet integrity checking based on the MAC-3DES-CBC (64-bit) keyed hash algorithm.
HmacSha1Signing	Enables the packet integrity checking based on the HMAC-SHA1 (160-bit) keyed hash algorithm.

To use the Zero Proof Authorization Security Provider, it is necessary

1. Create an instance of the ***Belikov.GenuineChannels.Security.ZeroProofAuthorization.KeyProvider_ZpaClient*** class on the client side, initialize it with the login and the password, and specify required security options:

[C#]

CLIENT SIDE

```
BasicChannelWithSecurity basicChannelWithSecurity =
    (BasicChannelWithSecurity) iConnectionProvider.Channel;
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey ( keyName,
    new KeyProvider_ZpaClient(ZpaFeatureFlags.ElectronicCodebookEncryption |
    ZpaFeatureFlags.Mac3DesCbcSigning, login, password));
```

[VB.NET]

CLIENT SIDE

```
Dim basicChannelWithSecurity As BasicChannelWithSecurity = _
    DirectCast(iConnectionProvider.Channel, BasicChannelWithSecurity)
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey (keyName, _
    New KeyProvider_ZpaClient (ZpaFeatureFlags.ElectronicCodebookEncryption _
    Or ZpaFeatureFlags.Mac3DesCbcSigning, login, password))
```

2. Create an instance of the ***Belikov.GenuineChannels.Security.ZeroProofAuthorization.KeyProvider_ZpaServer*** class on the server side; specify the required options and an instance of the object implementing the ***IAuthorizationManager*** interface:

[C#]

SERVER SIDE

```
BasicChannelWithSecurity basicChannelWithSecurity =
    (BasicChannelWithSecurity) iConnectionProvider.Channel;
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey ( keyName,
```

```
new KeyProvider_ZpaServer(ZpaFeatureFlags.ElectronicCodebookEncryption |
ZpaFeatureFlags.Mac3DesCbcSigning, iAuthorizationManager));
```

[VB.NET]

SERVER SIDE

```
Dim basicChannelWithSecurity As BasicChannelWithSecurity = _
DirectCast(iConnectionProvider.Channel, BasicChannelWithSecurity)
basicChannelWithSecurity.ITransportContext.IKeyStore.SetKey ( keyName, _
new KeyProvider_ZpaServer(ZpaFeatureFlags.ElectronicCodebookEncryption Or _
ZpaFeatureFlags.Mac3DesCbcSigning, iAuthorizationManager))
```

You must specify the same security options. Otherwise, the server will not be able to decrypt the content or perform integrity checking correctly.

The implementation of the IAuthorizationManager interface is usually simple. You get a login object and return the corresponding password. You are expected to fire a serializable exception if the provided login is wrong.

[C#]

```
using Belikov.GenuineChannels.Security;
using Belikov.GenuineChannels.Security.ZeroProofAuthorization;
using Belikov.GenuineChannels.TransportContext;

public class ZpaPasswordProvider : IAuthorizationManager
{
    public string GetPassword(object providedLogin)
    {
        // TODO: check the login and fetch the corresponding password
        return this.Password;
    }
}
```

[VB.NET]

```
Imports Belikov.GenuineChannels.Security;
Imports Belikov.GenuineChannels.Security.ZeroProofAuthorization;
Imports Belikov.GenuineChannels.TransportContext;

Public Class ZpaPasswordProvider
    Implements IAuthorizationManager

    Public Function GetPassword(ByVal providedLogin As String) As String
        " TODO: check the login and fetch the corresponding password
        Return Me.Password
    End Function
End Class
```

3. Enforce the Key Provider on the connection or invocation level. See the corresponding sections above for samples and explanations.

3.9. Chaining mode

Symmetric encryption and keyed hash algorithms define different chaining modes of operation of the cipher. The chaining mode of an algorithm determines whether and how the encryption of one block of data influences the encryption of the next block of data.

Use the CBC chaining mode only in Connection Level Security Sessions or for invocations making up an ordered sequence that cannot be disrupted by reestablishing or any asynchronous matters.

Use the ECB chaining mode if you are not sure whether the messages make up an ordered sequence.

Please note the following. Genuine Channels can send messages asynchronously. There are absolutely no guarantees that the message sent first will be received before the message sent after the first one. The GTCP implementation can open several connections, GHTTP uses two HTTP

keep-alive connections, reconnection can change a message position in the queue and so on.

SSPI integrity checking (content signing), SSPI encryption and symmetric algorithms in the CBC chaining mode require an uninterrupted non-repeatable (Broadcast Engine can send the same message twice under certain circumstances) **stream**. Consequently, you should distinguish all business and usual calls your application makes. Then, for example, if you make some call in one thread synchronously, it is an ideal candidate to use a separate SSPI Security Session. Just initialize the separate SSPI key provider and use this Security Session only during those calls. If you have asynchronous calls that can never be made concurrently, you can also use stream-dependent features.

It is not a Genuine Channels issue. If you implement some custom encryption, you will face the same problem.

4. Events in distributed environment

You can find a quick overview of different approaches to implement events in distributed environment at <http://www.genuinechannels.com/Content.aspx?id=27&type=1>.

The Broadcast Engine is a technology supported by Genuine Channels allowing you to deliver an invocation to several recipients concurrently. Each recipient must be an instance of the class derived from the MarshalByRefObject class and supporting a particular interface. There are three types of recipients supported: local, locally created and remote.

The local recipient is an MBR object located in the current appdomain.

The locally created recipient is a transparent proxy to the remote MBR object created by Activator.GetObject, Activator.CreateInstance or any similar means.

The remote object is a transparent proxy representing a remote MBR object.

```
[C#]
public interface IEventInterface
{
    object OnEvent1(object parameter);
}

public class LocalRecipient : MarshalByRefObject, IEventInterface
{
    public object OnEvent1(object parameter) { . . . }
    public LocalRecipient Get Yourself() { return this; }
}

static void Main(string[] args)
{
    // local recipient
    LocalRecipient local = new LocalRecipient();

    // locally created recipient
    IEventInterface locallyCreated = (IEventInterface) Activator.GetObject(typeof(IEventInterface),
        "ghhttp://myServerUrl/KnownEventReceiver.rem");

    // and remote recipient
    IEventInterface remote = locallyCreated.Get Yourself();
}

[VB.NET]
Public Interface IEventInterface
    Function OnEvent1(ByVal parameter As object) As Object
End Interface

Public class LocalRecipient
    Inherits MarshalByRefObject
    Implements IEventInterface

    Public Function OnEvent1(parameter As Object) As Object
    . . .
End Function

Public Function Get Yourself() As LocalRecipient
    Return Me
End Function
End Class

Public Shared Sub Main()
    ' local recipient
    Dim local As LocalRecipient = New LocalRecipient()

    ' locally created recipient
    Dim locallyCreated As IEventInterface = DirectCast(Activator.GetObject(GetType(IEventInterface), _
        "ghhttp://myServerUrl/KnownEventReceiver.rem"), IEventInterface)

    ' and remote recipient
    Dim remote As IEventInterface = locallyCreated.Get Yourself()
End Sub
```

All recipients are invoked concurrently, for example, if you have several local recipients, every recipient is invoked in its own thread taken from the Thread Pool. The Broadcast Engine uses

asynchronous approach to invoke locally created and remote recipients. Therefore, do not enforce the synchronous mode of sending when you use the Broadcast Engine, otherwise Connection Manager will send invocations sequentially in spite of the fact that the invocations are asynchronous.

In order to use the Broadcast Engine, you should follow these steps:

1. Create and initialize an instance of the Dispatcher class.
2. Add recipients supporting the appropriate interface to the created dispatcher.
3. Invoke a transparent proxy provided by the dispatcher. The invocation will be delivered to all recipients.
4. Analyze the results of the invocation.

There are two classes you will deal with: Dispatcher and ResultCollector.

Dispatcher is a collection of recipients supporting the same interface. If you need to have two absolutely different set of recipients, create two dispatchers. If you need to invoke only a subset of recipients, use a filter. If you need recipients to receive different events, declare several methods in the interface.

You can add or remove recipients whenever you want. All methods and properties provided by the Dispatcher class are thread-safe.

Dispatcher supports two modes of invocation: synchronous and asynchronous. In the synchronous mode the invocation returns when either all recipients answer or the timeout elapses. In the asynchronous mode you provide a callback that handles results of invocations. The control is returned almost immediately, as soon as all calls are initiated.

The advantages of the Broadcast Engine:

1. All calls are made concurrently.
2. A failure in invocation delivery to one recipient does not influence other recipients.
3. The Broadcast Engine automatically recognizes whether a recipient is available via a true multicast channel.
4. If a recipient available via a true multicast channel does not confirm message receiving within the specified period of time, the Broadcast Engine automatically repeats the invocation via the usual channel.
5. The Broadcast Engine automatically takes care of sponsorship of the attached MBR recipients. That is, it registers a sponsor for every added object and removes the sponsor when the recipient is excluded.
6. The Broadcast Engine guarantees that each recipient will be invoked only once per invocation.
7. The Broadcast Engine supports context-based filtering which allows you to apply different filters to different broadcast invocations concurrently and without any restrictions.
8. The Broadcast Engine can automatically exclude recipients that do not reply to the specified number of invocations.
9. The Broadcast Engine fires an event every time a recipient is excluded manually or due to persistent failures.

4.1. Broadcast Engine in practice

The primary goal of the Broadcast Engine is to simplify sending the same message to several recipients.

The secondary goal is to provide a transparent mechanism for using true multicast channels.

First, you **declare a well-known interface** in an assembly available on the client and on the server. Each method used for broadcasting must return an instance of the Object class. For example,

```
[C#]
public interface ITestCall
{
    // Method must return either Object or an instance of the
    // ResultCollector class
    object CheckBroadcast(int a, int b, byte[] array);
}

[VB.NET]
Public Interface ITestCall
    " Method must return either Object or an instance of the
    " ResultCollector class
    Function CheckBroadcast(ByVal a As Integer, ByVal b As Integer, ByVal array As byte()) As Object
End Interface
```

Then it is necessary to **create an instance** of the *Dispatcher* class on the server side (*Belikov.GenuineChannels.BroadcastEngine.Dispatcher*):

```
[C#]
public Dispatcher _dispatcher = new Dispatcher(typeof(ITestCalling));

[VB.NET]
Public _dispatcher As Dispatcher = New Dispatcher(GetType(ITestCalling))
```

and **attach a number of local, locally created or remote MBR objects**:

```
[C#]
public void RegisterObject(ITestCall mbr)
{
    _dispatcher.Add((MarshalByRefObject) mbr);
}

[VB.NET]
Public Sub RegisterObject(ByVal mbr As ITestCall)
    _dispatcher.Add(DirectCast(mbr, MarshalByRefObject))
End Sub
```

Of course, provided *MarshalByRefObject* should implement the specified interface (*ITestCall*), otherwise the object will not be able to receive broadcast messages.

Finally, you use **a transparent proxy for sending broadcast messages** and receiving client responses:

```
[C#]
// get transparent proxy
ITestCalling iTestCalling = (ITestCalling) _dispatcher.TransparentProxy;

// and use it
object o = iTestCalling.CheckBroadcast(20, 30, array);
ResultCollector resultCollector = (ResultCollector) o;

[VB.NET]
" get transparent proxy
Dim iTestCalling As ITestCalling = DirectCast(_dispatcher.TransparentProxy, ITestCalling)

" and use it
Dim o As object = iTestCalling.CheckBroadcast(20, 30, array)
Dim resultCollector As ResultCollector = DirectCast(o, ResultCollector)
```

The call will be finished when either all clients reply or after the timeout. Each dispatcher works in the synchronous mode by default. That is, the thread where you invoke dispatcher's transparent proxy is held until all clients reply. You can switch a dispatcher into the asynchronous mode, in which the specified handler is called as soon as an invocation is

completed.

```
[C#]
// this snippet enables asynchronous mode
// specify the timeout
_dispatcher.ReceiveResultsTimeout = TimeSpan.FromSeconds(40);
// set the callback
_dispatcher.BroadcastCallFinishedHandler = new BroadcastCallFinishedHandler(this.BroadcastCallFinishedHandler);
// and turn on the asynchronous mode. This assignment fails if
// _dispatcher.BroadcastCallFinishedHandler is null.
_dispatcher.CallsAsync = true;

public void BroadcastCallFinishedHandler(Dispatcher dispatcher, IMessage message, ResultCollector resultCollector)
{
    // analyze broadcast results
}

[VB.NET]
" this snippet enables asynchronous mode
" specify the timeout
_dispatcher.ReceiveResultsTimeout = TimeSpan.FromSeconds(40)
" set the callback
_dispatcher.BroadcastCallFinishedHandler = New BroadcastCallFinishedHandler(Me.BroadcastCallFinishedHandler)
" and turn on the asynchronous mode. This assignment fails if
" _dispatcher.BroadcastCallFinishedHandler is Nothing.
_dispatcher.CallsAsync = true

Public Sub BroadcastCallFinishedHandler(ByVal dispatcher As Dispatcher, _
    ByVal message As IMessage, ByVal resultCollector As ResultCollector)
    " analyze broadcast results
End Sub
```

When you initiate broadcast sending, an instance of the **ResultCollector** class is returned. It automatically starts collecting client responses. As it was mentioned before, the provided callback will be called either when all clients reply or after the timeout. **ResultCollector** contains successful responses (the **Successful** property) and failed responses (the **Failed** property).

```
[C#]
public void BroadcastCallFinishedHandler(Dispatcher dispatcher, IMessage message, ResultCollector resultCollector)
{
    lock(resultCollector)
    {
        foreach(DictionaryEntry entry in resultCollector.Successful)
        {
            IMessageReturnMessage iMethodReturnMessage = (IMessageReturnMessage) entry.Value;
            // here you've got client responses
            // including -out and -ref parameters

            Console.WriteLine("Returned object = {0}", iMethodReturnMessage.ReturnValue.ToString());
        }

        foreach(DictionaryEntry entry in resultCollector.Failed)
        {
            string mbrUri = (string) entry.Key;
            Exception ex = null;
            if (entry.Value is Exception)
                ex = (Exception) entry.Value;
            else
                ex = ((IMessageReturnMessage) entry.Value).Exception;

            MarshalByRefObject failedObject = dispatcher.FindObjectByUri(mbrUri);
            Console.WriteLine("Receiver {0} has failed. Error: {1}", mbrUri, ex.Message);

            // here you have failed MBR object (failedObject)
            // and Exception (ex)
        }
    }
}

[VB.NET]
Public Sub BroadcastCallFinishedHandler(Dispatcher dispatcher, IMessage message, ResultCollector resultCollector)
    SyncLock resultCollector
        Dim entry As DictionaryEntry
        For Each entry In resultCollector.Successful
```

```

        Dim iMethodReturnMessage As IMethodReturnMessage = DirectCast(entry.Value, IMethodReturnMessage)
        ' here you've got client responses
        ' including -out and -ref parameters

        Console.WriteLine("Returned object = {0}", iMethodReturnMessage.ReturnValue.ToString())
    Next

    For Each entry In ResultCollector.Failed
        Dim mbrUri As String = DirectCast(entry.Key, String)
        Dim ex As Exception = Nothing
        If entry.Value Is Exception Then
            ex = DirectCast(entry.Value, Exception)
        Else
            ex = (DirectCast(entry.Value,
                IMethodReturnMessage)).Exception
        End If

        Dim failedObject As MarshalByRefObject = Dispatcher.FindObjectByUri(mbrUri)
        Console.WriteLine("Receiver {0} has failed. Error: {1}", mbrUri, ex.Message)

        ' here you have failed MBR object (failedObject)
        ' and Exception (ex)
    Next
End SyncLock
End Sub

```

It is recommended to lock the *resultCollector* object while processing the results of an invocation.

4.2. Filtering recipients

If a set of recipients is not permanent and can be or must be changed for each invocation or group of invocations, you should use Multicast Filters.

In order to specify to what recipients invocations must be dispatched, you should provide an object implementing the *IMulticastFilter* interface. There are three different levels where a filter can be specified: thread context, local place and Dispatcher.

Let's study a simple example.

