



Why Genuine Channels?



Genrix

Genuine solutions for
successful development

Content

1. Accessibility for clients behind NAT, Firewall and proxy	3
2. Non-blocking calls	3
3. Guaranteed call timeout	3
4. Genuine Channels provides a stable connection	4
5. Clients detect whether the server was restarted.....	4
6. Easy server configuration.....	5
7. Queuing	5
8. Compression.....	5
9. Encryption	6
10. SSPI support	7
11. Thread execution.....	8
12. Broadcast Engine	8
13. IP multicast.....	9
14. GHTTP channels send several messages at one packet.....	10
15. Rendering transport characteristics and events at GUI level	10
16. Framework 1.1 support	11
17. Applet support	11
18. Detailed logging and debugging.....	11
19. Genuine Channels is distributed with sources.....	12
20. Licensing	12
21. Intensive development	13
22. Viable control over messages being sent	13
23. Good support!.....	13
Appendix A	15
Accessibility for clients behind NAT, Firewall and proxy	15
TCP Sample	15
GTCP Sample	15
GHTTP Sample	16
Appendix B	17
Non-blocking calls	17
TCP Sample	17
GTCP Sample	17
Appendix C	18
Guaranteed call timeout	18
TCP Sample.....	18
GTCP Sample	18
Appendix D	20
Broadcast Engine	20
GTCP Sample	20
Appendix E.....	22
IP multicast.....	22
GTCP Sample	22
Appendix F.....	23
Security Session DEMO	23
GTCP Sample	23

1. Accessibility for clients behind NAT, Firewall and proxy

Native TCP and HTTP channels use one TCP or HTTP connection for ferrying calls only in one direction. If you want to invoke client's object from the server, server must connect to the client's port. That is if your server needs to inform several clients or the specific client about an event generated by the server itself or by any of connected clients, then a TCP or HTTP connection is opened from the server to the client (to every client). Thereby a client has to listen to the specific TCP port and open additional connections in order to receive calls from the server. This scheme spawns firewall, NAT and proxy-related problems because such connections usually are denied by default.

Genuine TCP (GTCP) channel and Genuine HTTP (GHTTP) channel use connections opened by a client for invoking clients' objects. No any additional connections need to be opened.

Genuine Channels do not have any problems with firewalls, NAT and proxy servers. You will be able to receive server's events even if your client is thousands of miles apart and gets access through a proxy requiring authorization.

See the sample description in the Appendix A.

2. Non-blocking calls

Consider the following scenario. A client makes three similar calls to the server concurrently in different threads.

Native channels in framework 1.0 will perform only one call at a time, so all invocations will be performed consequently. Framework 1.0 SP2 or 1.1 will open three TCP or HTTP connections. In spite of the fact that three established connections idle away, each additional request will cause opening an additional TCP or HTTP connection. Opening a new connection consumes network resources and takes some time (one additional round-trip).

GTCP channel will send all calls via once opened TCP or HTTP connection. No additional connections will be opened, doesn't matter how many and what calls you need to make.

Genuine Channels always uses established connections effectively. Genuine Channels do not block calls like framework 1.0 does.

See the sample description in the Appendix B.

3. Guaranteed call timeout

Let's assume you use native TCP channel and make a synchronous call. Remote host hangs and the caller will wait for the response endlessly, until TCP connection will be closed. It looks like hanging of the remote host causes hanging of the caller.

You can make only asynchronous calls in critical places. In this case you will have to write 5 additional lines of code and analyze a result in separate method invoked asynchronously in a separate thread. So simple consequent processing will look like it was done by kind Dr. Frankenstein.

GTCP channel throws an exception if a response did not arrive for the specified time span, does not matter whether it is synchronous or asynchronous call. You just catch the exception where it is reasonable and continue processing. A problem on the client side will not make you server wasting its time for nothing.

With Genuine Channels an invocation time is always predictable and customizable. Either it finishes in time or an exception is thrown after a timeout.

You can assign specific timeout values for the specific calls.

With native TCP channel you probably will find your server hanging.

See the sample description in the Appendix C.

4. Genuine Channels provides a stable connection

Genuine Client channel is responsible for establishing a connection to the server. It is the main rule for all Genuine Channels. Genuine Server channel does not open any connections to the client, so we avoid NAT, firewall and proxy-related problems.

If a connection is broken by a network reason, then client channel automatically re-establishes the connection according to channel settings. Server recognizes connection re-establishing, no messages are lost. So you can unplug network cable and then plug it again. GTCP client will re-establish the TCP connection and your client-server solution will continue working.

GTCP, GHTTP and Shared Memory channels automatically send a ping after the specified time of network inactivity to check TCP, HTTP or Shared Memory connection.

