

[articles](#) [Q&A](#) [forums](#) [stuff](#) [lounge](#) [?](#)

Search for articles, questions,

[Follow](#)

A C++ Websocket server for realtime interaction with Web clients

Ahmed Charfeddine**Rate me!** 4.81 (22 votes)17 May 2012 [Apache](#)

A Websocket protocol implementation atop the ush Framework real time library plus a demo example featuring four types of communication workflows between the HTML5 web client and the server.

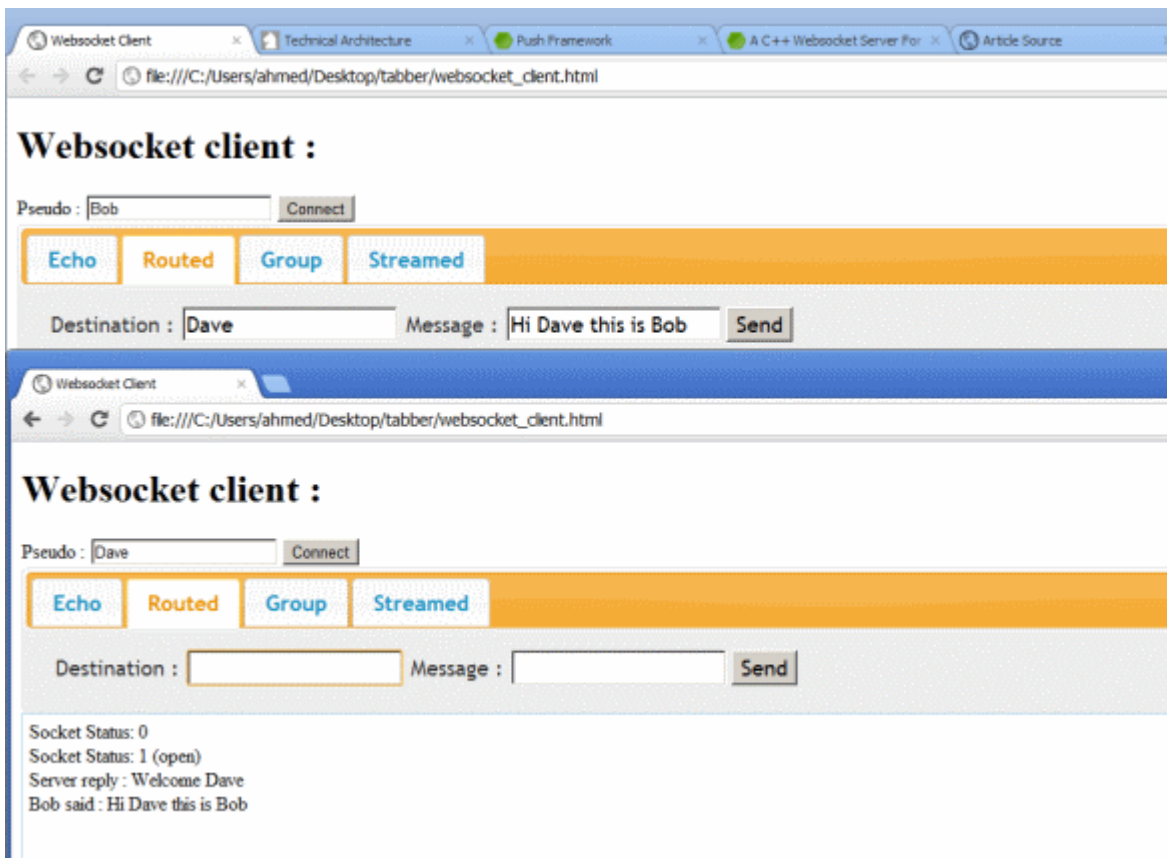


Is your email address OK? You are signed up for our newsletters but your email address is either unconfirmed, or has not been reconfirmed in a long time. Please **click [here](#) to have a confirmation email sent** so we can confirm your email address and start sending you newsletters again. Alternatively, you can **[update your subscriptions](#)**.

[Download source code - 347 KB](#)[Download Push Framework - 187 KB](#)[Download Web Client - 117 KB](#)[Download Websocket Protocol - 22 KB](#)[Download Websocket Server - 9.66 KB](#)

Table of contents

- [Introduction](#)
- [Protocol Extension Layer](#)
- [The Websocket Server](#)
- [The Client](#)
- [References](#)



Introduction

The introduction of the Websocket protocol marks an interesting milestone in the evolution of the web. Finally, it is possible for a web page to open a full duplex connection with a remote server and asynchronously receive data without having to poll for it. This opens the door for plenty of ideas to be easily doable by implementing a web front-end that gets deployed to multiple types of devices and a custom server side application able to handle thousands of simultaneously connected clients and which can be deployed on one low-cost server machine.