The known layer contains an event provider interface:

```

[C#]
/// <summary>
/// IFilteringEventProvider.
/// </summary>
public interface IFilteringEventProvider
{
    void Subscribe(IEventReceiver iEventReceiver, int divisor);
}

[VB.NET]
''' <summary>
''' IFilteringEventProvider.
''' </summary>
Public Interface IFilteringEventProvider
    Function Subscribe(ByVal iEventReceiver As IEventReceiver, ByVal divisor As Integer)
End Interface

```

A client provides a divisor (a number) while subscribing; the server keeps it in the Client Session. The Client Session is provided as a tag during registration of the client recipient. Therefore, the Multicast Filter always has access to Client Sessions and can make decisions based on the values stored in the Client Session.

```

[C#]
public class FilteringEventProvider : MarshalByRefObject,
    IFilteringEventProvider
{
    /// <summary>

```

```

/// Adds the receiver to the receiver list.
/// </summary>
/// <param name="iEventReceiver"></param>
/// <param name="divisor"></param>
public void Subscribe(IEventReceiver iEventReceiver, int divisor)
{
    // store divisor into Client Session
    GenuineUtility.CurrentSession["divisor"] = divisor;
    // and specify Client Session during adding the receiver
    this.Dispatcher.Add((MarshalByRefObject) iEventReceiver, GenuineUtility.CurrentSession);
}

```

[VB.NET]

Public Class FilteringEventProvider
Inherits MarshalByRefObject
Implements IFilteringEventProvider

```

' Adds the receiver to the receiver list.
Public Sub Subscribe(ByVal iEventReceiver As IEventReceiver, ByVal divisor As Integer)
    ' store divisor into Client Session
    GenuineUtility.CurrentSession("divisor") = divisor
    ' and specify Client Session during adding the receiver
    Me.Dispatcher.Add(DirectCast(iEventReceiver, MarshalByRefObject), GenuineUtility.CurrentSession)
End Sub

```

The server implements a simple filter that gets a number during its creation and collects those clients whose divisors produce the zero remainder after division:

[C#]

```

// server implementation
private class DivisorFilter : IMulticastFilter
{
    // get the number
    public DivisorFilter(int number)
    {
        this._number = number;
    }
    private int _number;

    public object[] GetReceivers(object[] cachedReceiverList, IMessage iMessage)
    {
        object[] resultList = new object[cachedReceiverList.Length];
        int resultListPosition = 0;

        // by all receivers
        for (int i = 0; i < cachedReceiverList.Length; i++)
        {
            // get ReceiverInfo instance
            ReceiverInfo receiverInfo = cachedReceiverList[i] as ReceiverInfo;
            if (receiverInfo == null)
                continue;

            // fetch the session
            ISessionSupport session = (ISessionSupport) receiverInfo.Tag;
            // and check on the call condition
            if ( (this._number % (int) session["divisor"]) == 0 )
                resultList[ resultListPosition++ ] = receiverInfo;
        }

        return resultList;
    }
}

```

[VB.NET]

```

" server implementation
Private Class DivisorFilter
    Implements IMulticastFilter

    " get the number
    Public Sub DivisorFilter(ByVal number As Integer)
        Me._number = number
    End Sub

    Private _number As Integer

```

```

Public Function GetReceivers(ByVal cachedReceiverList As Object(), ByVal iMessage As IMessage) As Object()

    Dim resultList(cachedReceiverList.Length) As Object
    Dim resultListPosition As Integer = 0

    " by all receivers
    Dim i As Integer
    For i = 0 To (cachedReceiverList.Length-1)
        " get ReceiverInfo instance
        Dim receiverInfo As ReceiverInfo = DirectCast(cachedReceiverList(i), ReceiverInfo)
        If Not Is Nothin(receiverInfo) Then
            " fetch the session
            Dim session As ISessionSupport = DirectCast(receiverInfo.Tag, ISessionSupport)
            " and check on the call condition
            If (Me._number Mod Convert.ToInt16(session("divisor"))) = 0 Then
                resultList(resultListPosition) = receiverInfo
                resultListPosition = resultListPosition + 1
            End If
        End If
    Next i

    Return resultList
End Function
End Class

```

The server application enforces the filter when it makes a broadcast invocation. There are different contexts where a broadcast filter can be specified. The following code applies the filter to the local scope only. Therefore, this filter is not applied to other possible invocations of this dispatcher made concurrently in other threads.

```

[C#]
/// <summary>
/// Fires an event.
/// </summary>
/// <param name="ignored1"></param>
/// <param name="ignored2"></param>
public void FireEvent(object ignored1, bool ignored2)
{
    int number = this._random.Next() % 100;

    // DivisorFilter will be used for all Dispatcher's calls
    // in the local place within using statement
    using(new DispatcherFilterKeeper(new DivisorFilter(number)))
    {
        this.IEventReceiver.ReceiveString( string.Format("This is a string N: {0}.", number) );
    }
}

[VB.NET]
''' <summary>
''' Fires an event.
''' </summary>
''' <param name="ignored1"></param>
''' <param name="ignored2"></param>
Public Sub FireEvent(ByVal ignored1 As object, ByVal ignored2 As Boolean)
    Dim number As Integer = Me._random.Next() Mod 100

    " DivisorFilter will be used for all Dispatcher's calls
    " in the local place within using statement
    Dim dispatcherFilterKeeper As DispatcherFilterKeeper

    Try
        dispatcherFilterKeeper = New DispatcherFilterKeeper(New DivisorFilter(number))
        Me.IEventReceiver.ReceiveString( String.Format("This is a string N: {0}.", number) )
    Finally
        dispatcherFilterKeeper.Dispose()
    End Try
End Sub

```

The filter makes decisions what recipients are to be invoked. To make such a decision, the filter has access to the current invoking message; you can cast the provided *IMessage* to ***IMethodCallMessage*** and access invocation parameters. However, this is highly unrecommended. Besides ***IMethodCallMessage***, the filter has access to the tag value of each recipient object. The

tag value is specified when a recipient is added to the dispatcher. Finally, the filter can use values defined in the filter's class. You can add anything you need during filter construction.

In this example, the filter uses Client Sessions specified in the recipient tags. Usually, it is a very convenient pattern.

Let us take a look at the *IMulticastFilter* interface

```
[C#]
/// <summary>
/// Provides methods for filtering receivers.
/// </summary>
public interface IMulticastFilter
{
    /// <summary>
    /// Returns receivers that should be called.
    /// </summary>
    /// <param name="cachedReceiverList">All registered receivers (read-only cached array).</param>
    /// <param name="iMessage">The call.</param>
    /// <returns>Receivers that will be called.</returns>
    object[] GetReceivers(object[] cachedReceiverList, IMessage iMessage);
}

[VB.NET]
''' <summary>
''' Provides methods for filtering receivers.
''' </summary>
Public Interface IMulticastFilter
    ''' <summary>
    ''' Returns receivers that should be called.
    ''' </summary>
    ''' <param name="cachedReceiverList">All registered receivers (read-only cached array).</param>
    ''' <param name="iMessage">The call.</param>
    ''' <returns>Receivers that will be called.</returns>
    Function GetReceivers(ByVal cachedReceiverList As Object(), ByVal iMessage As IMessage) As object()
End Interface
```

The filter gets a cached array of ReceiverInfo instances. **Never do anything with it!** It can be provided to several filters in different threads concurrently; another array will be created after a recipient is added or removed. Your filter should return a similar array of ReceiverInfo instances. Null entries will be ignored, that is why I created an array having the same size and left unnecessary elements null.

```
[C#]
public object[] GetReceivers(object[] cachedReceiverList, IMessage iMessage)
{
    object[] resultList = new object[cachedReceiverList.Length];
    int resultListPosition = 0;

[VB.NET]
Public Function GetReceivers(ByVal cachedReceiverList As Object(), ByVal iMessage As IMessage) As object ()
    Dim resultList(cachedReceiverList.Length) As Object
    Dim resultListPosition As Integer = 0
```

There are three ways to specify a filter. The most recommended way is to specify it in the local scope. In this case, it will be used for invocations made in the current thread within the scope. Besides the local scope, you can specify a filter to be used in the current thread or for all invocations made via a particular dispatcher instance. Just take a look at the fragment below and you will understand everything.

```
[C#]
IMulticastFilter filter1, filter2, filter3;
Dispatcher dispatcher1, dispatcher2, dispatcher3;

// by default all calls made via dispatcher1 will be filtered by filter1
dispatcher1.IMulticastFilter = filter1;
// and dispatcher2 uses filter2 by default
```

```

dispatcher2.IMulticastFilter = filter2;

// filter1 is used
((IReceiver) dispatcher1.TransparentProxy).CallMe();
// filter2 is used
((IReceiver) dispatcher2.TransparentProxy).CallMe();
// no filter is used
((IReceiver) dispatcher3.TransparentProxy).CallMe();

// WARNING: always use "using" statement. DispatcherFilterKeeper
// is a structure that will not be released by Garbage Collection.

// all calls inside 'using' statement are filtered out by filter1
using(new DispatcherFilterKeeper(filter1))
{
    // All calls within this scope are filtered by filter1.
    // Dispatcher or thread context settings do not matter,
    // filter1 will catch all calls via any of Dispatcher instance.

    // filter1 is used
    ((IReceiver) dispatcher1.TransparentProxy).CallMe();
    // filter1 is used
    ((IReceiver) dispatcher2.TransparentProxy).CallMe();
    // filter1 is used
    ((IReceiver) dispatcher3.TransparentProxy).CallMe();
}

// filter1 is used
((IReceiver) dispatcher1.TransparentProxy).CallMe();
// filter2 is used
((IReceiver) dispatcher2.TransparentProxy).CallMe();
// no filter is used
((IReceiver) dispatcher3.TransparentProxy).CallMe();

// now set thread's filter
// Thread context filter have higher priority than Dispatcher's filter
// but lower than local scope filter.
Dispatcher.SetCurrentFilter(filter3);

// filter3 is used
((IReceiver) dispatcher1.TransparentProxy).CallMe();
// filter3 is used
((IReceiver) dispatcher2.TransparentProxy).CallMe();
// filter3 is used
((IReceiver) dispatcher3.TransparentProxy).CallMe();

// all calls inside using statement will be filtered out by filter2
using(new DispatcherFilterKeeper(filter2))
{
    // filter2 is used
    ((IReceiver) dispatcher1.TransparentProxy).CallMe();
    // filter2 is used
    ((IReceiver) dispatcher2.TransparentProxy).CallMe();
    // filter2 is used
    ((IReceiver) dispatcher3.TransparentProxy).CallMe();
}

// still filter3 is used
((IReceiver) dispatcher1.TransparentProxy).CallMe();
// still filter3 is used
((IReceiver) dispatcher2.TransparentProxy).CallMe();
// still filter3 is used
((IReceiver) dispatcher3.TransparentProxy).CallMe();

// now remove thread's filter, so Dispatcher's settings start working again
Dispatcher.SetCurrentFilter(null);

// filter1 is used
((IReceiver) dispatcher1.TransparentProxy).CallMe();
// filter2 is used
((IReceiver) dispatcher2.TransparentProxy).CallMe();
// no filter is used
((IReceiver) dispatcher3.TransparentProxy).CallMe();

```

[\[VB.NET\]](#)
Dim filter1, filter2, filter3 As IMulticastFilter
Dim dispatcher1, dispatcher2, dispatcher3 As Dispatcher

```

" by default all calls made via dispatcher1 will be filtered by filter1
dispatcher1.IMulticastFilter = filter1
" and dispatcher2 uses filter2 by default
dispatcher2.IMulticastFilter = filter2

" filter1 is used
DirectCast(dispatcher1.TransparentProxy, IReceiver).CallMe()
" filter2 is used
DirectCast(dispatcher2.TransparentProxy, IReceiver).CallMe()
" no filter is used
DirectCast(dispatcher3.TransparentProxy, IReceiver).CallMe()

" WARNING: always use "Try ... Finally" statement. DispatcherFilterKeeper
" is a structure that will not be released by Garbage Collection.

" all calls inside 'Try ... Finally' statement are filtered out by filter1
Dim dispatcherFilterKeeper As DispatcherFilterKeeper
Try
    dispatcherFilterKeeper = New DispatcherFilterKeeper(filter1)
" All calls within this scope are filtered by filter1.
" Dispatcher or thread context settings do not matter,
" filter1 will catch all calls via any of Dispatcher instance.

    " filter1 is used
    DirectCast(dispatcher1.TransparentProxy, IReceiver).CallMe()
" filter1 is used
    DirectCast(dispatcher2.TransparentProxy, IReceiver).CallMe()
" filter1 is used
    DirectCast(dispatcher3.TransparentProxy, IReceiver).CallMe()
Finally
    dispatcherFilterKeeper.Dispose()
End Try

" filter1 is used
DirectCast(dispatcher1.TransparentProxy, IReceiver).CallMe()
" filter2 is used
DirectCast(dispatcher2.TransparentProxy, IReceiver).CallMe()
" no filter is used
DirectCast(dispatcher3.TransparentProxy, IReceiver).CallMe()

" now set thread's filter
" Thread context filter have higher priority than Dispatcher's filter
" but lower than local scope filter.
Dispatcher.SetCurrentFilter(filter3);

" filter3 is used
DirectCast(dispatcher1.TransparentProxy, IReceiver).CallMe()
" filter3 is used
DirectCast(dispatcher2.TransparentProxy, IReceiver).CallMe()
" filter3 is used
DirectCast(dispatcher3.TransparentProxy, IReceiver).CallMe()

" all calls inside Try ... Finally statement will be filtered out by filter2
Try
    dispatcherFilterKeeper = New DispatcherFilterKeeper(filter2)
    " filter2 is used
    DirectCast(dispatcher1.TransparentProxy, IReceiver).CallMe()
" filter2 is used
    DirectCast(dispatcher2.TransparentProxy, IReceiver).CallMe()
" filter2 is used
    DirectCast(dispatcher3.TransparentProxy, IReceiver).CallMe()
Finally
    dispatcherFilterKeeper.Dispose()
End Try

" still filter3 is used
DirectCast(dispatcher1.TransparentProxy, IReceiver).CallMe()
" still filter3 is used
DirectCast(dispatcher2.TransparentProxy, IReceiver).CallMe()
" still filter3 is used
DirectCast(dispatcher3.TransparentProxy, IReceiver).CallMe()

" now remove thread's filter, so Dispatcher's settings start working again
Dispatcher.SetCurrentFilter(Nothing)

" filter1 is used
DirectCast(dispatcher1.TransparentProxy, IReceiver).CallMe()
" filter2 is used

```



```
DirectCast(dispatcher2.TransparentProxy, IReceiver).CallMe()
" no filter is used
DirectCast(dispatcher3.TransparentProxy, IReceiver).CallMe()
```

It is worth mentioning that filters set in the thread context or in the local scope context control all invocations made via any Dispatcher.

4.3. Dispatcher parameters

You can tune the process of broadcasting via the Dispatcher properties.

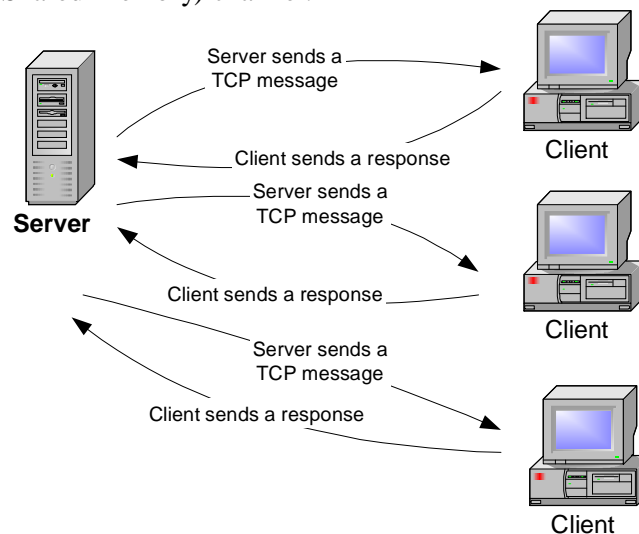
Property name	Type	Default value	Comment
<i>ReceiveResultsTimeout</i>	TimeSpan	120 seconds	Represents a value indicating the period of time after which the result collector stops receiving responses from recipients.
<i>CallIsAsync</i>	Boolean	false	The false value of this property means that the results of broadcast invocations will be available immediately after completion of a transparent proxy invocation. The true value means that the results of broadcast invocations will be available to the method specified by BroadcastCallFinishedHandler called asynchronously.
<i>BroadcastCallFinishedHandler</i>	Delegate	null	Represents a value specifying what method to be called after a broadcast invocation is completed in the asynchronous manner. That is, this method will be invoked only if the value of the CallIsAsync property is true.
<i>MaximumNumberOfConsecutiveFailsToExcludeReceiverAutomatically</i>	Integer	4	As soon as a recipient (an MBR object) does not confirm receiving this number of times in a row, it will be automatically removed from the list of recipients. Set it to zero to disable automatic exclusion.
<i>MaximumNumberOfConsecutiveFailsToEnableSimulationMode</i>	Integer	4	As soon as a recipient (an MBR object) that is available via a true broadcast channel does not receive invocations this number of times in a row, it is automatically switched to the usual mode and starts receiving events via the usual channel. However, if the Dispatcher finds out that the recipient becomes reachable via the true broadcast channel again, it is immediately switched back into the multicast mode.

<i>IgnoreRecurrentCalls</i>	Boolean	true	Represents a value determining whether the caller will assign a unique GUID to each call. It will enable clients to execute the call only once even if it is received via different transports (for example, via a true multicast channel and the usual GTCP channel).
<i>IMulticastFilter</i>	Filter	null	Represents a value indicating what Multicast Filter is to be used if no filter is specified in the current thread context or in the local scope context.
<i>BroadcastReceiverHasBeenExcludedEventHandler</i>	Event		Occurs when a recipient is excluded from the Dispatcher list of recipients either manually or due to persistent failures.
<i>AsyncEventStub</i>	Method		Implements the default implementation of the asynchronous callback. If you need to enforce the asynchronous type of invocation and do not want to process its results, you can create a delegate pointing to this method.
<i>TransparentProxy</i>	Transparent Proxy		The transparent proxy that is used to invoke recipients.

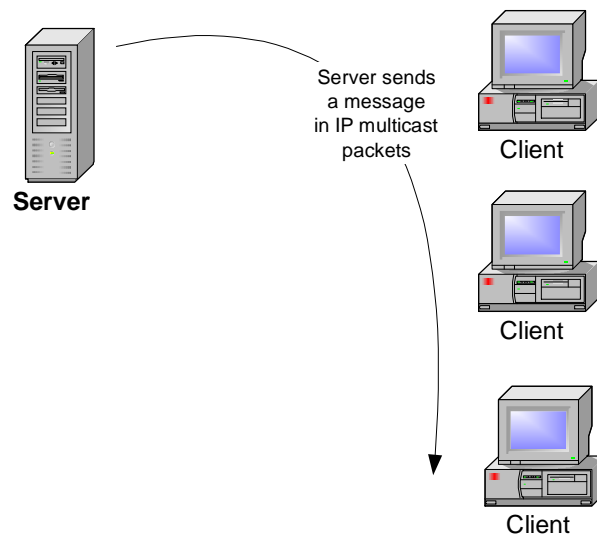
4.4. IP Multicasting

True broadcast channels provide unidirectional transport from one sender to several recipients. Right now Genuine Channels provide only one channel that uses IP multicasting for this purpose. The future Broadcast MSMQ channel is another good example.

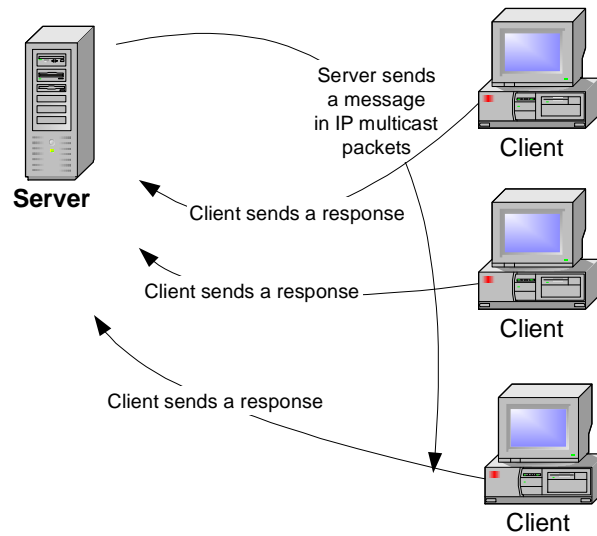
When you use the Broadcast Engine, all recipients are in the Simulation mode by default. The simulation mode is a mode when recipients receive messages via the usual (GTCP, GHTTP, GXHTTP, GUDP or Shared Memory) channel.