You should not bother whether something will break TCP connection and your clients will stop receiving server's events.

Genuine Channels either say you firmly that the connection was broken and can not be re-established, or merely re-establish a connection and continue working.

5. Clients detect whether the server was restarted

GTCP and GHTTP channels recognize and let you to know whether the server has been restarted. If your client receives any events from the server, it can re-register its listener and continue receiving events.

Take a look at the GTCP sample. When you close server's window, client recognizes that TCP connection was broken and tries to re-establish TCP connection according to default GTCP client channel settings. After starting the server, client re-establishes a connection and recognizes that the server was restarted. It will throw "server was restarted" exception on all active calls and fire corresponding GTCP client channel event. Every client in my example just re-registers its listener and continues receiving chat messages.

With Genuine Channels you will not be afraid of skipping important server events.

See the sample description in the Appendix F.

6. Easy server configuration

MSDN says

However, when the computer name does not resolve with reasonable speed (if at all) and when the computer has more than one NIC, either physical or virtual (this is often the case with a dial-up connection or VPN network adapter), you should set the machineName property to the IP address of the NIC that is currently in use for that connection.

You have to setup correct server settings in order to let it assign an entry point for CAO or obtained MBR objects with native channels. Sometimes it is not a trivial task especially if your server machine has several NICs and clients should have access from several of them.

Genuine Channels do not have any of such problems. It is enough for clients to know server's address to connect to the server, that's all.

Genuine Channels are much more configurable and provide better conditions for hosting at ASP. Sometimes it is a key factor.

7. Queuing

Every time you make a call, a message is put into the message queue. Each connection has its own message queue containing messages (invocations and replies) to be sent to the remote host. Accordingly message queue takes into consideration the number of messages and summary size of all messages in bytes. These values are calculated for each connection and are increased when a message is required to be sent to the remote host and decreased when a message has been successfully sent. So they sway between zero and appropriate upper bound for every connection. Genuine Channels apply customizable constraints on these values. As soon as at least one queue value exceeds set limit, connection is broken, all calls are released, and all invocations result in the appropriate exception.

Heavily loaded server causes connection queues to grow up. Then it will disconnect several clients and continue its processing successfully. It is usually better to drop several clients out than increase load problems at the server.

If a client's connection is too slow, queue constraints will make a client to reconnect. If a client stops responding to the servers' requests without closing HTTP or TCP connection (usually due to an implementation bug), queue constraints will disconnect such client.

Queue constraints guarantee that a network exchange produced by your client-server solution is always within appropriate reasonable bounds and does not consume too much memory and network resources.

Genuine Channels' queue constraints allow you to create stable and predictable client-server solutions.

See the sample description in the Appendix F.

8. Compression

We use SharpZipLib library (<http://www.icsharpcode.net/OpenSource/SharpZipLib>) to compress messages.

Usually GZIP-based algorithm deflates messages containing text and DataSet for 5-10 times. The use of compression is determined by Security Session's parameter. So you can force compression for the specific invocations as well as for all invocations in the specific thread or channel. Compression will be performed before any content encryption will be performed, it doesn't matter what kind of content encryption you use.

Receiving host automatically recognizes whether a message has been deflated and inflates it when necessary.

Compress your content with Genuine Channels. You do not need to install any sinks. You can enable compression for the specific calls.

See the sample description in the Appendix F.

9. Encryption

Want to hide and protect your content being transferred? Genuine Channels provide wide possibilities to secure your calls. Such things as encryption, compression, threading, impersonation, delegation and so on are managed by Security Sessions. You can create any number of Security Sessions. There are five different contexts with different priorities the Security Session can be specified in: global, channel, connection, thread and local place.

Genuine Channels provide several types of Security Sessions with different characteristics. You can use known keys for the encryption as well as you can let Genuine Channels to generate unique key for each connection. You can force the specific Security Session only for the specific calls.

For example, it will take you exactly 3 lines at client and 1 at server to make Genuine Channels generate an asymmetric key, send a public key, generate a symmetric key, exchange it using the generated asymmetric key and then use the symmetric key for the encryption in the specific context.

Client side

#1 Create an instance of the Key Provider that will spawn Security Session for each connection:

```
GlobalKeyStore.SetKey("/Test/SES",  
                      new KeyProvider_SelfEstablishingSymmetric ());
```

#2 Fetch an interface to set up this Security Session in the channel context:

```
ISetSecurityContext iSetSecurityContext =  
    ChannelServices.GetChannel("gtcp")as ISetSecurityContext;
```

#3 Set the Security Session in the channel context, so all invocations made via this channel will be encrypted:

```
iSetSecurityContext.SecurityContext = "/Test/SES";
```


Server side

#1 Create an instance of the Key Provider that will spawn Security Session for each connection:

```
GlobalKeyStore.SetKey("/Test/SES",  
                      new KeyProvider_SelfEstablishingSymmetric());
```

Secure your data with Genuine Channels!