In this article, a Websocket server application is developed and we showcase its interaction with a webpage. The solution is based on a real time communication library that was previously published at [CodeProject: Push Framework](#). The protocol layer is devised in an independent library project that can be easily reused by developers.

Protocol Extension Layer

The solution presented in this article is based on Push Framework which provides a foundation for creating real time servers able to manage large numbers of simultaneously connected clients. Push Framework is protocol independent: It is up to us to give protocol details and information by making concrete implementation of the following "abstract classes":

- **IncomingPacket**: What's the prototype for incoming messages, i.e., messages sent by the client and which the server needs to react to?
- **OutgoingPacket**: What's the prototype for outgoing messages? Most protocols are symmetric, so it should be the same as **IncomingPacket**.
- **Protocol**: How incoming packets are deserialized and how outgoing packets are serialized so they get sent through the network.

To make **PushFramework::Protocol** a concrete class, the following virtual methods must be overridden:

- **encodeOutgoingPacket**: it takes an **OutgoingPacket** and encodes it.
- **frameOutgoingPacket**: it takes an encoded **OutgoingPacket** and inserts it into the output socket buffer.

- ✎ **tryDeframeIncomingPacket**: it provides a reference to the received data. These should be examined and an **IncomingPacket** object may be returned.
- ✎ **decodeIncomingPacket**: if **tryDeframeIncomingPacket** succeeds in making an **IncomingPacket**, this function should decode its content.

These methods are requested internally by PF at serialization and de-serialization times and they provide enough abstraction to the majority of protocols.

Dealing with the Websocket protocol, it should be understood that the encoding/decoding part is a separate implementation: This is because, the spec is more of a framing protocol. It details how the payload is encapsulated into a frame along with header information so that it is transmitted into the network. But it does not impose how the payload is "encoded". So the methods that are really relevant to Websocket are **frameOutgoingPacket** and **tryDeframeIncomingPacket**. In our example, we do not do a big job in the encoding stage. Developers might find it suitable to modify this, by adding a JSON layer, for example.

The spec., however, talks about two communication stages in the protocol, which leads us to creating two types of data structures:

- A handshake message: when a connection is accepted at the transport layer, a handshake stage where some negotiation is made begins.
- A websocket data message: this will represent the data messages that are exchanged once the handshake stage is accomplished.

The framing code should distinguish between the two stages.

Hide Shrink ▲ Copy Code

```
int WebsocketProtocol::tryDeframeIncomingPacket( PushFramework::DataBuffer& buffer,
PushFramework::IncomingPacket*& pPacket, int& serviceId,
unsigned int& nExtractedBytes, ConnectionContext* pContext )
{
    if (buffer.GetDataSize() == 0)
        return Protocol::eIncompletePacket;

    WebsocketConnectionContext* pCxt = (WebsocketConnectionContext*) pContext;

    if (pCxt->GetStage() == WebsocketConnectionContext::HandshakeStage)
    {
        WebsocketHandshakeMessage* pMessage =
            new WebsocketHandshakeMessage(buffer.GetBuffer(), buffer.GetDataSize());
        serviceId = 0;
        nExtractedBytes = buffer.GetDataSize();
        pPacket = pMessage;
        return Protocol::Success;
    }

    //In the other cases, we should expect a data message :
    int nMinExpectedSize = 6;
    if (buffer.GetDataSize() < nMinExpectedSize)
        return Protocol::eIncompletePacket;

    BYTE payloadFlags = buffer.getAt(0);
    if (payloadFlags != 129)
        return Protocol::eUndefinedFailure;

    BYTE basicSize = buffer.getAt(1) & 0x7F;
    unsigned __int64 payloadSize;
    int masksOffset;

    if (basicSize <= 125)
    {
        payloadSize = basicSize;
        masksOffset = 2;
    }
    else if (basicSize == 126)
    {
        nMinExpectedSize += 2;
        if (buffer.GetDataSize() < nMinExpectedSize)
```

```

        return Protocol::eIncompletePacket;
    payloadSize = ntohs( *(u_short*) (buffer.GetBuffer() + 2) );
    masksOffset = 4;
}
else if (basicSize == 127)
{
    nMinExpectedSize += 8;
    if (buffer.GetDataSize() < nMinExpectedSize)
        return Protocol::eIncompletePacket;
    payloadSize = ntohl( *(u_long*) (buffer.GetBuffer() + 2) );
    masksOffset = 10;
}
else
    return Protocol::eUndefinedFailure;

nMinExpectedSize += payloadSize;
if (buffer.GetDataSize() < nMinExpectedSize)
    return Protocol::eIncompletePacket;

BYTE masks[4];
memcpy(masks, buffer.GetBuffer() + masksOffset, 4);

char* payload = new char[payloadSize + 1];
memcpy(payload, buffer.GetBuffer() + masksOffset + 4, payloadSize);
for (unsigned __int64 i = 0; i < payloadSize; i++) {
    payload[i] = (payload[i] ^ masks[i%4]);
}
payload[payloadSize] = '\0';

WebSocketDataMessage* pMessage = new WebSocketDataMessage(payload);
serviceId = 1;
nExtractedBytes = nMinExpectedSize;
pPacket = pMessage;

delete payload;
return Protocol::Success;
}