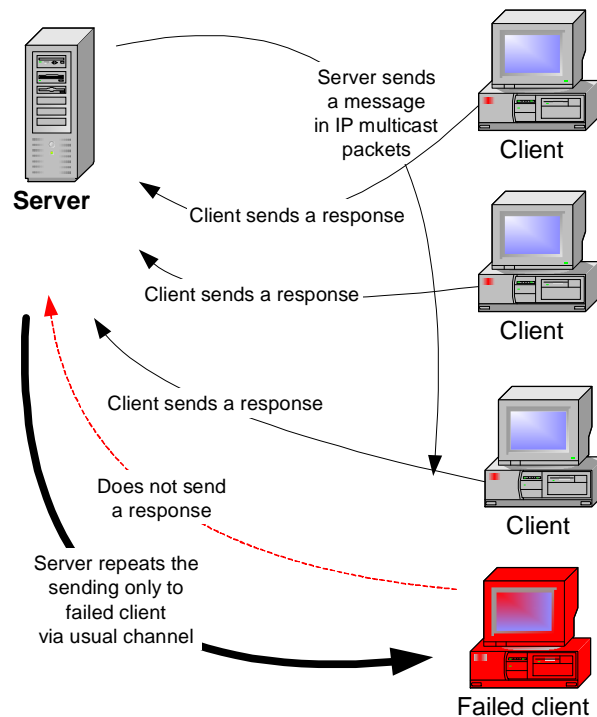
The Broadcast Engine serializes each broadcast invocation request only once for all recipients, but the content is transmitted via .NET Remoting sinks and network IP packets separately for each recipient. The more clients you have, the more CPU, memory and network resources are consumed.



When you use IP multicasting, the broadcast invocation hits the wire only once, saving you a lot of resources. But in contrast to GTCP, GHTTP or Shared Memory, IP multicasting uses datagrams and does not provide any guarantees that the message will be delivered to all or appropriate clients.



That's why clients are expected to answer via the usual channels. The Broadcast Engine collects results of sending and analyzes whether all the expected recipients have successfully received the invocation and answered a result.



The Broadcast Engine automatically repeats the invocation via the usual channel if the delivery has not been confirmed within the specified time span (*ReceiveResultsTimeout*).

4.4.1. Client Side

All clients receiving messages via a true broadcast channel get absolutely identical data. It is impossible to specify a particular client recipient that should be called, therefore, the invocation is sent to a Court, not to a particular MBR object.

The Court is a correspondence between a string and a couple of MBR objects. The first MBR object is invoked when a message associated with the court is received. The second MBR object specifies the way responses should be sent.

If no object is associated with a specific court, the *BroadcastUnknownCourtReceived* event is fired. This event signals that a client receives messages from the server via IP multicasting but does not know what to do with them.

The following piece of code demonstrates how you can associate an MBR object with a court (*Belikov.GenuineChannels.BroadcastEngine.CourtCollection*):

```

[C#]
// SAO
FetchCurrentBusinessObject fetchCurrentBusinessObject = new FetchCurrentBusinessObject();

// acquire MBR object from the server via the usual channel
IDolt serversEntryPoint = fetchCurrentBusinessObject.GetServerMbr();

Class1 broadcastReceiverObject = new Class1();

// and attach MBR receiver to the specific court
CourtCollection.Attach("TestCourt", broadcastReceiverObject, (MarshalByRefObject) serversEntryPoint);

// give it to the server which will attach it to the Dispatcher instance
fetchCurrentBusinessObject.SetClientBroadcastReceiver(broadcastReceiverObject);
  
```

```
[VB.NET]
" SAO
Dim fetchCurrentBusinessObject As FetchCurrentBusinessObject = New FetchCurrentBusinessObject()

' acquire MBR object from the server via the usual channel
Dim serversEntryPoint As IDolt = fetchCurrentBusinessObject.GetServerMbr()

Dim broadcastReceiverObject As Class1 = new Class1()

' and attach MBR receiver to the specific court
CourtCollection.Attach("TestCourt", broadcastReceiverObject, DirectCast(serversEntryPoint, MarshalByRefObject))

' give it to the server which will attach it to the Dispatcher instance
fetchCurrentBusinessObject.SetClientBroadcastReceiver(broadcastReceiverObject)
```

As you see, an instance of the `Class1` class is associated with the “TestCourt” court. The remote MBR object obtained from the server is specified as a return path. Every time a true multicast channel receives an invocation, it invokes the first MBR object (the instance of the `Class1` class in the example) and sends the response via the channel through which the `serversEntryPoint` MBR object was obtained. If you do not provide a return path, the response will be sent via the GUDP channel.

4.4.2. Server Side

The server sets up the IP multicast channel and adds its sender to the dispatcher. The GUDP channel implements the ***IBroadcastSenderProvider*** interface, which can be used to get a sender object for adding it to the dispatcher. If you know the name of the GUDP channel, you can get the channel by its name and cast it to ***IBroadcastSenderProvider***.

```
[C#]
// configures broadcast dispatcher to use IP multicast channel
IBroadcastSenderProvider iBroadcastSenderProvider = ChannelServices.GetChannel("BroadcastSender1") as
IBroadcastSenderProvider;

_dispatcher.Add(iBroadcastSenderProvider.GetBroadcastSender("TestCourt"));

[VB.NET]
" configures broadcast dispatcher to use IP multicast channel
Dim iBroadcastSenderProvider As IBroadcastSenderProvider = _
    DirectCast (ChannelServices.GetChannel("BroadcastSender1"),IBroadcastSenderProvider)
_dispatcher.Add(iBroadcastSenderProvider.GetBroadcastSender("TestCourt"))
```

The dispatcher automatically detects what recipients can receive invocations via the true broadcast channel and stops sending messages to them directly.

Initially, all recipients operates in the simulation mode and the first message is sent via the usual channel as well as via the true broadcast sender. If the ***IgnoreRecurringCall*** property is true (it is true by default), the invocation is executed only once.

After adding an IP multicasting sender to the dispatcher, every invocation made via the dispatcher’s transparent proxy is sent to all concerned hosts.

5. Channels

5.1. GTCP

The GTCP Connection Manager uses a single TCP connection for implementing a bidirectional link. The GTCP Connection Manager can work as a client and/or server connection manager concurrently. If you specify a port, GTCP Connection Manager can accept incoming connections and act as server.

When the TCP connection is broken, the client application automatically tries to reestablish it smoothly, according to the specified Transport Context Parameters. The server side waits for reconnection for the time span specified by the *MaxTimeSpanToReconnect* Transport Context parameter.

If the connection is not reestablished within the specified time span, the remote host is considered to be disconnected and all resources associated with this connection are released. If the connection is reestablished, all messages accumulated in the queue are sent through the reestablished connection. Generally, no messages can be lost if the connection has been broken and reestablished.

The GTCP Connection Manager supports persistent and named connection patterns and synchronous and asynchronous transport approaches.

5.1.1. Configuration via a configuration file

The use of configuration files gives you convenient configuration means without recompiling the application.

This is an example of the server configuration file. The red fragment illustrates how you can specify the GTCP channel.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="5M"
        sponsorshipTimeout="3M"
        renewOnCallTime="5M"
        leaseManagerPollTime="1M"
      />

      <service>
        <wellknown
          mode="Singleton"
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          objectUri="FetchCurrentBusinessObject.rem"
        />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

```

  <channels>
    <channel type="Belikov.GenuineChannels.GenuineTcp.GenuineTcpChannel, GenuineChannels"
      MaxContentSize="500000"
      PersistentConnectionSendPingAfterInactivity="1000"
      MaxQueuedItems="100"
      MaxTotalSize="10000000"
      port="8737" />
  </channels>

```

In order to initiate listening to several ports, it is necessary either to register several channels or to register one channel and use the *TransportContext.StartListening* method.

```
<channels>
  <channel type="Belikov.GenuineChannels. GenuineTcp.GenuineTcpChannel, GenuineChannels" port="8080" />
  <channel type="Belikov.GenuineChannels. GenuineTcp.GenuineTcpChannel, GenuineChannels" interface="."
    port="8081" />
  <channel type="Belikov.GenuineChannels. GenuineTcp.GenuineTcpChannel, GenuineChannels" interface="192.168.1.1"
    port="8082" />
</channels>
```

Each channel will start a listener thread to listen to the specified port.
You can add custom sinks as it is designed by .NET Remoting.
The following example creates sink chains with a binary formatter and **SecureServerChannelSinkProvider**.

```
<channel type="Belikov.GenuineChannels. GenuineTcp.GenuineTcpChannel, GenuineChannels">
  <clientProviders>
    <formatter ref="binary" />
    <provider type="Toub.Remoting. SecureServerChannelSinkProvider, SecureChannel"
      algorithm="DES" oaep="false" maxRetries="1" />
  </clientProviders>
</channel>
```

The sample of the client config file:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="5M"
        sponsorshipTimeout="3M"
        renewOnCallTime="5M"
        leaseManagerPollTime="1M"
      />

      <client>
        <wellknown
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          url="gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem"
        />
      </client>
      <channels>
        <channel type="Belikov.GenuineChannels. GenuineTcp.GenuineTcpChannel, GenuineChannels"
          MaxContentSize = "500000"
          PersistentConnectionSendPingAfterInactivity = "1000"
          MaxQueuedItems = "1000"
          MaxTotalSize = "10000000"
          TcpReconnectionTries = "1000"
          TcpSleepBetweenReconnection = "500"
          ConnectTimeout = "240000"
        >

        <!-- You can skip the sink description. Anyway, the binary formatter with typeFilterLevel set to Full will be set up by default. -->
        <serverProviders>
          <formatter ref="binary" />
        </serverProviders>
      </channel>
    </application>
  </system.runtime.remoting>
</configuration>
```

You are expected to create only one channel for connecting to any number of servers.
If you create several channels, only one will be used.

Having written the configuration file, you can configure .NET Remoting with the help of the **System.Runtime.Remoting.RemotingConfiguration.Configure** method:

```
[C#]
RemotingConfiguration.Configure("Your file name.config");
```


[\[VB.NET\]](#)`RemotingConfiguration.Configure("Your file name.config")`

5.1.2. Programmatic configuration

Programmatic configuration is very useful if you want to specify channel parameters at run time. This sample sets up the GTCP channel accepting inbound connections to port 8737.

[\[C#\]](#)

```
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.DotNetRemotingLayer;
using Belikov.GenuineChannels.GenuineTcp;

IDictionary props = new Hashtable();
props["name"] = "GTCP1";
props["priority"] = "100";
props["port"] = "8737";

// Null entries specify the default formatters.
BinaryServerFormatterSinkProvider srv = GenuineUtility.GetDefaultServerSinkChain();

// or (absolutely the same)
BinaryServerFormatterSinkProvider srv = null;

// or
BinaryServerFormatterSinkProvider srv = new BinaryServerFormatterSinkProvider();

BinaryClientFormatterSinkProvider clnt = new BinaryClientFormatterSinkProvider();

GenuineTcpChannel channel = new GenuineTcpChannel(props, clnt, srv);
ChannelServices.RegisterChannel(channel);

WellKnownServiceTypeEntry WKSTE = new WellKnownServiceTypeEntry(
typeof(KnownObjects.FetchCurrentBusinessObject), "FetchCurrentBusinessObject.rem", WellKnownObjectMode.Singleton);
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);
```

[\[VB.NET\]](#)

```
Imports System.Collections
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Lifetime
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels.DotNetRemotingLayer
Imports Belikov.GenuineChannels.GenuineTcp

Dim props As IDictionary = new Hashtable()
Props("name") = "GTCP1"
Props("priority") = "100"
Props("port") = "8737"

" Null entries specify the default formatters.
Dim srv As BinaryServerFormatterSinkProvider = GenuineUtility.GetDefaultServerSinkChain()

" or (absolutely the same)
Dim srv As BinaryServerFormatterSinkProvider = Nothing

" or
Dim srv As BinaryServerFormatterSinkProvider = New BinaryServerFormatterSinkProvider()

Dim clnt As BinaryClientFormatterSinkProvider = New BinaryClientFormatterSinkProvider()

Dim channel As GenuineTcpChannel = New GenuineTcpChannel(props, clnt, srv)
ChannelServices.RegisterChannel(channel)

Dim WKSTE As WellKnownServiceTypeEntry = New WellKnownServiceTypeEntry( _
    GetType(KnownObjects.FetchCurrentBusinessObject), _
    "FetchCurrentBusinessObject.rem", WellKnownObjectMode.Singleton)
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE)
```

Do not forget to add a reference to the *System.Runtime.Remoting.dll* assembly. Parameter names are case insensitive. Genuine Channels accept only strings as values of the parameters during initialization (and only during initialization).

Setting up the GTCP channel on client side:

```
[C#]
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.DotNetRemotingLayer;
using Belikov.GenuineChannels.GenuineTcp;

IDictionary props = new Hashtable();
GenuineTcpChannel channel = new GenuineTcpChannel(props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownClientTypeEntry remotetype = new WellKnownClientTypeEntry (
    typeof(KnownObjects.FetchCurrentBusinessObject),
    @"gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem");
RemotingConfiguration.RegisterWellKnownClientType(remotetype);

[VB.NET]
Imports System.Collections
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Lifetime
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels.DotNetRemotingLayer
Imports Belikov.GenuineChannels.GenuineTcp

Dim props As IDictionary = New Hashtable()
Dim channel As GenuineTcpChannel = New GenuineTcpChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

Dim remotetype As WellKnownClientTypeEntry = New WellKnownClientTypeEntry( _
    GetType(KnownObjects.FetchCurrentBusinessObject), _
    "gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem")
RemotingConfiguration.RegisterWellKnownClientType(remotetype)
```

A little bit more sophisticated scenario where two sinks are involved:

```
[C#]
// provide settings
IDictionary props = new Hashtable();
props["name"] = "GTCPClient1";
props["priority"] = 100.ToString();

props["MaxContentSize"] = 500000.ToString();
props["Compression"] = "false";
props["PersistentConnectionSendPingAfterInactivity"] = 1000.ToString();
props["MaxQueuedItems"] = 100000.ToString();
props["MaxTotalSize"] = 1000000.ToString();
props["TcpReconnectionTries"] = 100.ToString();
props["TcpSleepBetweenReconnection"] = 500.ToString();

// server-side sinks
BinaryServerFormatterSinkProvider srv = new BinaryServerFormatterSinkProvider();

GenuineClientTrackerServerSinkProvider tracker = new GenuineClientTrackerServerSinkProvider();
tracker.Next = srv;

GenuineBroadcastInterceptorServerSinkProvider interceptor = new GenuineBroadcastInterceptorServerSinkProvider();
interceptor.Next = tracker;

// client-side sink
BinaryClientFormatterSinkProvider client = new BinaryClientFormatterSinkProvider();

// setup channel
GenuineTcpChannel channel = new GenuineTcpChannel(props, client, interceptor);
ChannelServices.RegisterChannel(channel);
```

```
// and WKO
WellKnownClientTypeEntry remotetype = new WellKnownClientTypeEntry(typeof(KnownObjects.FetchCurrentBusinessObject),
@"gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem");

RemotingConfiguration.RegisterWellKnownClientType(remotetype);

[VB.NET]
" provide settings
Dim props As IDictionary = New Hashtable()
props("name") = "GTCPClient1"
props("priority") = 100.ToString()

props("MaxContentSize") = 500000.ToString()
props("Compression") = "false"
props("PersistentConnectionSendPingAfterInactivity") = 1000.ToString()
props("MaxQueuedItems") = 100000.ToString()
props("MaxTotalSize") = 1000000.ToString()
props("TcpReconnectionTries") = 100.ToString()
props("TcpSleepBetweenReconnection") = 500.ToString()

" server-side sinks
Dim srv As BinaryServerFormatterSinkProvider = New BinaryServerFormatterSinkProvider()

Dim tracker As GenuineClientTrackerServerSinkProvider = New GenuineClientTrackerServerSinkProvider()
tracker.Next = srv

Dim interceptor As GenuineBroadcastInterceptorServerSinkProvider = New GenuineBroadcastInterceptorServerSinkProvider()
interceptor.Next = tracker

" client-side sink
Dim client As BinaryClientFormatterSinkProvider = New BinaryClientFormatterSinkProvider()

" setup channel
Dim channel As GenuineTcpChannel = New GenuineTcpChannel(props, client, interceptor)
ChannelServices.RegisterChannel(channel)

" and WKO
Dim remotetype As WellKnownClientTypeEntry = New
WellKnownClientTypeEntry(GetType(KnownObjects.FetchCurrentBusinessObject),
"gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem")