See the sample description in the Appendix F.

10. SSPI support

MSDN says

If you need to secure your calls, you must use an HTTP-based application hosted in IIS, whether that is an ASP.NET application or a remoting application. This is because ASP.NET and .NET remoting use the security services provided by IIS. .NET remoting does not provide any security services when hosted outside IIS (for example, in a Windows Service).

It does not matter for Genuine Channels whether your application hosted in IIS or not. You will be able to create several Security Sessions and leverage SSPI impersonation and delegation when you need it.

It will take you one line at client and server to create SSPI Security Session. Then you can use it when you need it:

Client side

#1 Create an instance of the Key Provider that will spawn Security Session for each connection:

```
// create SSPI Security Session with process security context  
GlobalKeyStore.SetKey("/Test/SSPI_ProcessCredential",  
    new KeyProvider_SspiClient(  
        SspiFeatureFlags.Impersonation,  
        SupportedSspiPackages.NTLM,  
        null, string.Empty));  
  
// create SSPI Security Session with known credential  
GlobalKeyStore.SetKey("/Test/SSPI_KnownCredential",  
    new KeyProvider_SspiClient(  
        SspiFeatureFlags.Impersonation | SspiFeatureFlags.Encryption,  
        SupportedSspiPackages.NTLM,  
        new NetworkCredential(@"UserName", "password", "domainName"),  
        string.Empty));
```

#2 Make the server to create a file in impersonated security context

```
// the call will be performed in the impersonated security context  
using(new SecurityContextKeeper("/Test/SSPI_ProcessCredential"))  
    serverBusinessObject.CreateFile(@"c:\tmp\file1.txt");  
  
// the call will be performed in the impersonated security context  
using(new SecurityContextKeeper("/Test/SSPI_KnownCredential"))  
    serverBusinessObject.CreateFile(@"c:\tmp\file2.txt");
```

Server side

```
// allows clients to establish NTLM Security Sessions and impersonates
// executed calls in these contexts
GlobalKeyStore.SetKey("/Test/SSPI_Process",
    new KeyProvider_SspiServer(
        SspiFeatureFlags.Impersonation,
        SupportedSspiPackages.NTLM ));

GlobalKeyStore.SetKey("/Test/SSPI_Process",
    new KeyProvider_SspiServer(
        SspiFeatureFlags.Impersonation | SspiFeatureFlags.Encryption,
        SupportedSspiPackages.NTLM ));
```

Genuine Channels provide attractive opportunity to leverage impersonation and delegation in corporate environment. Your users have their credentials to work in a corporate environment. Why do not utilize this in your applications seamlessly?

See the sample description in the Appendix F.

11. Thread execution

By default every time you make an invocation the performing thread is taken from the Thread Pool. Number of threads run in the Thread Pool is limited. It is possible to increase the upper limit, but it is not convenient, especially if you host your solution at ASP. Also how do you get to know how much threads exactly do you need?

MSDN says

The thread pool has a default limit of 25 threads per available processor, which could be changed using CorSetMaxThreads as defined in the mscoree.h file. Each thread uses the default stack size and runs at the default priority. Each process can have only one operating system thread pool.

If your server performs long-duration operations, then it is possible to meet problems with the lack of the threads in the Thread Pool and this leads to server failure.

Genuine Channels allow you to customize this behavior. You can either specify what calls will be performed in the dedicated thread or you can get rid of the Thread Pool usage and execute all invocations in separate threads.

Thread Pool issue is a problem. But not for you if you use Genuine Channels.

See the sample description in the Appendix F.

12. Broadcast Engine

.NET brings in such terms as delegates and events and event-driven programming model simplifies building distributed solutions. And this scheme is completely incompatible with .NET Remoting. Yes, it is a frustration {LINK TO THE ARTICLE}. But not fatal.

Genuine Channels bring in Broadcast Engine concept to eliminate such problems.

In order to perform event firing with Broadcast Engine you need to construct Dispatcher object, subscribe event receivers and then use transparent proxy object. Each time you call a method of the transparent proxy, the call is delivered to all event receivers. All invocations are made

concurrently, failed receivers do not affect others, recurrent calls are ignored by default and Broadcast Engine takes care about sponsorship automatically.

You can call your events either in asynchronous mode, provided handler will be called after an invocation in this case. In synchronous mode the control will be returned after an event invocation will be finished. Dispatcher automatically calculates number of consequent failures of each receiver and can remove receivers that did not reply for specific number of times consequently.

You can specify different properties for each dispatcher such as event timeout, call mode, remove conditions.