```

The WebSocket Server

In WebSocketServer, we instantiate a main object derived from **PushFramework::Server**, initialize it by describing a Protocol object, a service object, and a **ClientFactory** object, then we start it by calling the **::Start** member function.

When this function is called, many resources are put in place:

- A listening thread
- A pool of threads (IO Workers) to service IO events
- A main thread to manage the overall server structures
- A number of "streaming threads", these will stream out data in broadcast queues to subscribers

The protocol object provided should be derived from the **WebSocketProtocol** class designed in the separate DLL project. As for the **ClientFactory** subclass, it should manage the lifecycle of connected clients. Particularly, it decides on the transition of when a newly accepted connection (**PhysicalConnection**) is transformed into a legitimate client (**LogicalConnection**). In our case, this transition is dependent on two validations: handshake validation as described by the WebSocket protocol, and a login validation where we just require that clients send a unique pseudonym.

Hide Shrink ▲ Copy Code

```

int WebSocketClientFactory::onFirstRequest( IncomingPacket& _request,
    ConnectionContext* pConnectionContext, LogicalConnection*& lpClient,
    OutgoingPacket*& lpPacket )
{

```

```

//received messages belong to a physical connection
//that still did not transform into a logical connection :
//understand in which stage we are :
WebsocketConnectionContext* pCxt = (WebsocketConnectionContext*) pConnectionContext;

if (pCxt->GetStage() == WebsocketConnectionContext::HandshakeStage)
{
    WebsocketHandshakeMessage& request = (WebsocketHandshakeMessage&) _request;
    if (!request.Parse())
    {
        return ClientFactory::RefuseAndClose;
    }

    WebsocketHandshakeMessage *pResponse = new WebsocketHandshakeMessage();
    if (WebsocketProtocol::ProcessHandshake(request, *pResponse))
    {
        lpPacket = pResponse;
        pCxt->SetStage(WebsocketConnectionContext::LoginStage);
    }
    return ClientFactory::RefuseRequest;
    // Will not close the connection, but we still wait
    // for login message to create a logical client.
}

if (pCxt->GetStage() == WebsocketConnectionContext::LoginStage)
{
    WebsocketDataMessage& request = (WebsocketDataMessage&) _request;

    WebsocketClient* pClient = new WebsocketClient(request.GetArg1());
    lpClient = pClient;

    WebsocketDataMessage *pResponse = new WebsocketDataMessage(LoginCommunication);
    pResponse->SetArguments("Welcome " + request.GetArg1());
    lpPacket = pResponse;

    pCxt->SetStage(WebsocketConnectionContext::ConnectedStage);

    return ClientFactory::CreateClient;
}

//Impossible to come here.
}

```

The server business code is organized into "Service" classes. Each is bound to a particular type of request:

[Hide](#) [Copy Code](#)

```

WebsocketServer server;
server.registerService(EchoCommunication, new EchoService, "echo");
server.registerService(RoutedCommunication, new RoutedCommunicationService, "routed");
server.registerService(GroupCommunication, new GroupCommunicationService, "grouped");
server.registerService(StreamedCommunication, new StreamedCommunicationService, "streamed");

```

Let's see the source code for two of them:

[Hide](#) [Copy Code](#)

```

void RoutedCommunicationService::handle( LogicalConnection* pClient, IncomingPacket* pRequest )
{
    WebsocketDataMessage& request = (WebsocketDataMessage&)(*pRequest);
    WebsocketClient& client = (WebsocketClient&) (*pClient);

    LogicalConnection* pRecipient = FindClient(request.GetArg1().c_str());
    if (pRecipient)
    {

```

```

        WebSocketDataMessage response(RoutedCommunication);
        response.SetArguments(client.getKey(), request.GetArg2());
        pRecipient->PushPacket(&response);
    }
}

```

For the forth situation, all the servers care about is handling the subscribe, unsubscribe requests:

[Hide](#) [Copy Code](#)

```

void StreamedCommunicationService::handle( LogicalConnection* pClient, IncomingPacket* pRequest )
{
    WebSocketDataMessage& request = (WebSocketDataMessage&)(*pRequest);
    WebSocketClient& client = (WebSocketClient&) (*pClient);

    string opType = request.GetArg1();

    if (opType == "subscribe")
    {
        broadcastManager.SubscribeConnectionToQueue(client.getKey(), "streamingQueue");
    }

    if (opType == "unsubscribe")
    {
        broadcastManager.UnsubscribeConnectionFromQueue(client.getKey(), "streamingQueue");
    }
}

```

In fact, PF already has a publish/subscribe mechanism, so that we just care about setting up the queues, subscribe clients to these, and publish messages. Message senders do not know about receivers nor are receivers aware about senders. Available data is streamed continuously to those who are interested in it.