RemotingConfiguration.RegisterWellKnownClientType(remotetype)
```

5.1.3. GTCP Channel Parameters

These parameters are processed only by the GTCP channel. The GTCP Transport Context does not recognize and process these parameters.

All values must be provided as strings during channel initialization.

GTCP Channel Parameters		
Name	Type, default value, unit	Description
<i>interface</i>	string "0.0.0.0" (bytes)	An optional parameter specifying the interface the listening socket will be associated with.
<i>port</i>	int	An optional parameter specifying the port number the listening socket will be associated with.
<i>name</i>	string gtcp	Represents a value indicating the name of the channel. You can always get the channel by its name with the help of the ChannelServices.GetChannel method. Every channel must have a unique name.

<i>priority</i>	int 0	An integer representing the priority assigned to this channel. Higher numbers indicate a higher chance of being chosen to connect first.
<i>prefix</i>	string gtcp	A string value indicating the URL prefix served by this channel. See the “Channel Prefixes” section below for more details.

5.1.4. GTCP Transport Context Parameters

This is a complete list of all parameters supported by the GTCP Transport Context.

Message parameters		
Name	Type, default value, unit	Description
<i>MaxContentSize</i>	int 20 000 000 (bytes)	The maximum size of a single message allowed to be sent or received.
<i>MaxQueuedItems</i>	int 100 (messages)	The maximum number of items in the queue. See the Queuing section below for more details.
<i>MaxTotalSize</i>	int 20 000 000 (bytes)	The maximum total size of all messages in the queue. See the Queuing section below for more details.
<i>NoSizeChecking</i>	bool false	Enables or disables message size checking. If message size checking is off, this considerably increases performance for messages containing streams that do not support the Stream.Length property, because no exceptions are thrown and caught. The MaxContentSize and MaxTotalSize queue constraints are ignored if the value of this parameter is false. See the Queuing section below for more details.
<i>Compression</i>	bool, false	Initializes the default Security Session with compression at the Transport Context level.
<i>InvocationTimeout</i>	TimeSpan, 120 000 (milliseconds)	The invocation timeout. An exception will be dispatched to the caller if the response to the message is not received within this time period specified by this value.
<i>SyncResponses</i>	bool, true	Represents a value indicating whether to force the synchronous transport approach for response delivering.

Common Transport Parameters		
<i>ConnectTimeout</i>	TimeSpan 120 000 (milliseconds)	An exception is dispatched to the caller if no connection to the remote host is established within this time span.
<i>SecuritySessionForPersistentConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.

<i>SecuritySessionForNamedConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.
<i>SecuritySessionForInvocationConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.
<i>SecuritySessionForOneWayConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.
<i>ClosePersistentConnectionAfterInactivity</i>	TimeSpan 100 000 (milliseconds)	The period of inactivity to close opened or accepted persistent connections after.
<i>CloseNamedConnectionAfterInactivity</i>	TimeSpan 120 000 (milliseconds)	The period of inactivity to close opened or accepted named connections after.
<i>CloseInvocationConnectionAfterInactivity</i>	TimeSpan 15 000 (milliseconds)	The period of inactivity to close opened or accepted invocation connections after.
<i>CloseOneWayConnectionAfterInactivity</i>	TimeSpan 15 000 (milliseconds)	The period of inactivity to close opened or accepted one-way connections after.
<i>PersistentConnectionSendPingAfterInactivity</i>	TimeSpan 40 000 (milliseconds)	An empty message (6 byte) is sent to the remote host if there are no messages sent to the remote host within this time span.
<i>MaxTimeSpanToReconnect</i>	TimeSpan 180 000 (milliseconds)	The time span before considering a persistent connection to be broken if it is not reestablished.
<i>ReconnectionTries</i>	int, 180, (tries)	The maximum number of reconnection attempts before declaring a connection to be broken.
<i>SleepBetweenReconnections</i>	TimeSpan, 500, (milliseconds)	The time span to wait for after every reconnection failure. Do not set this value to zero due to the following reason. All events generated by Genuine Channels are distributed in separate threads, so you have a chance to receive the event that the connection is reestablished before you receive the event that the connection is being reestablished. As you see, you have 500 milliseconds to process this event.

GTCP Transport Context parameters		
Name	Type, default value, unit	Description

<i>TcpMaxSendSize</i>	Int, 150 000, (bytes)	The maximum size of a packet.
<i>TcpReadRequestBeforeProcessing</i>	bool, true	Determines whether the message is read into an intermediate storage before deserialization. GTCP Connection Manager can deserialize messages straight from the socket. If you do not use DXM and want to increase performance, you can set it to false.
<i>TcpDoNotResendMessages</i>	bool, false	Represents a value indicating whether the last synchronous message is resent if a connection is reestablished.
<i>TcpDisableNagling</i>	bool, false	Represents a value indicating whether the GTCP Connection Manager must disable nagling for opened and accepted connections.
<i>TcpPreventDelayedAck</i>	bool, false	Represents a value indicating whether the GTCP Connection Manager must send short messages to the server in order to prevent delayed acknowledgement.
<i>TcpSendBufferSize</i>	Int, see MSDN, (bytes)	Specifies the total per-socket buffer space reserved for sending. This is unrelated to the maximum message size or the size of a TCP window.
<i>TcpReceiveBufferSize</i>	Int, see MSDN, (bytes)	Specifies the total per-socket buffer space reserved for receiving. This is unrelated to the maximum message size or the size of a TCP window.

5.1.5. Events

Event name	Description
<i>GeneralConnectionEstablished</i>	The GTCP client channel has connected to the server. The GTCP server channel has accepted a connection opened by the client.
<i>GTcpConnectionAccepted</i>	The GTCP server channel has accepted a connection. You can analyze the IP address of the remote host and decline the connection. See the explanation below.
<i>GeneralConnectionClosed</i>	The GTCP server channel has released all resources associated with an appropriate client connection and will not be able to accept a reconnection from it. The GTCP client channel has closed the connection to the remote peer.
<i>GeneralConnectionReestablishing</i>	The GTCP client channel recognizes that the connection is broken but will attempt to reconnect to the server automatically. The GTCP server channel recognizes that the connection is broken and the client is expected to reestablish the connection within the specified time span.

<i>GeneralListenerStarted</i>	The GTCP Connection Manager has started listening to the specified local end point.
<i>GeneralListenerShutDown</i>	The GTCP Connection Manager has stopped listening to the specified local end point.
<i>GeneralListenerFailure</i>	The listening socket cannot accept a connection due to an exception.

If you need to filter out incoming connections by the client IP address,

1. You can associate the server's socket with a particular interface. Either specify the "interface" GTCP channel parameter or specify the interface directly during invocation of the ***TransportContext.StartListening*** method. You can call the ***TransportContext.StartListening*** method several times if you need to associate it with several interfaces.

[C#]

```
ConnectionManager connectionManager = channel.TransportContext.ConnectionManager;
connectionManager.StartListening("gtcp://192.168.3.1:8737");
connectionManager.StartListening("gtcp://192.168.4.1:8737");
connectionManager.StartListening("gtcp://127.0.0.1:8737");
```

[VB.NET]

```
Dim connectionManager As ConnectionManager = channel.TransportContext.ConnectionManager
connectionManager.StartListening("gtcp://192.168.3.1:8737")
connectionManager.StartListening("gtcp://192.168.4.1:8737")
connectionManager.StartListening("gtcp://127.0.0.1:8737")
```

2. You can intercept the GTcpConnectionAccepted event, cast the ***GenuineEventArgs.AdditionalInfo*** property to the ***ConnectionAcceptedCancellableEventParameter*** type and analyze socket's properties. Set the ***ConnectionAcceptedCancellableEventParameter.Cancel*** variable to true if you want to decline the connection.

[C#]

```
public static void GenuineChannelsEventHandler(object sender,
    GenuineEventArgs e)
{
    if (e.AdditionalInfo is ConnectionAcceptedCancellableEventParameter)
    {
        ConnectionAcceptedCancellableEventParameter parameter =
            (ConnectionAcceptedCancellableEventParameter) e.AdditionalInfo;

        // here you have
        string ipEndPointAsString = parameter.IPEndPoint.ToString();
        Socket socket = parameter.Socket;

        if (/* if something is wrong */)
            parameter.Cancel = true;
    }
}
```

[VB.NET]

```
Public Shared Sub GenuineChannelsEventHandler(ByVal sender As Object,
    ByVal e As GenuineEventArgs)

    If e.AdditionalInfo Is ConnectionAcceptedCancellableEventParameter Then
        Dim parameter As ConnectionAcceptedCancellableEventParameter = DirectCast(e.AdditionalInfo, _
            ConnectionAcceptedCancellableEventParameter)

        " here you have
        Dim ipEndPointAsString As String = parameter.IPEndPoint.ToString()
        Dim socket As Socket = parameter.Socket

        If ( ' if something is wrong ' ) Then
            parameter.Cancel = true
        End If
```



```
End If
End Sub
```

The socket of the TCP connection being established is available while establishing Connection Level Security Session and the *GTcpConnectionAccepted* event via the *GenuineUtility.CurrentSocket* property.

5.2. GHTTP

GHTTP Connection Manager uses two keep-alive HTTP connections to provide a bidirectional connection to the server without polling.

GHTTP Server Connection Manager works only inside IIS and accepts connections established by GHTTP Client Connection Manager or GXHTTP Connection Manager. GHTTP Client Connection Manager can establish connections to GHTTP Server Connection Manager or to GXHTTP Connection Manager. All GHTTP and GXHTTP Connection Managers can send several messages in one HTTP packet (the size of the actual HTTP packet can exceed the value of the *MaxContentSize* parameter). It helps a lot if you have many small messages to be sent.

5.2.1. HTTP handler

In order to set up the HTTP Server channel, you should bind the *Belikov.GenuineChannels.GenuineHttp.HttpServerHandler* HTTP handler to a particular request type.

In general, it is recommended to attach it to the .rem extension (like the native HTTP channel does) if you are not planning to use the native HTTP channels. In this case, just insert the following snippet into the <system.web> tag of your Web.config file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>

  ...

  <httpHandlers>
    <remove verb="*" path="*.rem"/>
    <add verb="*" path="*.rem" type="Belikov.GenuineChannels.GenuineHttp.HttpServerHandler, GenuineChannels"/>
  </httpHandlers>

  </system.web>
</configuration>
```

You can create your own type, but in this case it is necessary to specify it at Start -> Control Panel -> Administrative Tools -> Internet Information Services -> Choose the site or virtual directory -> Properties -> Configuration (button) -> Mappings -> Add your own extension.

It's not convenient, especially if you host your solution at ISP provider.

Alternatively, you can attach the HTTP handler to an unused type. There are a lot of unused but mapped types that direct to *System.Web.HttpForbiddenHandler*. Take a look at *C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\CONFIG\machine.config* and find the following fragment:

```
<httpHandlers>
  <add verb="*" path="trace.axd" type="System.Web.Handlers.TraceHandler"/>
  <add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory"/>
  <add verb="*" path="*.ashx" type="System.Web.UI.SimpleHandlerFactory"/>
  <add verb="*" path="*.asmx" type="System.Web.Services.Protocols.WebServiceHandlerFactory, System.Web.Services,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" validate="false"/>
```

```
<add verb="" path="*.rem" type="System.Runtime.Remoting.Channels.Http.HttpRemotingHandlerFactory,
System.Runtime.Remoting, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" validate="false"/>
<add verb="" path="*.soap" type="System.Runtime.Remoting.Channels.Http.HttpRemotingHandlerFactory,
System.Runtime.Remoting, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" validate="false"/>
<add verb="" path="*.asax" type="System.Web.HttpForbiddenHandler"/>
<add verb="" path="*.ascx" type="System.Web.HttpForbiddenHandler"/>
...
```

You can choose any of these types and redirect it to the **Belikov.GenuineChannels.GenuineHttp.HttpServerHandler** HTTP handler:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    ...

    <httpHandlers>
      <remove verb="" path="*.cs"/>
      <add verb="" path="*.cs" type="Belikov.GenuineChannels.GenuineHttp.HttpServerHandler, GenuineChannels"/>
    </httpHandlers>

  </system.web>
</configuration>
```

Then simply name your objects using the chosen extension:

```
[C#]
protected void Application_Start(Object sender, EventArgs e)
{
    //...
    WellKnownServiceTypeEntry WKSTE = new WellKnownServiceTypeEntry(
        typeof(KnownObjects.FetchCurrentBusinessObject),
        "MyBusinessProvider.cs", WellKnownObjectMode.Singleton);
    RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);
}

[VB.NET]
Protected Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
    "...
    Dim WKSTE As WellKnownServiceTypeEntry = New WellKnownServiceTypeEntry(
        GetType(KnownObjects.FetchCurrentBusinessObject), _
        "MyBusinessProvider.cs", WellKnownObjectMode.Singleton)
    RemotingConfiguration.RegisterWellKnownServiceType(WKSTE)
End Sub
```

5.2.2. Configuration via a configuration file

It is recommended to set up the HTTP server channel only programmatically. If you define the GHTTP server channel in the Web.config file, it will not work. You will have to create a separate configuration file and read it somewhere in the Global.asax Application_Start method.

```
[C#]
protected void Application_Start(Object sender, EventArgs e)
{
    GlobalLoggerContainer.Logger = new BinaryLog ( @"c:\tmp\IIS_Server_Log.txt" );

    RemotingConfiguration.Configure( @"C:\inetpub\wwwroot\CheckRemoteIIS\Server.config");
}

[VB.NET]
Protected Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
    GlobalLoggerContainer.Logger = New BinaryLog ( "c:\tmp\IIS_Server_Log.txt")

    RemotingConfiguration.Configure( "C:\inetpub\wwwroot\CheckRemoteIIS\Server.config")
End Sub
```

```
<system.runtime.remoting>
  <application>
    <service>
      <wellknown mode="Singleton"
        type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
        objectUri="MyBusinessProvider.rem" />
    </service>
    <channels>
      <channel
        type="Belikov.GenuineChannels.GenuineHttp.GenuineHttpServerChannel,
          GenuineChannels" name="ghhttp" />
    </channels>
  </application>
</system.runtime.remoting>
```

The GHTTP client channel setup:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="60M"
        sponsorshipTimeout="60M"
        renewOnCallTime="60M"
        leaseManagerPollTime="60M"
      />
      <client>
        <wellknown
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          url="ghhttp://localhost:80/CheckRemotellS/MyBusinessProvider.rem"
        />
      </client>
      <channels>
        <channel type="Belikov.GenuineChannels.GenuineHttp.GenuineHttpClientChannel, GenuineChannels"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

5.2.3. Programmatic configuration

The server channel must be named “ghhttp”. If you specify several server channels, only one with the “ghhttp” name will be used.

The GHTTP server channel:

```
[C#]
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.GenuineHttp;
using Belikov.GenuineChannels.Logbook;

IDictionary props = new Hashtable();
props["name"] = "ghhttp";
props["priority"] = "100";

// Null entries specify the default formatters.
GenuineHttpServerChannel channel = new GenuineHttpServerChannel(props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownServiceTypeEntry WKSTE = new WellKnownServiceTypeEntry( typeof(KnownObjects.FetchCurrentBusinessObject),
"a.rem", WellKnownObjectMode.Singleton );
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);

[VB.NET]
Imports System.Collections
```

```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels.GenuineHttp
Imports Belikov.GenuineChannels.Logbook

Dim props As IDictionary = New Hashtable()
props("name") = "ghttp"
props("priority") = "100"

" Null entries specify the default formatters.
Dim channel As GenuineHttpServerChannel = New GenuineHttpServerChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

Dim WKSTE As WellKnownServiceTypeEntry = New WellKnownServiceTypeEntry( _
    GetType(KnownObjects.FetchCurrentBusinessObject), "a.rem", WellKnownObjectMode.Singleton )
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE)
```

The GHTTP client channel:

```
[C#]
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.GenuineHttp;

IDictionary props = new Hashtable();
GenuineHttpClientChannel channel = new GenuineHttpClientChannel(props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownClientTypeEntry remotetype = new
    WellKnownClientTypeEntry(typeof(KnownObjects.FetchCurrentBusinessObject),
        @"http://dima/CheckIIS/FetchCurrentBusinessObject.rem");
RemotingConfiguration.RegisterWellKnownClientType(remotetype);

[VB.NET]
Imports System.Collections
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Lifetime
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels.GenuineHttp

Dim props As IDictionary = New Hashtable()
Dim channel As GenuineHttpClientChannel = New GenuineHttpClientChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