If a list of recipients is not constant and depends on the call, you can leverage Broadcast Engine Filtering functionality. In this case you can provide a filter that will filter out all recipients for each call. Filter is also set in an appropriate context (almost like it is done with Security Session). Genuine Channels provide means to recognize server restarting, so you will be able to re-subscribe to servers' events.

Genuine Channels contain all the necessary things to manage events in distributed applications.

See the sample description in the Appendix D.

13. IP multicast

MSDN says

IP multicast is an extension of IP that allows for efficient group communication. IP multicast arose out of the need for a lightweight, scalable conferencing solution that solved the problems associated with real-time traffic over a datagram, best-effort network. There are many advantages to using IP multicast: scalability, fault tolerance, robustness, and ease of setup.

The IP multicast conferencing model incorporates the following key features:

- 1) No global coordination is needed to add and remove members from a conference.
- 2) To reach a multicast group, a user sends data to a single multicast IP address. No knowledge of the other users in a group is necessary.
- 3) To receive data, users register their interest in a particular multicast IP address with a multicast-aware router. No knowledge of the other users in a group is necessary.
- 4) Routers hide the multicast implementation details from the user.
- 5) Traditional connection-oriented conferencing suffers from a number of problems:
 - a) *User complexity*: Users must know the location of every user they wish to converse with, limiting scalability and fault-tolerance and rendering it difficult for users to add and remove themselves from a conference.
 - b) *Wasted bandwidth*: A user wishing to broadcast data to n users must send data through n connections.

The total bandwidth required for multiparty conferences in which all users are sending data goes up as the square of the number of parties involved, leading to huge scalability problems. IP Multicast takes advantage of the actual network topology to eliminate the transmission of redundant data down the same communications links.

IP multicast implements a lightweight, session-based communications model, which places relatively little burden on conference users. Using IP multicast, users send only one copy of their information to a group IP address that reaches all recipients. IP multicast is designed to scale well as the number of participants expands—adding one more user does not add a corresponding amount of bandwidth. Multicasting also results in a greatly reduced load on the sending server.

IP multicast routes these one-to-many data streams efficiently by constructing a spanning tree, in which there is only one path from one router to any other. Copies of the stream are made only when paths diverge.

In addition to initial IP multicast benefits Genuine Channels provide you the easiest way of leveraging this functionality in a few lines of your code. You use Broadcast Engine just as you firing a usual event with Genuine Channels. Each time you invoke transparent proxy's method, all IP multicast clients receive the call.

Other important thing is that Genuine Channels guarantee you that even if client will not receive IGMP packet, Genuine Channels will re-send it via reliable stream (GTCP, GHTTP) after specified timeout automatically.

Please notice that Broadcast Engine automatically recognizes whether a client is able to receive events via IP multicast. So you create an event, enable IP multicast option for it, and then Broadcast Engine automatically recognizes clients receiving the event via IP multicast and sends the event via usual channels to clients that does not receive IGMP packets. If client received events via IP multicast and then it becomes unavailable for receiving IGMP packets, Broadcast Engine will automatically switch it for receiving the event via a usual channel.

If you have 100 clients in a LAN receiving the same event, then you can save 99% of server CPU time, server memory and server network resources by using IP multicast. With Genuine Channels you can have any number of clients receiving IP multicast packets on the same computer.

Reduce your server load with Genuine Channels greatly!

See the sample description in the Appendix E.

14. GHTTP channels send several messages at one packet

Genuine HTTP channels can send several invocations (according to specified settings) at one HTTP request when possible. This feature is useful if you need to make a lot of small invocations in several threads concurrently.

Genuine HTTP Channel can significantly decrease network traffic and raise the performance of your solution.

15. Rendering transport characteristics and events at GUI level

End users like live applications. Interactive applications that show network traffic counters, maybe in a graphic form, download progress and general network status. Genuine Channels provide all the necessary means for this.

As well as network counters, Genuine Channels provide channel's event firing for each important network or processing event. For example, connection established event, connection broken event, connection restored event, IP multicast message received event and so on.

Make your application alive with Genuine Channels!

See the sample description in the Appendix F.

16. Framework 1.1 support

Framework 1.1 brought in *TypeFilterLevel* parameter in the *BinaryServerFormatterSink* class and *configuration > system.runtime.remoting > customErrors* tag in the configuration file. It is not always convenient to analyze framework version and set correct values for these parameters. There is no easy way to set *customErrors* parameter programmatically. Genuine Channels set *TypeFilterLevel* to *Full* and *customErrors* to *off* by default in the framework 1.1.

Genuine Channels set up reasonable .NET Remoting parameter values by default according to the framework version.