The Client

Our web page displays four tabs, in each tab we can trigger one type of operation:

- Echo tab: a message is sent and all the server does is echo it back to the client.
- Routed Communication: a message is sent to a particular client, the server takes care of routing it to its destination.
- Group Communication: a message is sent to the server so it is published to a broadcast queue. We can remotely subscribe to the queue, to begin receiving all content.
- Streamed Communication: allows subscription/unsubscription to a broadcast queue of which content is published automatically. A server thread will do this publishing, so we can experience real time data in the client.

To login, the client enters a pseudonym then clicks "Connect". The server then replies back.

Websocket client :

Pseudo :

Message :

Socket Status: 0
Socket Status: 1 (open)
Server reply : Welcome Bob
Server echo msg : hello world

You can test the different types of communication workflows like echo communication and streamed communication where you get a real time stream of messages automatically created by the server and sent to the webpage:

References

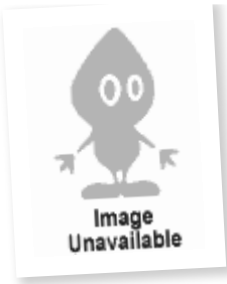
- [The WebSocket RFC](#)
- [Push Framework](#)

License

This article, along with any associated source code and files, is licensed under [The Apache License, Version 2.0](#)

Share

About the Author



Ahmed Charfeddine



Technical Lead
Tunisia



Follow
this Member

Services:

http://www.pushframework.com/?page_id=890

Comments and Discussions

Add a Comment or Question



Email Alerts

Search Comments



First Prev Next

How can i stop the old message 📌

jerrygoo 17-Dec-19 20:46

How can i send a long message ? 📌

Handsome_hl 5-Jan-17 1:25

Re: How can i send a long message ? 📌

wwwsiter 1-Mar-18 5:10

when clients in streamed mode if I close the client ,the websocketserver cpu usage goes to 50% 📌

Member 12291753 27-Apr-16 21:10

How to response close msg 📌

hbsbiscuit 22-Sep-15 7:26

Cross platform mobile client development 📌

Member 11628551 21-Apr-15 1:41

Problem with x64 📌

Kai Liu (Victor) 24-Mar-15 3:36

problem with newer versions of PushFramework 📌

roozbehuofu 5-Dec-13 19:01

Have you tested it for secure connection (using SSL) 📌

anthillbpt 17-Oct-13 11:07

Re: Have you tested it for secure connection (using SSL) 📌

Ahmed Charfeddine 17-Oct-13 13:04

client disconnection 📌

Member 3058875 2-Oct-13 3:24

WSS? 📌

eisenbricher 17-Jul-13 7:58

Is binary data transfer possible using this? 📌

eisenbricher 28-Nov-12 2:59

Re: Is binary data transfer possible using this? 📌

Ahmed Charfeddine 28-Nov-12 16:29

Re: Is binary data transfer possible using this? 📌

eisenbricher 29-Nov-12 5:37

Re: Is binary data transfer possible using this? 📌

Ahmed Charfeddine 30-Nov-12 8:05

Re: Is binary data transfer possible using this? 📌

eisenbricher 4-Dec-12 23:21

i have met a problem 📌

shixingui 22-Nov-12 19:42

Re: i have met a problem 📌

shixingui 23-Nov-12 0:30

Re: i have met a problem 📌

Ahmed Charfeddine 23-Nov-12 3:27

Vote 5 📌

Musthafa (Member 379898) 4-Nov-12 19:55

websocket client 📌

Member 8250121 24-Sep-12 7:18

Re: websocket client 📌

Ahmed Charfeddine 24-Sep-12 8:23

Re: websocket client 📌











Member 8250121 25-Sep-12 6:53

Re: websocket client 📌

Ahmed Charfeddine 25-Sep-12 6:58

Refresh

1 2 Next »

 General  News  Suggestion  Question  Bug  Answer  Joke  Praise  Rant  Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#)

[Advertise](#)

[Privacy](#)

[Cookies](#)

[Terms of Use](#)

Layout: [fixed](#) | [fluid](#)

Article Copyright 2012 by Ahmed Charfeddine
Everything else Copyright © [CodeProject](#), 1999-2020

Web01 2.8.200504.1