Dim remotetype As WellKnownClientTypeEntry = _
    New WellKnownClientTypeEntry(GetType(KnownObjects.FetchCurrentBusinessObject), _
        "http://dima/CheckIIS/FetchCurrentBusinessObject.rem")
RemotingConfiguration.RegisterWellKnownClientType(remotetype)
```

5.2.4. GHTTP Channel Parameters

These parameters are processed only by the GHTTP channel. The GHTTP Transport Context does not recognize and process these parameters.

All parameter values must be provided as strings.

GHTTP Channel Parameters (both client and server GHTTP channels support the same parameters)		
Name	Type, default value, unit	Description
<i>name</i>	string ghttp	Represents a value indicating the name of the channel. You can always get the channel by its name with the help of the <i>ChannelServices.GetChannel</i> method. Every channel must have a unique name.

priority	int 0	An integer representing the priority assigned to this channel. Higher numbers indicate a higher chance of being chosen to connect first.
prefix	string gtcp	A string value indicating URL prefixes services by this channel. See the “Channel Prefixes” section below for more details.

5.2.5. GHTTP Transport Context Parameters

This is a complete list of all parameters supported by GHTTP Transport Contexts.

Message parameters		
Name	Type, default value, unit	Description
MaxContentSize	int 20 000 000 (bytes)	The maximum size of a single message allowed to be sent or received.
MaxQueuedItems	int 100 (messages)	The maximum number of items in the queue. See the Queuing section below for more details.
MaxTotalSize	int 20 000 000 (bytes)	The maximum total size of all messages in the queue. See the Queuing section below for more details.
NoSizeChecking	bool false	Enables or disables message size checking. If message size checking is off, this considerably increases performance for messages containing streams that do not support the Stream.Length property, because no exceptions are thrown and caught. The MaxContentSize and MaxTotalSize queue constraints are ignored if the value of this parameter is false. See the Queuing section below for more details.
Compression	bool, false	Initializes the default Security Session with compression at the Transport Context level.
InvocationTimeout	TimeSpan, 120 000 (milliseconds)	The invocation timeout. An exception will be dispatched to the caller if the response to the message is not received within this time period specified by this value.

Common Transport Parameters		
ConnectTimeout	TimeSpan 120 000 (milliseconds)	An exception is dispatched to the caller if no connection to the remote host is established within this time span.
SecuritySessionForPersistentConnections	String null (name)	The name of the key provider to create a Security Session used at the connection level.
SecuritySessionForInvocationConnections	String null (name)	The name of the key provider to create a Security Session used at the connection level.

<i>ClosePersistentConnectionAfterInactivity</i>	TimeSpan 100 000 (milliseconds)	The period of inactivity to close opened or accepted persistent connections after.
<i>CloseInvocationConnectionAfterInactivity</i>	TimeSpan 15 000 (milliseconds)	The period of inactivity to close opened or accepted invocation connections after.
<i>PersistentConnectionSendPingAfterInactivity</i>	TimeSpan 40 000 (milliseconds)	An empty message (6 byte) is sent to the remote host if there are no messages sent to the remote host within this time span.
<i>MaxTimeSpanToReconnect</i>	TimeSpan 180 000 (milliseconds)	The time span before considering a persistent connection to be broken if it is not reestablished.
<i>ReconnectionTries</i>	int, 180, (tries)	The maximum number of reconnection attempts before declaring a connection to be broken.
<i>SleepBetweenReconnections</i>	TimeSpan, 500, (milliseconds)	The time span to wait for after every reconnection failure. Do not set this value to zero due to the following reason. All events generated by Genuine Channels are distributed in separate threads, so you have a chance to receive the event that the connection is reestablished before you receive the event that the connection is being reestablished. As you see, you have 500 milliseconds to process this event.

HTTP Transport Context parameters		
Name	Type, default value, unit	Description
<i>HttpProxyUri</i>	String	Client side only. The URI of the proxy.
<i>HttpWebUserAgent</i>	String	Client side only. The name of the web user agent.
<i>HttpAuthUserName</i>	String	Client side only. The username associated with the credentials.
<i>HttpAuthPassword</i>	String	Client side only. The password for the username associated with the credentials.
<i>HttpAuthDomain</i>	String	Client side only. The domain associated with these credentials.
<i>HttpAuthCredential</i>	NetworkCredential	Client side only. The network credentials provided for HTTP authentication.

<i>HttpUseGlobalProxy</i>	Bool false (boolean)	Client side only. Determines whether to use the proxy server specified by the GlobalProxySelection.Select property value.
<i>HttpBypassOnLocal</i>	Bool false (boolean)	Client side only. Determines the state of the WebProxy.BypassProxyOnLocal property.
<i>HttpUseDefaultCredentials</i>	Bool false (boolean)	Client side only. Determines whether it is necessary to use the default credential.
<i>HttpAllowWriteStreamBuffering</i>	Bool false (boolean)	Client side only. Determines whether it is necessary to enable the write stream buffering.
<i>HttpUnsafeConnectionSharing</i>	Bool false (boolean)	Client side only, Framework 1.1 or better. Enables high-speed NTLM-authenticated connection sharing.
<i>HttpRecommendedPacketSize</i>	Int 100 000 (bytes)	The recommended size of the HTTP request sent to the server at a time. The HTTP client Connection Manager stops adding messages to the request as soon as its size exceeds this value.
<i>HttpAuthentication</i>	Bool false	Server side only. Enables or disables security context impersonation if the HTTP Basic, digest or Integrated Windows authentication is used.
<i>HttpStoreAndProvideHttpContext</i>	Bool false	Server side only. Enables or disables storing and providing HttpContexts during invocations processed by the GHTTP server Transport Context hosted inside IIS.
<i>HttpAsynchronousRequestTimeout</i>	TimeSpan, 180 000 (milliseconds)	Client side only. Determines the maximum amount of time after which the GHTTP Client Connection Manager starts canceling web requests and initiates the process of connection reestablishing.
<i>HttpMimeMediaType</i>	String	Client side only. Defines the MIME Media type to be specified during each HTTP request.

5.2.6. Events

Event name	Description
<i>GeneralConnectionEstablished</i>	The GHTTP client channel has connected to the server.
<i>GHttpConnectionAccepted</i>	The GHTTP server channel has accepted a client connection.

<i>GeneralConnectionReestablishing</i>	The GHTTP client Transport Context has recognized that the connection is broken and started trying to reconnect to the server. The GHTTP server Transport Context has recognized that the connection is broken and started waiting for reconnection.
<i>GeneralConnectionClosed</i>	The GHTTP server channel has released all resources associated with the client connection. The GHTTP client channel has closed the connection and will not try to connect to the server anymore.
<i>GeneralServerRestartDetected</i>	The remote server has changed its GUID due to restart. The client application should resubscribe to all server events.

5.2.7. HTTP authentication

The GHTTP Transport Context supports basic, digest and integrated Windows authentication. To enable authentication, you should:

1. Specify *AuthUserName*, *AuthPassword* and, optionally, *AuthDomain* parameters on the client side.
2. If you use Framework 1.1, you can avoid authentication for every invocation by specifying the *UnsafeConnectionSharing* parameter.
3. Specify the *HttpAuthentication* parameter on the server side.

If you use the integrated Windows authentication and need to use the current security context, use the *HttpAuthCredential* parameter:

[C#]

```
IDictionary properties = new Hashtable();
properties["HttpAuthCredential"] = CredentialCache.DefaultCredentials;
ChannelServices.RegisterChannel(new GenuineHttpClientChannel(properties, null, null));
```

[VB.NET]

```
Dim properties As IDictionary = New Hashtable()
Properties("HttpAuthCredential") = CredentialCache.DefaultCredentials
ChannelServices.RegisterChannel(New GenuineHttpClientChannel(properties, Nothing, Nothing))
```

Although since release 2.4.0 the GHTTP Transport Context provides full support of HTTP authentication, it is not recommended to use it due to the following reasons.

1. Every invocation may probably cause authentication. It is slow.
2. The HTTP client Connection Manager will have to create an additional copy of the sent content (see *HttpRequest.AllowWriteStreamBuffering* property for the details).

I would advise you to use the SSPI Security Provider instead.

1. Only one Security Session is established for all invocations.
2. You can have several impersonation contexts.
3. You can use the Kerberos package for delegation.
4. Passwords are never sent openly, unlike in case with the basic authentication.

5.2.8. If HTTP channel does not work

First of all, make sure that the server runs on the expected framework. Installation of .NET Framework usually enforces the usage of this .NET Framework for all IIS-hosted projects. You can use the *C:\WINDOWS\Microsoft.NET\Framework\%X.X.XXXX\aspnet_regiis.exe* utility to

enforce the usage of the required framework for all (or only for specific) IIS-hosted projects.

Then, check that your Web.config file contains the ***Belikov.GenuineChannels.GenuineHttp.HttpServerHandler*** HTTP handler:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>

  ...

  <httpHandlers>
    <remove verb="*" path="*.rem"/>
    <add verb="*" path="*.rem" type="Belikov.GenuineChannels.GenuineHttp.HttpServerHandler, GenuineChannels" />
  </httpHandlers>

  </system.web>
</configuration>
```

Please note that the GHTTP server channel should be named “ghttp”.

After that, make sure that you have specified the correct address (URL) of the remote host. You can try to load any aspx or html page from the hosted IIS project. If you get error 404, the URL is invalid or mistyped.

It is highly recommended to use the “.rem” extension for GHTTP handlers and known MBR URIs.

5.3. GUDP

GUDP Connection Manager uses the User Datagram Protocol (UDP) as a network transport. If a source message exceeds the maximum size of the packet (*PacketSize*), it is automatically split into several packets.

Due to the fact that the UDP protocol is a connectionless protocol, UDP datagrams sent to the remote endpoint are not guaranteed to arrive, nor are they guaranteed to arrive in the same order they are sent in. The Genuine Channels UDP Connection Manager assembles received messages independently of the order and time incoming packets arrive, according to Transport Context settings. However, if at least one packet is lost, the entire message is lost.

Messages sent through the loopback (to the localhost, 127.0.0.1) never get lost until the receive buffer is overloaded. It may happen only if you send a big message and the receiving side does not have enough time to receive it. By default, the receiving buffer is 64000 bytes (the socket low water mark is 8KB, but GUDP Connection Manger increases it up to 64000 bytes).

Since the UDP transport is connectionless, there is neither GUDP server channel nor GUDP client channel. There is only one GUDP channel that can receive message from any number of remote hosts and send or reply to them. You can have any number of GUDP channels listening to the same or different ports.

The GUDP channel supports IP multicasting via the Broadcast Engine. You can have any number of IP Multicast recipients bound to the same port on the same PC. Usually, it is a very good idea to use IP multicasting in distributed software since it saves a lot of CPU, memory and network resources.

5.3.1. Configuration via a configuration file

If an application plays the role of a client, specify the “*gudp://0.0.0.0:0*” address. If it is a server application the clients are expected to connect to, just assign the port: “*gudp://0.0.0.0:PortNumber*”.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="3M"
        sponsorshipTimeout="3M"
        renewOnCallTime="3M"
        leaseManagerPollTime="3M"
      />
      <service>
        <wellknown
          mode="Singleton"
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          objectUri="FetchCurrentBusinessObject.rem"
        />
      </service>
      <channels>
        <channel
          type="Belikov.GenuineChannels.GenuineUdp.GenuineUdpChannel, GenuineChannels"
          Address="gudp://0.0.0.0:11000" />
        </channels>
      </application>
    </system.runtime.remoting>
  </configuration>
```

5.3.2. Programmatic configuration

If an application plays the role of a client, specify the “*gudp://0.0.0.0:0*” address. If it is a server application the clients are expected to connect to, just assign the port: “*gudp://0.0.0.0:PortNumber*”.

```
[C#]
IDictionary props = new Hashtable();
props["Address"] = "gudp://0.0.0.0:0";
GenuineUdpChannel channel = new GenuineUdpChannel(props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownClientTypeEntry remotetype = new WellKnownClientTypeEntry(typeof(KnownObjects.FetchCurrentBusinessObject),
@"gudp://127.0.0.1:11000/FetchCurrentBusinessObject.rem");
RemotingConfiguration.RegisterWellKnownClientType(remotetype);

[VB.NET]
Dim props As IDictionary = New Hashtable()
props("Address") = "gudp://0.0.0.0:0"
Dim channel As GenuineUdpChannel = New GenuineUdpChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

Dim remotetype As WellKnownClientTypeEntry = _
    New WellKnownClientTypeEntry(GetType(KnownObjects.FetchCurrentBusinessObject), _
    "gudp://127.0.0.1:11000/FetchCurrentBusinessObject.rem")
RemotingConfiguration.RegisterWellKnownClientType(remotetype)
```

5.3.3. GUDP Channel Parameters

These parameters are processed only by the GUDP channel. The GUDP Transport Context does not recognize and process these parameters. All parameter values must be provided as strings.

GUDP Channel Parameters		
Name	Type, default value, unit	Description
<i>Address</i>	string	Represents the address the socket is associated with. For example, “ <i>gudp://0.0.0.0:11000</i> ”.
<i>name</i>	string gudp	Represents a value indicating the name of the channel. You can always get the channel by its name with the help of the <i>ChannelServices.GetChannel</i> method. Every channel must have a unique name.
<i>priority</i>	int 0	An integer representing the priority assigned to this channel. Higher numbers indicate a higher chance of being chosen to connect first.
<i>prefix</i>	string gudp	A string value indicating the URL prefixes served by this channel. See the Channel Prefixes section below for more details.

5.3.4. GUDP Transport Context Parameters

This is a complete list of all parameters supported by GUDP Transport Context.

Message parameters		
Name	Type, default value, unit	Description
<i>MaxContentSize</i>	int 20 000 000 (bytes)	The maximum size of a single message allowed to be sent or received.
<i>MaxQueuedItems</i>	int 100 (messages)	The maximum number of items in the queue. See the Queuing section below for more details.
<i>MaxTotalSize</i>	int 20 000 000 (bytes)	The maximum total size of all messages in the queue. See the Queuing section below for more details.
<i>NoSizeChecking</i>	bool false	Enables or disables message size checking. If message size checking is off, this considerably increases performance for messages containing streams that do not support the <i>Stream.Length</i> property, because no exceptions are thrown and caught. The <i>MaxContentSize</i> and <i>MaxTotalSize</i> queue constraints are ignored if the value of this parameter is false. See the Queuing section below for more details.
<i>Compression</i>	bool, false	Initializes the default Security Session with compression at the Transport Context level.
<i>InvocationTimeout</i>	TimeSpan, 120 000 (milliseconds)	The invocation timeout. An exception will be dispatched to the caller if the response to the message is not received within this time period specified by this value.

Common Transport Parameters		
<i>CloseInvocationConnectionAfterInactivity</i>	TimeSpan 15 000 (milliseconds)	The period of inactivity to close opened or accepted invocation connections after.

GUDP Transport Context parameters		
Name	Type, default value, unit	Description
<i>UdpJoinTo</i>	String	The address to join the IP multicast socket to. For example, 227.37.37.37.
<i>UdpTtl</i>	Int (1 by OS) (hosts)	The TTL of IP packets. The default value prevents messages from being forwarded outside a single subnetwork. Multicast datagrams with the TTL of 0 are not transmitted over any subnetwork. Multicast datagrams with the TTL of greater than 1 can be delivered to more than one subnetwork, if there are one or more multicast routers attached to the first subnetwork.
<i>UdpReceiveBuffer</i>	Int 64 000 (bytes)	The socket low water mark defining the size of the receive buffer.
<i>UdpPacketSize</i>	Int 450 (bytes)	The maximum size of a packet that can be sent.
<i>UdpMtu</i>	Int 4 500 (bytes)	The maximum possible size of a packet that can be received.
<i>UdpMulticastTo</i>	String	The network to direct the IGMP packets to. For example, gbudp://227.37.37.37:11000.
<i>UdpAssembleTimeSpan</i>	TimeSpan 100 000 (milliseconds)	Represents a value indicating how long the GUDP Connection Manager can keep packets related to the same message. If all packets of one message are not received within this time span, the message is lost.
<i>CompatibilityLevel</i>	Int 1	Since release 2.5.1, the GUDP Connection Manager provides correct URI and the version of HostInformation. Set this parameter to zero if you do not want to send this information or you need to support clients using versions 2.4.x or 2.5.0.

5.3.5. Events

Event name	Description
<i>GUdpSocketException</i>	The sending or receiving operation has a failed message.
<i>BroadcastUnknownCourtReceived</i>	The IP multicast recipient receives a message addressed to an unassigned court.

5.3.6. IP Multicasting

Having specified *UdpMulticastTo* or *UdpJoinTo* parameters during GUDP Transport Context setup, you make the GUDP Connection Manager set the broadcast flag on the UDP socket. The UDP socket starts using the IGMP protocol for sending packets to all available network nodes (via the UDP WinSock interface).

You can use IP multicasting either on a local network (255.255.255.255 and TTL=1) or in a larger area. In the second case, you should spare an IP address from class D (224.0.0.0 to 239.255.255.255), join clients to that network and set the TTL parameter to a necessary value. Please consider that some addresses are reserved for appropriate protocols, for example, ICP Router Discovery uses 224.0.0.1 and 224.0.0.2, RIP v2 uses 224.0.0.9, OSPF – 224.0.0.5 and 224.0.0.6 and so on.

If you want to use IP multicasting on the Internet, visit www.RIPE.net.

Set the server address to 127.0.0.1, bind clients to the address 0.0.0.0 and do not join clients to any networks if you want to emulate multicasting via the localhost for debugging purposes.

You can find more details about the IGMP protocol and IP Multicasting in RFC 1112 and RFC 2236.

Formally speaking, IP multicasting is a one-way transport supported by the Broadcast Engine. The Broadcast Engine can receive responses via any of usual channels (GTCP, GHTTP, GXHTTP, Shared Memory or GUDP). The GUDP channel plays the role of the IP multicast sender, which sends IGMP packets to the specified network.

If a message exceeds the value of the PacketSize parameter, the sender fragments the message into several packets. The receiving GUDP Transport Context assembles packets, defragments messages and invokes them on the specified court. Messages can arrive in any order and the same packet can be repeated, it does not matter. Please, refer to the Broadcast Engine section for further explanations about courts and true broadcast channels.

5.3.7. Configuration via a configuration file

You will have to know the server channel name in order to create a multicast sender. Therefore, it is always a good idea to specify the name parameter when you configure the GUDP channel on the server side:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="2M"
        sponsorshipTimeout="2M"
        renewOnCallTime="2M"
        leaseManagerPollTime="2M"
      />
      <service>
        <wellknown
          mode="Singleton"
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          objectUri="FetchCurrentBusinessObject.rem"
        />
      </service>
      <channels>
        <channel
          type="Belikov.GenuineChannels.GenuineTcp.GenuineTcpChannel,
            GenuineChannels"
          MaxContentSize = "500000"
          PersistentConnectionSendPingAfterInactivity = "1000"
          MaxQueuedItems = "1000000"
          MaxTotalSize = "100000000"
          port = "8737">
```

```

</channel>

    <channel type="Belikov.GenuineChannels.GenuineUdp.GenuineUdpChannel,
        GenuineChannels"
        name = "BroadcastSender1"
        Compression = "false"
        Address = "gudp://0.0.0.0:0"
        UdpMulticastTo = "gbudp://227.37.37.37:11000"
        />
    </channels>
</application>
</system.runtime.remoting>
</configuration>

```

The client configuration file:

```

<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="2M"
        sponsorshipTimeout="2M"
        renewOnCallTime="2M"
        leaseManagerPollTime="2M"
      />

      <client>
        <wellknown
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          url="gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem"
        />
      </client>
      <channels>
        <channel type="Belikov.GenuineChannels.GenuineTcp.GenuineTcpChannel,
          GenuineChannels"
          MaxContentSize = "500000"
          PersistentConnectionSendPingAfterInactivity = "1000"
          MaxQueuedItems = "1000000"
          MaxTotalSize = "1000000000"
          TcpReconnectionTries = "1000"
          TcpSleepBetweenReconnection = "500"
          MaxTimeSpanToReconnect = "240000"
          >
        </channel>
        <channel
          type="Belikov.GenuineChannels.GenuineUdp.GenuineUdpChannel, GenuineChannels"
          name="BroadcastListener1"
          Compression="false"
          Address="gudp://0.0.0.0:11000"
          UdpJoinTo="227.37.37.37"
        />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

5.3.8. Programmatic configuration

Setting up an IP multicast sender (usually on the server side):

```

[C#]
using System.Collections;
using System.Reflection;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels;
using Belikov.GenuineChannels.DotNetRemotingLayer;
using Belikov.GenuineChannels.GenuineUdp;

IDictionary props = new Hashtable();
props["name"] = "GTCPClient1";
props["priority"] = 100.ToString();
props["port"] = "8737";

```

```

props["MaxContentSize"] = "500000";
props["PersistentConnectionSendPingAfterInactivity"] = "1000";
props["MaxQueuedItems"] = "1000000";
props["MaxTotalSize"] = "100000000";
props["TcpReconnectionTries"] = "1000";

GenuineTcpChannel tcpChannel = new GenuineTcpChannel(props, null, null);
ChannelServices.RegisterChannel(tcpChannel);

props = new Hashtable();
props["name"] = "BroadcastSender1";
props["Compression"] = "false";
props["Address"] = "gudp://0.0.0.0:0";
props["UdpMulticastTo"] = "gbudp://227.37.37.37:11000";

GenuineUdpChannel broadcastChannel = new GenuineUdpChannel(props, null, null);
ChannelServices.RegisterChannel(broadcastChannel);

WellKnownServiceTypeEntry WKSTE = new WellKnownServiceTypeEntry( typeof(KnownObjects.FetchCurrentBusinessObject),
"FetchCurrentBusinessObject.rem", WellKnownObjectMode.Singleton );
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);

// configure broadcast dispatcher to use IP multicast channel
IBroadcastSenderProvider iBroadcastSenderProvider = ChannelServices.GetChannel("BroadcastSender1") as
IBroadcastSenderProvider;

Class1.BroadcastDispatcher.Add(iBroadcastSenderProvider.GetBroadcastSender("TestCourt"));
// ...