17. Applet support

I have to admit that User Controls hosted on a web page will never become so popular like Java applets. At least until guys at Microsoft set up correct security options by default. But anyway it is possible to use Genuine Channels for communication between User Control hosted on a web page and your GHTTP-enabled server. Generally you can host absolutely random GUI application on a web page that gives you easy upgrade and distribution options.

It is possible to use Genuine Channels in a User Control hosted on a web page.

You can download a sample from www.genuinechannels.com

18. Detailed logging and debugging

Have you ever met the situation when native channels do not say exactly the reason of the problem? You spent a lot of your time, looked through and along MSDN and newsgroups. And the only thing you understand is that a mistake may be everywhere and there is no way to value the time it will take to be fixed.

MSDN says

Although there are only a few settings in the preceding configuration file, most of the problems using .NET remoting occur because some of these settings are either incorrect or do not match the configuration settings for client applications. It is easy to mistype a name, forget a port, or neglect an attribute. If you are having problems with your remoting application, check your configuration settings first.

Native channels do not say the reason of the problem. Even if a native channel throws an exception, there is no easy way to understand what is happening at the programming level. Generally you can analyze the text of the error message, but it is not a good approach. What you will do if they translate the framework to other languages? Genuine Channels provide accurate exception information as well as programming identifier for each type of exception. All the exception types are documented. The text of the exception can be modified, but the exception identifier will remain the same.

You have your server running for a long time and in a certain moment it just stops working? What will you do? How do you think you can find the reason quickly? Genuine Channels possess with a detailed logging which can be performed either into a file or into an attached object implementing specific interface. You always are able to understand what is going on. Genuine Channels compiled in DEBUG mode provide detailed log capabilities with the traffic being sent. Genuine Channels compiled in RELEASE mode output only exceptions and important events into the log.

Save your programming and debugging time with Genuine Channels!

See the sample description in the Appendix F.

19. Genuine Channels is distributed with sources

Let us assume that sometimes your solution fails due to an unknown reason. First of all you need to localize the error. Then you need to understand what is happening and why it is so. It would be great to go along the execution path in a debugger. And it would be great to be able to fix something, not to seek out for an endless numbers of different workarounds, right? Are you ready to study how .NET Remoting works in assembler or you prefer C#? Or you have a great intuition and can predict all the situations in 15,000 lines of code?

With Genuine Channels you can easily localize the problem and understand whether it is a Genuine Channels issue or not. You will be able to trace the execution and see what and why is happening. You will be able to fix or correct an inconvenient behavior on your own as well as ask me to do this.

With Genuine Channels you will be able to understand how your distributed solution works exactly. There will not be any blank spots and unpredictable situations.

20. Licensing

Genuine Channels product is distributed according to a developer-based politic. You need to buy as many licenses as many programmers will use it for the programming. We do not apply any restrictions to number of installations or end users. So you can develop client-server solution and then distribute and sell it without any overhead charges.

We want you just to use Genuine Channels, that is why we do not constraint number of installations or end users.

21. Intensive development

Having bought a license for 2.x branch, you will receive all future 2.x versions and fixes for free. We have good plans for 2.x branch.

Release 2.4 will be dedicated entirely to the performance. We will try to raise the performance and provide all the proper options for this.

Release 2.5 will be dedicated to clusters and farms. We will introduce MSMQ and MSMQ broadcast channels, NLB and cluster support, provide capabilities for building fault-tolerance systems and increasing number of servers at run-time.

Remember, we always listening to the needs of our customers. And generally the development of the Genuine Channels is directed by customers. We need to know your problems to improve our product!

Genuine Channels is a product with the future. Why do not design our future together?

22. Viable control over messages being sent

Let us consider it again. With Genuine Channels you will have a lot of different possibilities at a hand.

- Events and Broadcast Engine
- Built-in Compression
- Encryption
- SSPI support
- Queue Constraints
- Built-in Client Session
- Remote Peer URI and Network Address
- Customizable Call Timeout
- Execution in the Dedicated Thread
- Genuine Channel Events and traffic counters
- Logging
- and so on...

You can always ask me about anything else as well as use custom sinks provided by any third-party.

Being able to choose you will feel less under pressure. Genuine Channels grant you a possibility to choose.

23. Good support!

You've studied .NET Remoting, defined the most challengeable areas for your client-server solution and need to know whether Genuine Channels can help you? *Ask me!*

Or you merely have some questions about Genuine Channels and .NET Remoting? *Ask me!*
You need a feature and do not know how to implement it in a short and vivid manner? *Ask me!*

You think you've found a bug? *Let me know!*
You think we can improve Genuine Channels? *Let me know!*

You will feel yourself safer with our support. .NET Remoting is mostly undocumented technology and my experience accumulated while supporting my customers can certainly be useful for you and your client-server solutions.