[VB.NET]
Imports System.Collections
Imports System.Reflection
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels
Imports Belikov.GenuineChannels.DotNetRemotingLayer
Imports Belikov.GenuineChannels.GenuineUdp

Dim props As IDictionary = New Hashtable()
props("name") = "GTCPClient1"
props("priority") = 100.ToString()
props("port") = "8737"

props("MaxContentSize") = "500000"
props("PersistentConnectionSendPingAfterInactivity") = "1000"
props("MaxQueuedItems") = "1000000"
props("MaxTotalSize") = "100000000"
props("TcpReconnectionTries") = "1000"

Dim tcpChannel As GenuineTcpChannel = _
    New GenuineTcpChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(tcpChannel)

props = new Hashtable()
props("name") = "BroadcastSender1"
props("Compression") = "false"
props("Address") = "gudp://0.0.0.0:0"
props("UdpMulticastTo") = "gbudp://227.37.37.37:11000"

Dim broadcastChannel As GenuineUdpChannel = New GenuineUdpChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(broadcastChannel)
Dim WKSTE As WellKnownServiceTypeEntry = New WellKnownServiceTypeEntry( _
    GetType(KnownObjects.FetchCurrentBusinessObject), "FetchCurrentBusinessObject.rem", _
    WellKnownObjectMode.Singleton )
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE)

' configure broadcast dispatcher to use IP multicast channel
Dim iBroadcastSenderProvider As IBroadcastSenderProvider = _
    DirectCast(ChannelServices.GetChannel("BroadcastSender1"), IBroadcastSenderProvider)

Class1.BroadcastDispatcher.Add(iBroadcastSenderProvider.GetBroadcastSender("TestCourt"))
' ...

```

Setting up an IP multicast recipient (usually on the client side):

```
[C#]
using System.Collections;
using System.Reflection;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels;
using Belikov.GenuineChannels.DotNetRemotingLayer;
using Belikov.GenuineChannels.GenuineUdp;

IDictionary props = new Hashtable();
props["name"] = "GTCPClient1";
props["priority"] = 100.ToString();

props["MaxContentSize"] = 500000.ToString();
props["Compression"] = "false";
props["PingTimeOut"] = 1000.ToString();
props["MaxQueuedItems"] = 1000000.ToString();
props["MaxTotalSize"] = 100000000.ToString();
props["TcpReconnectionTries"] = 1000.ToString();
props["TcpSleepBetweenReconnection"] = 500.ToString();

GenuineTcpChannel channel = new GenuineTcpChannel(props, null, null);
ChannelServices.RegisterChannel(channel);

props = new Hashtable();
props["name"] = "BroadcastSender1";
props["Compression"] = "false";
props["Address"] = "gudp://0.0.0.0:11000";
props["UdpJoinTo"] = "227.37.37.37";

GenuineUdpChannel broadcastChannel = new GenuineUdpChannel(props, null, null);
ChannelServices.RegisterChannel(broadcastChannel);

WellKnownClientTypeEntry remotetype = new WellKnownClientTypeEntry(typeof(KnownObjects.FetchCurrentBusinessObject),
@"gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem");
RemotingConfiguration.RegisterWellKnownClientType(remotetype);

[VB.NET]
Imports System.Collections
Imports System.Reflection
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels
Imports Belikov.GenuineChannels.DotNetRemotingLayer
Imports Belikov.GenuineChannels.GenuineUdp

Dim props As IDictionary = New Hashtable()
props("name") = "GTCPClient1"
props("priority") = 100.ToString()

props("MaxContentSize") = 500000.ToString()
props("Compression") = "false"
props("PingTimeOut") = 1000.ToString()
props("MaxQueuedItems") = 1000000.ToString()
props("MaxTotalSize") = 100000000.ToString()
props("TcpReconnectionTries") = 1000.ToString()
props("TcpSleepBetweenReconnection") = 500.ToString()

Dim channel As GenuineTcpChannel = New GenuineTcpChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

props = new Hashtable()
props("name") = "BroadcastSender1"
props("Compression") = "false"
props("Address") = "gudp://0.0.0.0:11000"
props("UdpJoinTo") = "227.37.37.37"

Dim broadcastChannel As GenuineUdpChannel = New GenuineUdpChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(broadcastChannel)

Dim remotetype WellKnownClientTypeEntry = New WellKnownClientTypeEntry( _
    GetType(KnownObjects.FetchCurrentBusinessObject), _
    "gtcp://127.0.0.1:8737/FetchCurrentBusinessObject.rem")
RemotingConfiguration.RegisterWellKnownClientType(remotetype)
```


5.4. Shared Memory

The shared memory Transport Context uses shared memory (take a look at the “Creating Named Shared Memory” article in MSDN) for providing a bidirectional link.

Shared memory Connection Manager provides all the necessary functionality such as pinging and support of several clients.

5.4.1. Configuration via a configuration file

The client side:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="1M"
        sponsorshipTimeout="1M"
        renewOnCallTime="1M"
        leaseManagerPollTime="1M"
      />

      <client>
        <wellknown
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          url="gshmem://FBP/GetCurrentBusinessObject.rem"
        />
      </client>

      <channels>
        <channel type="Belikov.GenuineChannels.GenuineSharedMemory.GenuineSharedMemoryChannel, GenuineChannels"
          PersistentConnectionSendPingAfterInactivity="5000"
          ClosePersistentConnectionAfterInactivity="15000"
          MaxContentSize="300000"
          SMShareSize="100000"
        />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

On the server side the address being listened to **must start from the “gshmem://”** prefix.
The server side:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="1M"
        sponsorshipTimeout="1M"
        renewOnCallTime="1M"
        leaseManagerPollTime="1M"
      />

      <service>
        <wellknown
          mode="Singleton"
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          objectUri="GetCurrentBusinessObject.rem"
        />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

```

        SMLShareSize="100000"
    >
</channel>
</channels>
</application>
</system.runtime.remoting>
</configuration>

```

5.4.2. Programmatic configuration

The Shared Memory server channel setup:

```

[C#]
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.Logbook;
using Belikov.GenuineChannels.GenuineSharedMemory;

IDictionary props = new Hashtable();
props["name"] = "GenuineChannels";
props["priority"] = "100";
props["listen"] = "gshmem://FBP";

// Null entries specify the default formatters.
GenuineSharedMemoryChannel channel = new GenuineSharedMemoryChannel(props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownServiceTypeEntry WKSTE = new WellKnownServiceTypeEntry( typeof(KnownObjects.FetchCurrentBusinessObject),
"FetchCurrentBusinessObject.rem", WellKnownObjectMode.Singleton );
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);

[VB.NET]
Imports System.Collections
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels.Logbook
Imports Belikov.GenuineChannels.GenuineSharedMemory

Dim props As IDictionary = New Hashtable()
props("name") = "GenuineChannels"
props("priority") = "100"
props("listen") = "gshmem://FBP"

" Null entries specify the default formatters.
Dim channel As GenuineSharedMemoryChannel = New GenuineSharedMemoryChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

Dim WKSTE As WellKnownServiceTypeEntry = New WellKnownServiceTypeEntry(
    GetType(KnownObjects.FetchCurrentBusinessObject), "FetchCurrentBusinessObject.rem", _
    WellKnownObjectMode.Singleton )
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE)

```

The client channel setup:

```

[C#]
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.GenuineSharedMemory;

IDictionary props = new Hashtable();
GenuineSharedMemoryChannel channel = new GenuineSharedMemoryChannel (props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownClientTypeEntry remotetype = new
    WellKnownClientTypeEntry(typeof(KnownObjects.FetchCurrentBusinessObject),
    @"gshmem://FBP/FetchCurrentBusinessObject.rem");
RemotingConfiguration.RegisterWellKnownClientType(remotetype);

[VB.NET]
Imports System.Collections
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Lifetime

```

```
Imports System.Runtime.Remoting.Channels
Imports Bellkov.GenuineChannels.GenuineSharedMemory

Dim props As IDictionary = New Hashtable()
Dim channel As GenuineSharedMemoryChannel = New GenuineSharedMemoryChannel (props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

Dim remotetype As WellKnownClientTypeEntry = _
    New WellKnownClientTypeEntry(GetType(KnownObjects.FetchCurrentBusinessObject), _
    "gshmem://FBP/GetCurrentBusinessObject.rem")
RemotingConfiguration.RegisterWellKnownClientType(remotetype)
```

5.4.3. Shared Memory Channel Parameters

These parameters are processed only by the Shared Memory channel. The Shared Memory Transport Context does not recognize and process these parameters. All parameter values must be provided as strings.

Shared Memory Channel Parameters		
Name	Type, default value, unit	Description
<i>listen</i>	string	Specifies the name of the global share through which client applications can open connections to the server. For example, "gshmem://MyShareName".
<i>name</i>	string gshmem	Represents a value indicating the name of the channel. You can always get the channel by its name with the help of the <i>ChannelServices.GetChannel</i> method. Every channel must have a unique name.
<i>priority</i>	int 0	An integer representing the priority assigned to this channel. Higher numbers indicate a higher chance of being chosen to connect first.
<i>prefix</i>	string gshmem	A string value indicating the URL prefix served by this channel. See the "Channel Prefixes" section below for more details.

5.4.4. Shared Memory Transport Context Parameters

These parameters work exclusively with Shared Memory Transport Context. In addition, you may need to specify parameters described in the Parameter Provider section.

Message parameters		
Name	Type, default value, unit	Description
<i>MaxContentSize</i>	int 20 000 000 (bytes)	The maximum size of a single message allowed to be sent or received.
<i>NoSizeChecking</i>	bool false	Enables or disables message size checking. If message size checking is off, this considerably increases performance for messages containing streams that do not support the <i>Stream.Length</i> property, because no exceptions are thrown and caught. The <i>MaxContentSize</i> and <i>MaxTotalSize</i> queue constraints are ignored if the value of this parameter is false. See the <i>Queuing</i> section below for more details.

<i>Compression</i>	bool, false	Initializes the default Security Session with compression at the Transport Context level.
<i>InvocationTimeout</i>	TimeSpan, 120 000 (milliseconds)	The invocation timeout. An exception will be dispatched to the caller if the response to the message is not received within this time period specified by this value.

Common Transport Parameters		
<i>ConnectTimeout</i>	TimeSpan 120 000 (milliseconds)	An exception is dispatched to the caller if no connection to the remote host is established within this time span.
<i>SecuritySessionForPersistentConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.
<i>ClosePersistentConnectionAfterInactivity</i>	TimeSpan 100 000 (milliseconds)	The period of inactivity to close opened or accepted persistent connections after.
<i>PersistentConnectionSendPingAfterInactivity</i>	TimeSpan 40 000 (milliseconds)	An empty message (6 byte) is sent to the remote host if there are no messages sent to the remote host within this time span.

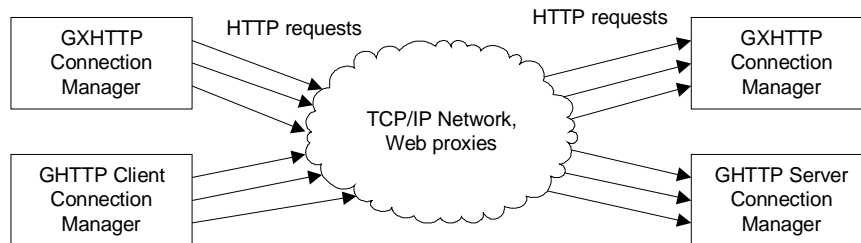
Shared Memory Transport Context parameters		
Name	Type, default value, unit	Description
<i>SMShareSize</i>	Int 300 000 (bytes)	The size of the memory share in bytes.
<i>SMSendTimeout</i>	TimeSpan 4 000 (milliseconds)	The maximum time span within which the message must be completely received by the remote host.
<i>SMSessionLocal</i>	bool, false	Represents a value indicating whether the GShMem channel must use session-local objects.

5.4.5. Events

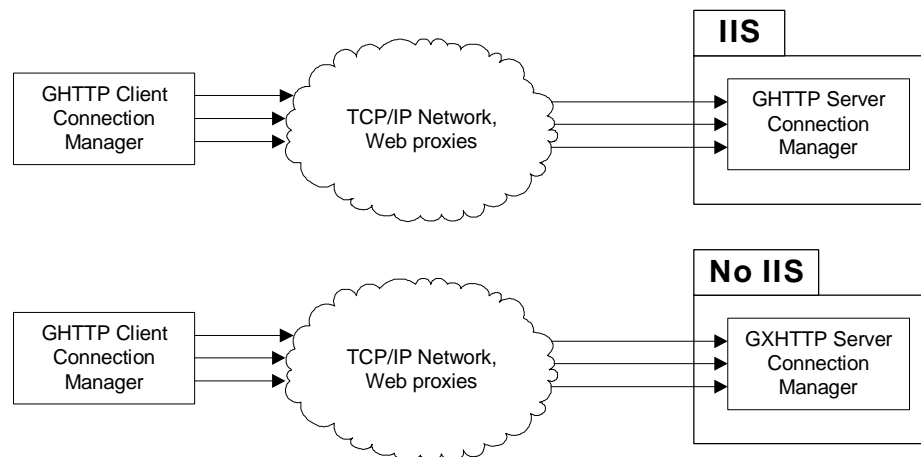
Event name	Description
<i>GeneralConnectionEstablished</i>	The shared memory connection is established.
<i>GeneralConnectionClosed</i>	The Shared Memory Transport Context has released all resources associated with the specified connection.
<i>GeneralListenerFailure</i>	The Shared Memory server does not accept incoming connections anymore.

5.5. GXHTTP Channel

GXHTTP Connection Managers use sockets to implement the HTTP client and server connection logic compatible with the GHTTP Client Connection Manager and the GHTTP Server Connection Manager. Therefore, you can use GXHTTP Connection Manager to open connections to the remote GXHTTP Connection Manager or the GHTTP Server Connection Manager and accept connections established by the remote GXHTTP Connection Manager and the GHTTP Client Connection Manager.



Although the GXHTTP Connection Manager can act as a client Connection Manager, that is, it can send HTTP requests, it is highly recommended to use GXHTTP Connection Manager on the server side only, because the GXHTTP Connection Manager does not support web proxies and HTTP authorization when act as a client.



5.5.1. Configuration via a configuration file

The use of configuration files gives you convenient configuration means without recompiling your application.

This is an example of the client configuration file. The red fragment illustrates how you can specify a GXHTTP channel on the client size.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="5M"
        sponsorshipTimeout="3M"
        renewOnCallTime="5M"
        leaseManagerPollTime="1M"
      />
    <client>
      <wellknown
```

```

        type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
        url="ghhttp://remotehost:8737/ MyBusinessProvider.rem"
    />
</client>
<channels>
    <channel type="Belikov.GenuineChannels.GenuineXHttp.GXHttp, GenuineChannels "
        PersistentConnectionSendPingAfterInactivity="1000" />
</channels>
</application>
</system.runtime.remoting>
</configuration>

```

GXHTTP server channel setup:

```

<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="60M"
        sponsorshipTimeout="60M"
        renewOnCallTime="60M"
        leaseManagerPollTime="60M"
      />
      <service>
        <wellknown
          mode="Singleton"
          type="KnownObjects.FetchCurrentBusinessObject, KnownObjects"
          objectUri="FetchCurrentBusinessObject.rem"
        />
      </service>
    </application>
    <channels>
      <channel type="Belikov.GenuineChannels.GenuineXHttp.GXHttp, GenuineChannels" port="8737" />
    </channels>
  </system.runtime.remoting>
</configuration>

```

5.5.2. Programmatic configuration

Setting up the GXHTTP channel on the server side:

```

[C#]
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.GenuineXHttp;

IDictionary props = new Hashtable();
props["name"] = "ghhttp";
props["priority"] = "100";
props["port"] = "8737";

// Null entries specify the default formatters.
GenuineXHttpChannel channel = new GenuineXHttpChannel (props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownServiceTypeEntry WKSTE = new WellKnownServiceTypeEntry( typeof(KnownObjects.FetchCurrentBusinessObject),
"a.rem", WellKnownObjectMode.Singleton );
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE);

[VB.NET]
Imports System.Collections
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels.GenuineXHttp

Dim props As IDictionary = New Hashtable()
props("name") = "ghhttp"
props("priority") = "100"
props("port") = "8737"

' Null entries specify the default formatters.
Dim channel As GenuineXHttpServerChannel = New GenuineXHttpServerChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

```

```
Dim WKSTE As WellKnownServiceTypeEntry = New WellKnownServiceTypeEntry(
GetType(KnownObjects.FetchCurrentBusinessObject), "a.rem", WellKnownObjectMode.Singleton )
RemotingConfiguration.RegisterWellKnownServiceType(WKSTE)
```

Setting up the GXHTTP channel on the client side:

```
[C#]
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Channels;
using Belikov.GenuineChannels.GenuineHttp;

IDictionary props = new Hashtable();
GenuineHttpServerChannel channel = new GenuineHttpServerChannel (props, null, null);
ChannelServices.RegisterChannel(channel);

WellKnownClientTypeEntry remotetype = new WellKnownClientTypeEntry(
    typeof(KnownObjects.FetchCurrentBusinessObject),
    @"ghhttp://dima:8737/GetCurrentBusinessObject.rem");
RemotingConfiguration.RegisterWellKnownClientType(remotetype);

[VB.NET]
Imports System.Collections
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Lifetime
Imports System.Runtime.Remoting.Channels
Imports Belikov.GenuineChannels.GenuineHttp

Dim props As IDictionary = New Hashtable()
Dim channel As GenuineHttpClientChannel = New GenuineHttpClientChannel(props, Nothing, Nothing)
ChannelServices.RegisterChannel(channel)

Dim remotetype As WellKnownClientTypeEntry = New
    WellKnownClientTypeEntry(GetType(KnownObjects.FetchCurrentBusinessObject),
        "ghhttp://dima:8737/GetCurrentBusinessObject.rem")
RemotingConfiguration.RegisterWellKnownClientType(remotetype)
```

5.5.3. GXHTTP Channel Parameters

These parameters are processed only by the GXHTTP channel. The GXHTTP Transport Context does not recognize and process these parameters.

All parameter values must be provided as strings.

GXHTTP Channel Parameters		
Name	Type, default value, unit	Description
<i>name</i>	string ghhttp	Represents a value indicating the name of the channel. You can always get the channel by its name with the help of the <i>ChannelServices.GetChannel</i> method. Every channel must have a unique name.
<i>priority</i>	int 0	An integer representing the priority assigned to this channel. Higher numbers indicate a higher chance of being chosen to connect first.
<i>prefix</i>	string gtcp	A string value indicating URL prefixes services by this channel. See the “Channel Prefixes” section below for more details.
<i>interface</i>	string “0.0.0.0” (bytes)	A string representing the interface the listening socket will be associated with.
<i>port</i>	int	An integer defining the port number the listening socket will be associated with.

5.5.4. GHTTP Transport Context Parameters

This is a complete list of all parameters supported by the GHTTP Transport Context.

Message parameters		
Name	Type, default value, unit	Description
<i>MaxContentSize</i>	int 20 000 000 (bytes)	The maximum size of a single message allowed to be sent or received.
<i>MaxQueuedItems</i>	int 100 (messages)	The maximum number of items in the queue. See the Queuing section below for more details.
<i>MaxTotalSize</i>	int 20 000 000 (bytes)	The maximum summary size of all messages in the queue. See the Queuing section below for more details.
<i>NoSizeChecking</i>	bool false	Enables or disables message size checking. If message size checking is off, this considerably increases performance for messages containing streams that do not support the Stream.Length property, because no exceptions are thrown and caught. The MaxContentSize and MaxTotalSize queue constraints are ignored if the value of this parameter is false. See the Queuing section below for more details.
<i>Compression</i>	bool, false	Initializes the default Security Session with compression at the Transport Context level.
<i>InvocationTimeout</i>	TimeSpan, 120 000 (milliseconds)	The invocation timeout. An exception will be dispatched to the caller if the response to the message is not received within this time period specified by this value.

Common Transport Parameters		
<i>ConnectTimeout</i>	TimeSpan 120 000 (milliseconds)	An exception is dispatched to the caller if no connection to the remote host is established within this time span.
<i>SecuritySessionForPersistentConnections</i>	String null (name)	The name of the key provider to create a Security Session used at the connection level.
<i>ClosePersistentConnectionAfterInactivity</i>	TimeSpan 100 000 (milliseconds)	The period of inactivity to close opened or accepted persistent connections after.

<i>PersistentConnectionSendPingAfterInactivity</i>	TimeSpan 40 000 (milliseconds)	An empty message (6 byte) is sent to the remote host if there are no messages sent to the remote host within this time span. On the client side this parameter specifies the timeout value for SENDER connections. On the server side this parameter specifies the timeout value for LISTENER connections.
<i>MaxTimeSpanToReconnect</i>	TimeSpan 180 000 (milliseconds)	The time span before considering a persistent connection to be broken if it is not reestablished.
<i>ReconnectionTries</i>	int, 180, (tries)	The maximum number of reconnection attempts before declaring a connection to be broken.
<i>SleepBetweenReconnections</i>	TimeSpan, 500, (milliseconds)	The time span to wait for after every reconnection failure. Do not set this value to zero due to the following reason. All events generated by Genuine Channels are distributed in separate threads, so you have a chance to receive the event that the connection is reestablished before you receive the event that the connection is being reestablished. As you see, you have 500 milliseconds to process this event.

XHTTP Transport Context parameters		
Name	Type, default value, unit	Description
<i>XHttpReadHttpMessageTimeout</i>	TimeSpan, 60000, (milliseconds)	Server side only. A time span defining the period of time during which an HTTP request must be received completely. If it is not received within this time span, the connection is closed.

5.5.5. Events

Event name	Description
<i>GeneralConnectionEstablished</i>	The GXHTTP channel has connected to the server.
<i>GHttpConnectionAccepted</i>	The GXHTTP server channel has accepted a connection from the client.

<i>GeneralConnectionReestablishing</i>	<p>A connection established to the server is broken and the GXHTTP Connection Manager is trying to reconnect to the server.</p> <p>An accepted connection is broken and the GXHTTP Connection Manager will wait until it is reestablished.</p>
<i>GeneralConnectionClosed</i>	<p>The GXHTTP server channel has released all resources associated with the GXHTTP connection.</p>
<i>GeneralServerRestartDetected</i>	<p>The remote server has changed its GUID due to restart. All URIs (and obtained MBR-derived objects) became invalid. The local host should resubscribe to all remote events.</p>

6. Channel Events

The channel event subsystem provides means to capture and analyze network and lifetime events generated by Genuine Channels. Every Genuine Channel implements the *Belikov.GenuineChannels.DotNetRemotingLayer.IGenuineEventProvider* interface providing the *GenuineChannelsEvent* member.

This example below illustrates how you can attach a handler to events generated by a particular channel:

[C#]

```
using Belikov.GenuineChannels.DotNetRemotingLayer;

IGenuineChannelEventProvider iGenuineChannelEventProvider =
    ChannelServices.GetChannel("BroadcastSender1") as IGenuineChannelEventProvider;
iGenuineChannelEventProvider.GenuineChannelsEvent += new
    GenuineChannelsGlobalEventHandler(this.GenuineChannelsGlobalEventHandler);
```

[VB.NET]

```
Imports Belikov.GenuineChannels.DotNetRemotingLayer

Dim iGenuineChannelEventProvider As IGenuineChannelEventProvider = _
    DirectCast(ChannelServices.GetChannel("BroadcastSender1"), IGenuineChannelEventProvider)
AddHandler iGenuineChannelEventProvider.GenuineChannelsEvent, _
    New GenuineChannelsGlobalEventHandler(AddressOf Me.GenuineChannelsGlobalEventHandler)
```

Belikov.GenuineChannels.DotNetRemotingLayer.GenuineGlobalEventProvider provides the global *GenuineChannelsGlobalEvent* event fired for each Genuine Channels event. You can attach a handler to it if you want to process events generated by all channels and services.

[C#]

```
using Belikov.GenuineChannels;
using Belikov.GenuineChannels.DotNetRemotingLayer;

GenuineGlobalEventProvider.GenuineChannelsGlobalEvent +=
    new GenuineChannelsGlobalEventHandler(GenuineChannelsEventHandler);

public static void GenuineChannelsEventHandler(object sender, GenuineEventArgs e)
{
    Console.WriteLine("Global event: {0}, Url: {1}, Exception: {2}.",
        e.EventType,
        e.HostInformation == null ? "<null>" : e.HostInformation.ToString(),
        e.SourceException == null ? "<null>" : e.SourceException.Message);

    if (e.AdditionalInfo is ConnectionAcceptedCancelableEventParameter)
    {
        ConnectionAcceptedCancelableEventParameter connectionAcceptedCancelableEventParameter =
            (ConnectionAcceptedCancelableEventParameter) e.AdditionalInfo;
        Console.WriteLine("The connection is being accepted. Remote network entry: {0}.",
            connectionAcceptedCancelableEventParameter.IPEndPoint.ToString());
    }
}
```

[VB.NET]

```
Imports Belikov.GenuineChannels
Imports Belikov.GenuineChannels.DotNetRemotingLayer

AddHandler GenuineGlobalEventProvider.GenuineChannelsGlobalEvent, _
    New GenuineChannelsGlobalEventHandler (AddressOf GenuineChannelsEventHandler)

Public Shared Sub GenuineChannelsEventHandler(ByVal sender As Object, ByVal e As GenuineEventArgs)
    Dim hostInfoString As String
    If e.HostInformation Is Nothing Then
        hostInfoString = "<null>"
    Else
        hostInfoString = e.HostInformation.ToString()
    End If

    Dim sourceException As String
    If e.SourceException Is Nothing Then
        sourceException = "<null>"
    End If
End Sub
```

```

Else
    hostInfoString = e.SourceException.Message
End If

Console.WriteLine("Global event: {0}, Url: {1}, Exception: {2}.", e.EventType, hostInfoString, sourceException)

If e.AdditionalInfo Is ConnectionAcceptedCancellableEventParameter Then
    Dim connectionAcceptedCancellableEventParameter As ConnectionAcceptedCancellableEventParameter = _
        DirectCast ((e.AdditionalInfo, ConnectionAcceptedCancellableEventParameter)
    Console.WriteLine("The connection is being accepted. Remote network entry: {0}.", _
        connectionAcceptedCancellableEventParameter.IPEndPoint.ToString())
End If
End Sub

```

7. Queuing

The message queue is a queue of all messages that are to be sent through a specific transport connection. The message queue is necessary only for messages being sent and only if the connection is currently unavailable for sending. You can apply constraints to the maximum size of messages being sent, to the total size of all messages kept in the queue and to the total number of messages kept in the queue.

Why do you possibly need it? If a connection is slow or is not available for a long time, the queue starts to grow up. As soon as the queue parameters exceed the specified constraints, the connection is broken, the memory allocated for messages is released, and all callers receive exceptions that the message queue associated with the connection to a certain remote host is overrun.

As a result, if the server is overloaded, it will start dropping clients and continue its work. If some client has established a very slow connection, it starts reconnecting to the server again and again, until it establishes a connection with a reasonable speed.

Concrete constraint values are highly dependant on your task. If your client is expected to send only small messages, you can set the `MaxQueuedItems` parameter to a large value and the `MaxTotalSize` to a small value.

The only purpose of these settings is to provide conditions for stable and predictable work. The correct values will help you create a predictable software application.

8. Logging

By logging we mean putting down debugging records simplifying the analysis of occurred situations and the looking for errors. Genuine Channels use the binary log format to minimize both the resource consumption spent on writing logs and the total size of the log file.

To enable logging into a file, it is necessary to invoke the *Belikov.GenuineChannels.Logbook.GenuineLoggingServices.SetUpLoggingToFile* method, specify the name of the file and set the required options.

[C#]

```
// enable logging to a file
Belikov.GenuineChannels.Logbook.GenuineLoggingServices.SetUpLoggingToFile(filename, null);
```

[VB.NET]

```
" enable logging to a file
Belikov.GenuineChannels.Logbook.GenuineLoggingServices.SetUpLoggingToFile(filename, Nothing)
```

The file logger will automatically add the current date and the *.genchlog* extension to the file name. If you specify *ServerLog* as a file name, you will get *ServerLog.2004-10-04.genchlog*, *ServerLog.2004-10-05.genchlog* and so on.

Each log consists of log records that can be viewed with the help of the Genuine Log Viewer. Each record belongs to a particular event category and event subcategory. You can specify the level of the records that are written into the log. The zero level means that no records will be written. The first level allows records without binary content. The second level allows binary content to be included into the log. The third level makes it possible to include records containing personal or security information.

Log Category	Letter and Default Level	Description
Connection	C1	This category contains connection notifications.
ChannelEvent	E1	This category contains notifications describing what Channel Events are raised.
MessageProcessing	M2	This category contains notifications related to message processing.
ImplementationWarning	I3	This category contains notifications generated by Genuine Channels in unexpected or unwanted situations.
Security	S1	This category contains security notifications.
BroadcastEngine	B2	This category contains notifications describing Broadcast Engine invocations.
Transport	T1	This category contains general transport notifications.
DXM	X1	This category contains notifications generated by Direct Exchange Manager.
HostInformation	H1	This category contains notifications related to HostInformation objects serving Security Sessions, Client Sessions and network information with regard to remote hosts.

AcceptingConnection	A1	This category contains notifications informing you about accepting inbound connections and what local end points are listened to.
Debugging	D2	This category contains notifications generated by user code.
Version	V1	This category contains notifications revealing the environment information such as the Genuine Channels version, the CLR version, the version of OS and so on.
LowLevelTransport	L0	This category contains notifications generated by the low-level transport. These records simplify Genuine Channels debugging. By default, these records are not written.
StatisticCounters	N1	This category contains notifications generated by the Genuine Thread Pool and the Buffer Pool. These messages contain information about resource consumption.

You can use a string to specify category levels in the configuration file or programmatically:

```
[C#]
// Use default category levels
SetUpLoggingOptionsFromString(binaryLogWriter, GenuineLoggingServices.DefaultLoggingOptions);

// Use category levels by default as well
SetUpLoggingOptionsFromString(binaryLogWriter, null);

// If you want to enable low-level transport records, you can use either the full format
GenuineLoggingServices.SetUpLoggingToFile(name, "C1E1M2I3S1B2T1X1H1A1D2V1L2N1");

// Or the short format
GenuineLoggingServices.SetUpLoggingToFile(name, "L2");

[VB.NET]
' Use default category levels
SetUpLoggingOptionsFromString(binaryLogWriter, GenuineLoggingServices.DefaultLoggingOptions)

' Use category levels by default as well
SetUpLoggingOptionsFromString(binaryLogWriter, Nothing)

' If you want to enable low-level transport records, you can use either the full format
GenuineLoggingServices.SetUpLoggingToFile(name, "C1E1M2I3S1B2T1X1H1A1D2V1L2N1")

' Or the short format
GenuineLoggingServices.SetUpLoggingToFile(name, "L2")
```

Please, refer to the Debugging Guide for the detailed description of GenuineLogViewer and log records.

9. Client Session

There are two ways of implementing a Client Session. You can use the remote host URI to manage your own approach or just use the client session provided by *HostInformation*. It is not recommended to use Client Sessions with the **GUdp** channel and no Client Session is available for **one-way** messages; in all other cases you can call *GenuineUtility.CurrentSession* to get it.

1. Client Sessions are available during invocations and Genuine Channels events.
2. You can always get the Client Session associated with the host where a particular MBR object resides. You can use *GenuineUtility.FetchHostInformationFromMbr* for this.
3. Client Sessions are implemented by the *HostInformation* class. In other words, *HostInformation* is a Client Session and you can use any *HostInformation* as a Client Session.
4. Client Sessions have thread-safe implementation.
5. You will not be able to enumerate all keys or values. Only getting and setting are allowed.

Client Sessions are available while performing invocations, if the invocation is not one-way:

```
[C#]
public IChatRoom EnterToChatRoom(string nickname)
{
    GlobalRoom.AttachClient(nickname);
    GenuineUtility.CurrentSession["Nickname"] = nickname;
    return GlobalRoom;
}

[VB.NET]
Public Function EnterToChatRoom(ByVal nickname As String) As IChatRoom
    GlobalRoom.AttachClient(nickname)
    GenuineUtility.CurrentSession("Nickname") = nickname
    Return GlobalRoom
End Function
```

Use *e.HostInformation* to get the Client Session during Genuine Channels events:

```
[C#]
public static void GenuineChannelsEventHandler(object sender,
    GenuineEventArgs e)
{
    Console.WriteLine("Global event: {0}, Url: {1}, Exception: {2}", e.EventType, e.HostInformation.ToString(), e.SourceException);

    if (e.EventType == GenuineEventType.GeneralConnectionClosed)
    {
        string nickname = e.HostInformation["Nickname"] as string;
        if (nickname != null)
            Console.WriteLine("Client \"{0}\" has been disconnected.", nickname);
    }
}

[VB.NET]
Public Shared Sub GenuineChannelsEventHandler(ByVal sender As Object, ByVal e As GenuineEventArgs)

    Console.WriteLine("Global event: {0}, Url: {1}, Exception: {2}", e.EventType, e.HostInformation.ToString(), e.SourceException)

    If e.EventType = GenuineEventType.GeneralConnectionClosed Then
        Dim nickname As String = DirectCast(e.HostInformation("Nickname"), String)
        If Not IsNothing(nickname) Then
            Console.WriteLine("Client ""{0}"" has been disconnected.", nickname)
        End If
    End If
```


| End Sub

Client Sessions are released automatically as soon as the remote host is considered to be disconnected. Therefore, if you do not hold any references to a specific Client Session, it becomes a subject for Garbage Collecting.

Alternatively, you can use the client URI to mark data structures related to this specific client. Theoretically, this approach gives you more freedom (with additional programming) and allows you to use weak references.

Please note that the ***GenuineUtility.FetchChannelUriFromMbr*** method provides you access to all resources of a particular remote host via an arbitrary MBR object. For example, you can always access its Client Session if you have an MBR object obtained from it.

[C#]

```
using Belikov.GenuineChannels;
using Belikov.GenuineChannels.Security;
using Belikov.GenuineChannels.Connection;
using Belikov.GenuineChannels.TransportContext;

string uri;
ITransportContext iTransportContext;
GenuineUtility.FetchChannelUriFromMbr( anyMBRObject, out uri, out iTransportContext);
string nickname = iTransportContext.KnownHosts.Get(uri)["Nickname"] as string;
```

[VB.NET]

```
Imports Belikov.GenuineChannels
Imports Belikov.GenuineChannels.Security
Imports Belikov.GenuineChannels.Connection
Imports Belikov.GenuineChannels.TransportContext

Dim uri As String
Dim iTransportContext As ITransportContext
GenuineUtility.FetchChannelUriFromMbr(anyMBRObject, uri, iTransportContext)
Dim nickname As String = DirectCast(iTransportContext.KnownHosts.Get(uri)("Nickname"), String)
```

10. Thread Pooling Strategies

Genuine Channels use a customized Thread Pool internally (hereinafter, Genuine Thread Pool.) All operations requiring something to be performed in a separate thread use Genuine Thread Pool. Genuine Thread Pool can use different strategies to manage working threads.