Our customers develop commercially successful products with Genuine Channels.

And you?

Appendix A

Accessibility for clients behind NAT, Firewall and proxy

(File: "Accessibility for clients behind NAT, Firewall and proxy.zip")

TCP Sample

Installation

The sample was written in Visual Studio 2002.

You need to unpack provided archive and open *TCP 1.0 > Project.sln* solution file. Then rebuild the solution.

Press **F5** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the *Client > App.config* file. You should modify directly *Client > Bin > Debug > Client.exe.config* file if you need to change the IP address without recompiling the solution.

Event approach

I used the second approach from my article dedicated to events for implementing the event. You can find more information about it here:

<http://www.genuinechannels.com/Content.aspx?id=27&type=1>

GTCP Sample

Installation

The sample was written in Visual Studio 2002.

You need to unpack provided archive and open *GTCP 1.0 > Project.sln* solution file. Then rebuild the solution.

Press **F5** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the *Client > App.config* file. You should modify directly *Client > Bin > Debug > Client.exe.config* file if you need to change the IP address without recompiling the solution.

Design

Known layer contains only communication interfaces.

Server implements event provider interface and binds business object to known URI. Server uses Client Session to keep track of the specific client and its nickname.

Client implements event receiver interface and subscribes its listener at the server. Client builds local transparent proxy pointed to the known URI to access server's business object. Client's code uses several GTCP features such as determining server restarting and it shows Genuine Channels events. So if you close the server and then start it again, clients re-subscribe to the chat event and continue receiving chat messages.

You can find more information about this approach here:

<http://www.genuinechannels.com/Content.aspx?id=28&type=1>

You can uncomment line #59 in *Client > ChatClient.cs* file and line #34 in *ChatServer.cs* file to write a log into a file.

Event approach

I used Broadcast Engine for implementing the event and it resulted in

simpler source-code at the server side. You can find more information about Broadcast Engine in Programming Guide and in my article dedicated to events:

<http://www.genuinechannels.com/Content.aspx?id=27&type=1>

GHTTP Sample

Installation

The sample was written in Visual Studio 2002.

You need to unpack provided archive, open *Internet Information Services Console*, create virtual directory with **GHTTP 1.0 > GHTTPChatSample** alias and direct it to *GHTTPChatSample* directory. After that open *GHTTP 1.0 > Project.sln* solution file. Then rebuild the solution.

Press **F5** to run both client and server in Visual Studio.

Configuration

URL to the server is specified in the *Client > App.config* file. You should modify directly *Client > Bin > Debug > Client.exe.config* file if you need to change the IP address without recompiling the solution. Consider *Programming Guide > GHTTP Channels > Settings* section to specify proxy settings.

Design

GHTTP sample has absolutely the same design as GTCIP sample. *Default.aspx* page outputs all events and chat information. Periodically click **Refresh** in your browser to see the freshest content.

Known layer contains only communication interfaces.

Server implements event provider interface and binds business object to known URI.

Client implements event receiver interface and subscribes its listener at the server. Client builds local transparent proxy pointed to the known URI to access server's business object.

You can find more information about this approach here:

<http://www.genuinechannels.com/Content.aspx?id=28&type=1>

You can uncomment line #57 in *Client > ChatClient.cs* file and line #40 in *GHTTPChatSample > Global.asax.cs* file to write a log into a file.

Event approach

I used Broadcast Engine for implementing the event and it resulted in simpler source-code. You can find more information about Broadcast Engine in Programming Guide and in my article dedicated to events:

<http://www.genuinechannels.com/Content.aspx?id=27&type=1>

Appendix B

Non-blocking calls

(File: " Non-blocking calls.zip")

TCP Sample

Installation

The sample was written in Visual Studio 2002.
You need to unpack provided archive and open *TCP > Project.sln* solution file. Then rebuild the solution.
Press **F5** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the *Client > App.config* file. You should modify directly *Client > Bin > Debug > Client.exe.config* file if you need to change the IP address without recompiling the solution.

Design

Known layer contains the declaration of the operation interface.
Server provides the business object bound to the known URI. Server performs *Thread.Sleep* operation to simulate long-duration operation lasting for 20 seconds.
Clients start three threads. Each thread invokes the server's business object.

Issue

Use a timer to understand how much time it takes to complete all invocations.
Type in command line "*netstat -a*" and press **ENTER** to see a quantity of the opened connections to the port 8737.

GTCP Sample

Installation

The sample was written in Visual Studio 2002.
You need to unpack provided archive and open *GTCP 1.0 > Project.sln* solution file. Then rebuild the solution.
Press **F5** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the *Client > App.config* file. You should modify directly *Client > Bin > Debug > Client.exe.config* file if you need to change the IP address without recompiling the solution.

Design

Known layer contains only the declaration of the operation interface.
Server provides the business object bound to the known URI. Server performs *Thread.Sleep* operation to simulate long-duration operation lasting for 20 seconds.
Clients start three threads. Each thread invokes the server's business object.

Issue

Use a timer to understand how much time it takes to complete all invocations.
Type in command line "*netstat -a*" and press **ENTER** to see a quantity of the opened connections to the port 8737.

Appendix C

Guaranteed call timeout

(File: "Guaranteed call timeout.zip")

TCP Sample

Installation

The sample was written in Visual Studio 2002.
You need to unpack provided archive and open ***TCP > Project.sln*** solution file. Then rebuild the solution.
Press ***F5*** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the ***Client > App.config*** file. You should modify directly ***Client > Bin > Debug > Client.exe.config*** file if you need to change the IP address without recompiling the solution.

Design

Known layer contains the declaration of the operation interface.

Server provides the business object bound to the known URI. Server performs ***Thread.Sleep*** operation to simulate long-duration operation lasting for 24 hours.

Client invokes the server's business object and waits for 24 hours without any chance to interrupt this hanging, until the TCP connection will be broken.

Issue

If a client waits for a server for the uncontrolled time, it is not so dangerous. You can just close the client and run it again. But if the client holds on the call and, accordingly, holds on the server's thread, it can become a problem.

GTCP Sample

Installation

The sample was written in Visual Studio 2002.
You need to unpack provided archive and open ***GTCP 1.0 > Project.sln*** solution file. Then rebuild the solution.
Press ***F5*** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the ***Client > App.config*** file. You should modify directly ***Client > Bin > Debug > Client.exe.config*** file if you need to change the IP address without recompiling the solution.

Design

Known layer contains the declaration of the operation interface.

Server provides the business object bound to the known URI. Server performs ***Thread.Sleep*** operation to simulate long-duration operation lasting for 24 hours.

Client invokes the server's business object, waits for 120 seconds and

receives an exception that the remote side did not response for the specified timeout. After this, all call resources are released.

Issue

The caller side always receives an exception and all invocation resources are released after this. The invocation either finishes in time or an exception is thrown after a timeout. You can vary default timeout limit as well as you can specify the limit for specific calls or all calls in the appropriate context.

Appendix D

Broadcast Engine

(File: " Broadcast Engine.zip")

GTCP Sample

Installation

The sample was written in Visual Studio 2002.
You need to unpack provided archive and open **GTCP 1.0 > Project.sln** solution file. Then rebuild the solution.
Press **F5** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the **Client > App.config** file. You should modify directly **Client > Bin > Debug > Client.exe.config** file if you need to change the IP address without recompiling the solution.

Design

The sample implements a simple chat. Each client chooses its name after a starting, connects to the server, subscribes to the chat event and allows users to send messages to the server which, in its turn, re-sends messages to all other clients (except the sender).

Known layer contains the declaration of the chat event provider and chat message receivers. Chat room interface is declared as a separate interface; hence this sample can be easily modified for having several chat rooms.

Server provides the business object bound to the known URI. Server implements Chat and Chat Room functionality. Server sends a message to all clients except the message sender.

Client implements event receiver interface and subscribe its listener at the server. Client builds local transparent proxy pointed to the known URI to access server's business object. Client's code uses several GTCP features such as determining server restarting and it shows Genuine Channels events. So if you close the server and then start it again, clients re-subscribe to the chat event and continue receiving chat messages.

You can find more information about this approach here:
<http://www.genuinechannels.com/Content.aspx?id=28&type=1>

Issue

I used Broadcast Engine for implementing the event. You can find more information about Broadcast Engine in Programming Guide and in my article dedicated to events:

<http://www.genuinechannels.com/Content.aspx?id=27&type=1>

Take a look at the implementation of the *Server > UserFilter* class. It implements *IMulticastFilter* interface to filter out the message sender from the recipients list.

This filter is forced each time the event is fired by this call:

```
// force the filter to filter out the sender
using(new DispatcherFilterKeeper(
    new UserFilter(GenuineUtility.CurrentSession["UserId"] as
                                                           string)))
{
    IMessageReceiver iMessageReceiver =
        (IMessageReceiver)
this._dispatcher.TransparentProxy;
    iMessageReceiver.ReceiveMessage(message, nickname);
}
```

Appendix E

IP multicast

(File: "IP Multicast.zip")

GTCP Sample

Installation

The sample was written in Visual Studio 2002.

You need to unpack provided archive and open *GTCP 1.0 > Project.sln* solution file. Then rebuild the solution.

Press **F5** to run both client and server in Visual Studio.