Strategy	Behavior for usual work items	Behavior for long-duration work items
AlwaysNative	Always redirects requests to the native Thread Pool.	Always redirects requests to the native Thread Pool.
OnlyLongDuration	Always redirects requests to the native Thread Pool.	Runs them in the Genuine Thread Pool's working threads.
SwitchAfterExhaustion	Redirects requests to the native Thread Pool as long as it has at least <i>ThreadPoolAvailableThreads</i> working threads available.	Runs requests in the Genuine Thread Pool's working threads.
AlwaysThreads	Runs requests in the Genuine Thread Pool's working threads.	Runs requests in the Genuine Thread Pool's working threads.

IMPORTANT! During our stress tests, we found out that the native Thread Pool does not provide the correct number of available working threads when it is heavily loaded. Therefore, the *SwitchAfterExhaustion* strategy may fail to work as expected.

You can customize the Thread Pooling Strategy with the help of the *Belikov.GenuineChannels.Utilities.GenuineThreadPool* class. All properties are thread-safe.

Genuine Thread Pool Properties		
Name	Type, default value, unit	Description
<i>ThreadPoolAvailableThreads</i>	Int, 12, items	Gets or sets a value indicating the lowest number of working threads available in the native Thread Pool if you use the <i>SwitchAfterExhaustion</i> strategy.
<i>MaximumThreads</i>	Int, 500, threads	Gets or sets a value indicating the maximum number of working threads run by the Genuine Thread Pool.
<i>CloseAfterInactivity</i>	TimeSpan, 4 minutes	Gets or sets the period of time after which the Genuine Thread Pool terminates idle working threads.
<i>UnsafeQueuing</i>	Bool, true	Gets or sets a value indicating whether the <i>UnsafeQueueUserWorkItem</i> method will be used.
<i>GenuineThreadPoolStrategy</i>	Enum, <i>AlwaysThreads</i>	Gets or sets a value specifying the current Thread Pooling Strategy.
<i>AllowRequestQueuing</i>	Bool, true	Represents a boolean value indicating whether requests may be queued when the number of the Genuine Thread Pool's threads is close to <i>GenuineThreadPool.MaximumThreads</i> .

<i>MaximumRequestsQueued</i>	Int, 500	Represents an integer value indicating the maximum allowed number of queued requests if the number of the Genuine Thread Pool's threads has exceeded <i>GenuineThreadPool.MaximumThreads</i> .
-------------------------------------	-------------	--

11. Channel Prefix

.NET Remoting recognizes the used transport with the help of URI prefixes. For example, if you invoke “**gtcp**://192.168.10.3/Remote.rem”, .NET Remoting asks every registered channel whether it can provide access to this remote host. The result channel is chosen according to its priority value. If you have several channels supporting the same protocol, only one of them with the highest priority will always be used.

All Genuine Channels support the “prefix” parameter that specifies what URI prefix this channel serves.

```
<configuration>
<system.runtime.remoting>
<application>

<channels>
<channel
  type="Belikov.GenuineChannels.GenuineTcp.GenuineTcpChannel,GenuineChannels"
  prefix = "gcustom"
/>
</channels>

<client>
<wellknown
  type="KnownObjects.ChatRoom, KnownObjects"
  url="gcustom://127.0.0.1:8737/ChatRoom.rem"
/>
</client>
```

Hence, you can have several channels working via the same protocol with a different set of sinks (and different Transport Contexts). You specify directly which one will be used for every request. If you get an MBR object from the remote host, all invocations of this object will be sent through the channel the MBR object was obtained through.

If you are going to specify a custom channel prefix, please consider that

1. The prefix must start with the symbol ‘g’.
2. The length of the prefix may not exceed 8 symbols.
3. The length of the prefix may not be shorter than 4 symbols.

12. Compression

The Genuine Channels solution uses SharpZipLib library (<http://www.icsharpcode.net/OpenSource/SharpZipLib>) to compress outgoing packets.

Since release 2.3, the use of compression is specified in Security Session parameters. Please, take a look at the Security Session section for further details.

Compressing ratio is highly dependent on the content being sent. Usually, the usage of compression reduces traffic by 5-10 times for large text and DataSet values. In general, English text in the ASCII format is usually compressed by a factor of 2.5 to 3. UNICODE text is usually compressed by a factor of 4 to 6. This significantly increases the ratio of compression.

13. Deployment

You need *GenuineChannels.dll* and *ICSharpCode.SharpZipLib.dll* to be either in the GAC or in the directory where your executable files are placed. You can use precompiled *GenuineChannels.dll* for this purpose. It is stored in the DLL subfolder of the installation folder.

You can get the latest version of *ICSharpCode.SharpZipLib.dll* at <http://www.icsharpcode.net/OpenSource/SharpZipLib/default.asp>.

14. Error Messages

All exceptions thrown by Genuine Channels are located in the *GenuineExceptions.cs* file.

This allows you to translate error messages into any languages.

All Genuine Channels exceptions are inherited from the *Belikov.GenuineChannels.OperationException* class. You can use the class name or exception identifiers to analyze exceptions programmatically.

For example, an invocation of the server's SAO can cause *OperationException* with the error identifier "*GenuineChannels.Exception.Send.DestinationIsUnreachable*". If you get this exception, you can display the corresponding GUI form to the end user and ask to enter the valid address of the server.

```
[C#]
using Belikov.GenuineChannels;
try
{
    // ...
}
catch(OperationException ex)
{
    if (ex.OperationErrorMessage.ErrorIdentifier.IndexOf ( "DestinationIsUnreachable") > 0)
    {
        // show a form to user and...
    }
}

[VB.NET]
Imports Belikov.GenuineChannels
Try
    " ...
Catch ex As OperationException
    If ex.OperationErrorMessage.ErrorIdentifier.IndexOf ( "DestinationIsUnreachable") > 0 Then
        " show a form to user and...
    End If
End Try
```

This is a complete list of all error identifiers.

Identifier	Meaning
<i>GenuineChannels.Exception.Send.TooLargePacketSize</i>	The size of the message exceeds the MaxContentSize value.
<i>GenuineChannels.Exception.Send.TransportProblem</i>	The remote peer closed the connection before the entire message (chunked, according to the GTCP settings) has been sent.
<i>GenuineChannels.Exception.Send.ServerDidNotReply</i>	The remote host has not replied to the sent message for Belikov. GenuineChannels. Messaging. MessageManager. TimeSpanWaitReplyTimeout time.
<i>GenuineChannels.Exception.Send.Timeout</i>	The message has not been sent within the specified timeout.

<i>GenuineChannels.Exception.Send.NoSender</i>	It usually occurs on the server when the server tries to send a message to the client that has already closed the connection and has not reconnected. In addition, it can occur on the client, if the client uses a server MBR object (not SAO or CAO), or the destination URL is invalid.
<i>GenuineChannels.Exception.Send.DestinationIsUnreachable</i>	Occurs when there is no way to send the message to the remote host. For example, if a connection cannot be established.
<i>GenuineChannels.Exception.Send.NoNamedConnectionFound</i>	The requested named connection has not been found.
<i>GenuineChannels.Exception.Send.QueueIsOverloaded</i>	The size of the queue has exceeded the MaxQueuedItems parameter or the total queue size has exceeded the MaxTotalSize parameter.
<i>GenuineChannels.Exception.CanNotConnectToRemoteHost</i>	The Socket.Connect method has thrown an exception. The text of the exception is included to the error message.
<i>GenuineChannels.Exception.Connect.CanNotResolveHostName</i>	Dns.Resolve cannot resolve the specified host name. The host name is included into the error message.
<i>GenuineChannels.Exception.NoPortToListen</i>	The server channel cannot identify port to be listened to. That is, the port parameter has no correct string value.
<i>GenuineChannels.Exception.Connect.RemoteHostDoesNotRespond</i>	Fired when the remote host does not reply to the ping request.
<i>GenuineChannels.Exception.CanNotAcceptIncomingConnection</i>	An exception has occurred during the Socket.Accept call or accepting the socket. The error message contains the source error message.
<i>GenuineChannels.Exception.Connect.ConnectionReestablished</i>	The remote host has opened another connection and enforced connection reestablishing. The currently established connection is shutting down.
<i>GenuineChannels.Exception.Connection.LifetimeCycleEnded</i>	The invocation or one-way connection has been shut down according to the Finite Automaton Transition Diagram.
<i>GenuineChannels.Exception.Receive.IncorrectHostUpdate</i>	It is impossible to update the client URI or the server URL after the connection is established.
<i>GenuineChannels.Exception.Receive.IncorrectData</i>	The incoming stream is incorrect. The remote host uses an incorrect channel or an unexpected Security Session. This exception usually occurs when the remote host closes the connection or an exception is thrown during deserialization.
<i>GenuineChannels.Exception.Receive.TooLargePortionToReceive</i>	The size of the received message exceeds the MaxContentSize parameter.
<i>GenuineChannels.Exception.Receive.ConnectionClosed</i>	The connection has been forcibly closed.
<i>GenuineChannels.Exception.Receive.TooLargePortionToReceive</i>	The client has sent a portion of data, which is too large. The server has declined the request because it requires a larger memory buffer than it is allowed.

<i>GenuineChannels.Exception.Receive. Portion</i>	Only part of the expected data has been received. Probably, the connection has been unexpectedly closed.
<i>GenuineChannels.Exception.Receive. NoServices</i>	There are no services associated with the requested service name.
<i>GenuineChannels.Exception.Receive. NoServerChannel</i>	There is no GenuineHttpServerChannel named “ghttp” to perform incoming requests. You must create the Genuine HTTP server channel in order to receive client requests.
<i>GenuineChannels.Exception.Receive. NewSessionDetected</i>	The transport layer has detected that the remote host’s HostInformation had changed its identifier. This means that the remote host opened or accepted this connection for the first time and it does not know anything about us. All established Security Sessions became invalid and it is necessary to resubscribe to all remote host’s events.
<i>GenuineChannels.Exception.Receive. ServerHasBeenRestarted</i>	The client channel has detected that the GUID of the remote host had been changed due to restart.
<i>GenuineChannels.Exception.Receive. ConflictOfConnections</i>	Occurs when two connections requires the same resource.
<i>GenuineChannels.Exception. Processing.LogicError</i>	You are not supposed to see this message at all! If you do, please attach the binary file logger, reproduce the problem and send the log file to our support service.
<i>GenuineChannels.Exception. Processing. NoSessionAvailable</i>	An unknown Security Session has been specified for the message processing.
<i>GenuineChannels.Exception.Processing. ThreadPoolLimitExceeded</i>	The limit of threads has been exceeded.
<i>GenuineChannels.Exception. Processing.TransportConnectionFailed</i>	Thrown for those messages that are waiting for replies, but the connection to the remote host is closed.
<i>GenuineChannels.Exception.Processing. HostResourcesReleased</i>	All resources associated with the remote host have been released.
<i>GenuineChannels.Exception. Server.IncorrectUrl</i>	The invalid port number to listen to.
<i>GenuineChannels.Exception.Server. EndPointListened</i>	The specified end point is already in use or it has not finished shutting down yet.
<i>GenuineChannels.Exception.Server. ClientDidNotReconnectWithinTimeOut</i>	The client application has not reconnected within the specified timeout. The information associated with that client is being released.
<i>GenuineChannels.Exception.Windows. CanNotCreateOrOpen.NamedEvent</i>	The Windows API has returned an error. Please, use the Visual C++ Error Lookup utility to see the text of the error message.
<i>GenuineChannels.Exception.Windows. NamedEventError</i>	The Windows API has returned an error. Please, use the Visual C++ Error Lookup utility to see the text of the error message.
<i>GenuineChannels.Exception.Windows. CanNotCreateOrOpen.SharedMemory</i>	The Windows API has returned an error. Please, use the Visual C++ Error Lookup utility to see the text of the error message.

<i>GenuineChannels.Exception.Windows.SharedMemoryError</i>	The Windows API has returned an error. Please, use the Visual C++ Error Lookup utility to see the text of the error message.
<i>GenuineChannels.Exception.Windows.SspiError</i>	The SSPI API has returned an error. Please, use the Visual C++ Error Lookup utility to see the text of the error message.
<i>GenuineChannels.Exception.Windows.SspiDidNotProvideRequestedFeature</i>	The SSPI package does not provide the requested feature. The name of the requested feature is included into the error text.
<i>GenuineChannels.Exception.Channel.InvalidParameter</i>	The specified value of the parameter is not expected. Please, refer to Genuine Channels documentation for more details.
<i>GenuineChannels.Exception.Channel.ReconnectionFailed</i>	The client channel has not been able to reconnect to the server for the specified time and number of attempts. The channel will try to connect to the server again only when you invoke the SAO or CAO object of the remote host.
<i>GenuineChannels.Exception.Channel.ConnectionGracefullyClosed</i>	The connection has been closed according to a mutual agreement.
<i>GenuineChannels.Exception.Channel.ConnectionClosedAfterTimeout</i>	No messages have been received from the remote host for the specified time period. The connection is considered to be needless or broken.
<i>GenuineChannels.Exception.Channel.Closed</i>	All connections have been closed because the StopListening method have been called.
<i>GenuineChannels.Exception.Channel.Desynchronization</i>	The incoming packet does not contain the correct packet identifier. It means that the connection is out of order and must be closed.
<i>GenuineChannels.Exception.Broadcast.DestinationTypeMustBeAnInterface</i>	You must specify an interface while constructing a broadcast dispatcher.
<i>GenuineChannels.Exception.Broadcast.ClientSinkIsUnknown</i>	You must use only Genuine Channels if you use the Broadcast Engine.
<i>GenuineChannels.Exception.Broadcast.RemoteEndPointDidNotReplyForTimeOut</i>	The remote host has not replied to the message for the specified time.
<i>GenuineChannels.Exception.Broadcast.CallHasAlreadyBeenMade</i>	The Broadcast Engine has detected an already processed invocation.
<i>GenuineChannels.Exception.Broadcast.HandlerUninitialized</i>	It is impossible to activate the asynchronous mode while the Dispatcher.BroadcastCallFinishedHandler property is not initialized.
<i>GenuineChannels.Exception.Security.ContextNotFound</i>	An unknown Security Session is requested for message processing.
<i>GenuineChannels.Exception.Security.UnexpectedContext</i>	The Security Session can not be established due to an undetermined reason. Please, check the log files to reveal the source of the problem. The name of the Security Context is included into the message text.
<i>GenuineChannels.Exception.Security.InvalidTimeoutParameter</i>	The timeout parameter of the Security Session is invalid. It should look like: "Timeout=240000".

<i>GenuineChannels.Exception.Security.PasswordKnowledgeIsNotProved</i>	The remote host has not proved the knowledge of the password and will be disconnected.
<i>GenuineChannels.Exception.Security.WrongSignature</i>	The message integrity checking failure: the message has a wrong signature.
<i>GenuineChannels.Exception.Channel.TrialConditionExceeded</i>	The trial version limitation has been reached.

In order to localize error messages, you need to provide an object supporting the `Belikov.GenuineChannels.IGenuineExceptionLocalizer` interface. For example,

[C#]

```
class CustomErrorMessages : Belikov.GenuineChannels.IGenuineExceptionLocalizer
{
    public string Localize(string errorId, params object[] messageParameters)
    {
        switch (errorId)
        {
            case "GenuineChannels.Exception.Send.TooLargePacketSize":
                return string.Format("This is a customized text of the error message. Parameters: {0}-{1}.",
                    messageParameters);
        }

        return null;
    }
}
```

[VB.NET]

```
Class CustomErrorMessages
    Inherits Belikov.GenuineChannels.IGenuineExceptionLocalizer

    Public Function Localize(ByVal errorId As String, ByRef messageParameters() As Object) As String
        Select errorId
            Case "GenuineChannels.Exception.Send.TooLargePacketSize"
                Return String.Format("This is a customized text of the error message. _
                    Parameters: {0}-{1}.", messageParameters)
        End Select

        Return Nothing
    End Function
End Class
```

After that, you need to initialize the `GenuineExceptions.IGenuineExceptionLocalizer` member with your object.

[C#]

```
GenuineExceptions.IGenuineExceptionLocalizer = new CustomErrorMessages();
```

[VB.NET]

```
GenuineExceptions.IGenuineExceptionLocalizer = New CustomErrorMessages()
```

You do not need to localize all error messages. If you return a null reference, the default error message is used.

15. Chat Sample

The Genuine Channels installation package includes a simple chat demo. The server accepts connections opened by client applications and allows to send and receive text messages. Client applications connect to the server, register themselves as message recipients and send and receive messages.

In order to start it, please, open the Server Project Folder and click the Server.sln file. The solution is configured to start both the client and the server concurrently, therefore you just need to press F5 to start both applications. Use the command line or Windows Explorer to run as many client applications concurrently as you wish. If you want to use a client from another machine, please, change the remote URL in the *Client.config.exe* file:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime
        leaseTime="10M"
        sponsorshipTimeout="10M"
        renewOnCallTime="10M"
        leaseManagerPollTime="1M"
      />

      <channels>
        <channel type="Belikov.GenuineChannels.GenuineTcp.GenuineTcpChannel, GenuineChannels"
        />
      </channels>
    </application>
  </system.runtime.remoting>
  <appSettings>
    <add
      key="RemoteHostUri"
      value="gtcp://127.0.0.1:8737"
    />
  </appSettings>
</configuration>
```

By default, clients connect to the same machine (127.0.0.1) they are running on.

It is worth mentioning that the chat client applications subscribe to channel events and correctly process situations when the server is restarted:

```
[C#]
// main
GenuineGlobalEventProvider.GenuineChannelsGlobalEvent += new
GenuineChannelsGlobalEventHandler(GenuineChannelsEventHandler);

// handler code
if (e.EventType == GenuineEventType.GeneralServerRestartDetected)
{
    // server has been restarted so we have to register our listener again
    ChatRoom chatRoom = new ChatRoom();
    chatRoom.AttachClient(new ChatClient());
}

[VB.NET]
" main
HewHandler GenuineGlobalEventProvider.GenuineChannelsGlobalEvent, _
    New GenuineChannelsGlobalEventHandler(AddressOf GenuineChannelsEventHandler)

" handler code
If e.EventType = GenuineEventType.GeneralServerRestartDetected Then
    " server has been restarted so we have to register our listener again
    Dim chatRoom As ChatRoom = New ChatRoom()
    chatRoom.AttachClient(New ChatClient())
End If
```

The server uses the Broadcast Engine to send messages to the client applications. Therefore, it does not care about sponsorship and failed recipients.