Configuration

Server IP address is specified in the *Client > App.config* file. You should modify directly *Client > Bin > Debug > Client.exe.config* file if you need to change the IP address without recompiling the solution.

There is no need to modify broadcast addresses, default values should work in any LAN. You should change the port if the port 11000 is already in use in your LAN.

Design

The sample implements a simple chat. Each client chooses its name after a starting, connects to the server, subscribes to the chat event and allows users to send messages to the server which, in its turn, re-sends messages to all other clients (except the sender).

Known layer contains the declaration of the chat event provider and chat message receivers. Chat room interface is declared as a separate interface; hence this sample can be easily modified for having several chat rooms.

Server provides the business object bound to the known URI. Server implements Chat and Chat Room functionality. Server sends a message to all clients except the message sender. Server enables the usage of IP multicast by means of Broadcast Engine.

Client implements the event receiver interface and subscribes its listener to the event. Client builds local transparent proxy pointed to the known URI to access server's business object. Client uses several GTCP features such as determining server restarting and shows Genuine Channels events. So if you close the server and then start it again, clients re-subscribe to the chat event and continue receiving chat messages. Also client binds its receiver to the chosen court for receiving chat messages via IP multicast.

Issue

I used Broadcast Engine for implementing the event. You can find more information about Broadcast Engine in Programming Guide and in my article dedicated to events:

<http://www.genuinechannels.com/Content.aspx?id=27&type=1>

If a client is able to receive message via IGMP, it will start receiving them via IGMP automatically. If a client is unreachable for IP multicast, it will receive chat messages via IP multicast. You should not bother about this in your applications, the best option is chosen by Broadcast Engine automatically.

Appendix F

Security Session DEMO

(File: "Security Session DEMO.zip")

GTCP Sample

Installation

The sample was written in Visual Studio 2002.

You need to unpack provided archive and open *GTCP 1.0 > Project.sln* solution file. Then rebuild the solution.

Press **F5** to run both client and server in Visual Studio.

Configuration

All network settings are edited in the main dialogs of client and server applications.

Design

This applications is designed to show such Genuine Channels features as

- Traffic counters
- Connection status
- Security Session
- Re-establishing a connection if you unplug and then plug back the network cable
- Intercepting the traffic and Genuine Channels events
- Recognizing the restarting of the server
- Queue overflowing
- Compression
- Traffic encryption
- SSPI support, impersonation and delegation
- Performing requests in the dedicated threads
- Logging (the traffic is being intercepted directly from the log)

Known layer:

- Contains event receiver and event provider interfaces
- Establishes described Security Session interface
- Creates file interface
- Contains built-in 256 bit Rijndael key

Server implements and provides all the necessary business objects and binds them to the known URI. Server fires an event three times per second. The timer is located on the server dialog. You can stop firing events by unchecking the corresponding check box. You can stop the server, change its port and start it again at any moment at run-time.

Client implements the event receiver interface. Client tries to establish a TCP connection with the server located at the entered IP address, subscribes to the event and waits for them.

Client automatically updates traffic counters and calculate CPS value each 500 milliseconds. Client can inform you about each event and show the traffic if you check the corresponding check boxes located on the third panel.

Client subscribes to Genuine Channels events and shows network status,

network event on the third tab page.

The controls on the second panel show Security Session possibilities.

Rendering events

Client application implements its own log writer to intercept and show the traffic (*IEventLogger* interface). You can use built-in *BinaryLog* and *FileLog* classes for writing log information into a file.

Client subscribes to Genuine Channels events to indicate the network status and report about network events.

Security Session

The sample allows designing and creating *Security Sessions* at run-time. Generally you will not need this feature in your application. You will probably define several *Security Sessions* and use them in different contexts.

In order to check the appropriate *Security Session*, you need to choose it and specify call parameters for this Security Session. See *Programming Guide* for additional information. 15 seconds is the maximum error for the *Timeout* parameter.

Created Security Session is applied only to the create file invocation.

If you want to check *SSPI*, you need to know several user accounts at the server. Having created a file on the *NTFS* partition in the specific security context you can open files' properties in *Explorer*, choose *Security* tab and see what user has created it.

SSPI never sends password. You can switch off server's events, enable traffic writing, create a file and then analyze the traffic.

Restarting the server

You will not be able to perform this test in Visual Studio. Open Client and Server *bin\debug* directories in the *Explorer*.

Start the server and the client. Connect to the server from the client. Then close server application and start it again after some time. The client recognizes that the server has been restarted and subscribes event listener to the server again. The connection can be broken due to queue overflow, just switch off the event sending to prevent this.

Breaking the connection

Try to unplug the network cable and plug it back after some time. Client either re-establishes the connection or the connection is closed due to the queue overflow.