Peer to Peer Communications

By

Corey Gilbert

Submitted to
the Faculty of the Information Technology Program
in Partial Fulfillment of the Requirements for
the Degree of Bachelor of Science
in Information Technology

University of Cincinnati
College of Applied Science

June 2008
Peer to Peer Communications

by

Corey Gilbert

Submitted to
the Faculty of the Information Technology Program
in Partial Fulfillment of the Requirements for
the Degree of Bachelor of Science
in Information Technology

__________________________ __________________
Corey Gilbert        Date

__________________________ __________________
Dr. Hazem Said, Faculty Advisor     Date

__________________________ __________________
Dr. Hazem Said, Department Head     Date
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>ii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
</tbody>
</table>

1. Statement of the Problem  
   1.1 Definition of the Need  
   1.2 Product Description  
   1.3 Component Purposes  
     1.3.1 P2P Communications Server  
     1.3.2 P2P Communications API  
     1.3.3 P2P Communications Web site  

2. User Profiles  
   2.1 Site Administrator  
   2.2 Software Developers  
   2.3 Extended Users  

3. Design Protocol  
   3.1 Application Protocols  
   3.2 Component Architecture  
     3.2.1 Server Architecture  
     3.2.2 Client Architecture  
   3.3 P2PComm API Activation  
     3.3.1 Authentication  
     3.3.2 Registration  
     3.3.3 Securing Communications (RSA)  
     3.3.4 Communicating (UDP)  
   3.4 Area of Information Technology  
   3.5 Database  

4. Deliverables  

5. Development  
   5.1 Timeline  
     5.1.1 Senior Design I Accomplishments  
     5.1.2 Senior Design II Accomplishments  
     5.1.3 Senior Design III Accomplishments  
   5.2 Budget  
   5.3 Hardware and Software  

6. Proof of Design  
   6.1 P2PComm API  
   6.2 P2P Communications Server  

25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3 P2PComm.net Web site</td>
<td>27</td>
</tr>
<tr>
<td>6.4 Demo Application</td>
<td>31</td>
</tr>
<tr>
<td>7. Conclusion and Recommendations</td>
<td>32</td>
</tr>
<tr>
<td>7.1 Conclusion</td>
<td>32</td>
</tr>
<tr>
<td>Appendix A</td>
<td>33</td>
</tr>
<tr>
<td>References</td>
<td>38</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>System Diagram</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Application Protocols</td>
<td>5</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Server Architecture</td>
<td>6</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Client Architecture</td>
<td>7</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Authentication</td>
<td>8</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Device Registration</td>
<td>9</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Securing Communications (RSA)</td>
<td>10</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Communicating (UDP)</td>
<td>11</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Database Diagram</td>
<td>12</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Budget</td>
<td>14</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Authentication Web Service</td>
<td>16</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Securing Communications Client-Side</td>
<td>18</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Securing Communications Server-Side</td>
<td>19</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Receive</td>
<td>20</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Send</td>
<td>21</td>
</tr>
<tr>
<td>Figure 16</td>
<td>ClientProcessor Data Message</td>
<td>22</td>
</tr>
<tr>
<td>Figure 17</td>
<td>ClientProcessor Source Message</td>
<td>23</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Client Processor Unsecure Source Message</td>
<td>23</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Raising Events on the Main Thread</td>
<td>24</td>
</tr>
<tr>
<td>Figure 20</td>
<td>ServerProcessor Enumeration</td>
<td>25</td>
</tr>
<tr>
<td>Figure 21</td>
<td>DeviceMonitor Infinite Routine</td>
<td>26</td>
</tr>
</tbody>
</table>
Figure 22: P2PComm.net Home Page 27
Figure 23: P2PComm.net Login and Registration 28
Figure 24: P2PComm.net Server Usage Total 29
Figure 25: P2PComm.net Server Usage Summaries 29
Figure 26: P2PComm.net Server Usage Summaries by Account 30
Figure 27: P2PComm.net Account Activation 30
Figure 28: Whiteboard Online 31
Figure 29: MessageSerializer Class 33
Abstract

Peer to Peer Communications (P2PComm) is a system designed to mitigate the frustrations of developing real-time Internet based applications. It is a tool used by software developers to make communicating across the Internet easier. The utility automatically handles authentication, security, and data exchange between computers connected over the Internet. The system was designed such that developers with little experience can create real-time Internet based applications with no prior knowledge of network communications. There are also tools available on the P2PComm web site that can assist the developer while he/she is developing programs that facilitates the utility.

P2PComm uses Microsoft’s .NET framework 2.0 and was developed using Microsoft’s Visual Studios 2005 development platform. It uses Microsoft’s SQL Server 2005 as the backend database.

P2PComm is managed using a Web 2.0 interface. All account creations and registrations are handled through the Web site.

P2PComm has been tested and is fully functional. It is available on the Internet at http://P2PComm.net
Peer to Peer Communications

1. Statement of the Problem

1.1 Definition of the Need

Peer to Peer Communications (P2PComm) is a Microsoft’s .Net framework 2.0 programming library and allows developers to easily create real-time internet based applications. Software that communicates using the internet is becoming more common every day. Developing software that communicates across the Internet requires two components: a server component and a client component. Developing these components can take time and resources away from software developers. There is a need to make programming real-time Internet based applications easier.

1.2 Product Description

The P2PComm system handles communications between computers across the Internet. The system does not allow cross communications between accounts. Figure 1 is a system diagram that shows how devices are grouped together. Devices that use the same account can communicate. In Figure 1, “Device A” from cannot communicate from “Device C” because they use separate accounts. However, the server will allow communications from “Device A” to “Device B.” A developer using the P2PComm system will register their application on the Web site. When he/she implements the P2PComm Application Programming Interface (API), he/she must supply the API with the registered application name and password. This will force all instances of the application to use the same account allowing communications to be exchanged across the internet.
The P2PComm API was designed to be simple. After a software developer has downloaded the API, he/she will add the library reference to his/her project. Once the developer is finished creating the application, the application will use the API to communicate across the internet to other computers running the same application. The developer facilitating the utility does not need any experience programming applications that communicate over networks or the Internet.

The P2PComm Web site allows application registration and reporting. The Web site has a dashboard view that shows current activity of the registered account. The dashboard view also shows the data consumption used by the server. This keeps the developer informed of the status of their application.
1.3 Component Purposes

The P2PComm system is composed of three main components.

1.3.1 P2P Communications Server

There was a need to keep track of client-side Internet Protocol (IP) addresses. Without the IP information of connecting clients, clients would not be able to communicate over the Internet. There was also a need to keep the names of clients unique. The server will not allow a client to login if the name that the client specified is already registered. The P2P Communications Server keeps communications.

1.3.2 P2P Communications API

There was a need to make all applications using the P2PComm system to understand the same protocols for communication. Once the API is added as a reference to an application, it can send and receive information to the server using the standard application protocol defined by the system.

1.3.3 P2P Communications Web site

Making sure the data being sent across the Internet is accurate is not an easy task to do. There was a need to be able to easily view the information that is processed by the server. On the P2PComm Web site, there are dashboard views that show this information. It helps developers see the information being transmitted between clients.

2. User Profiles

There are three user groups for the P2PComm system.

2.1 Site Administrator

The site administrator is responsible for activating new accounts that get requested. Software developer that is registering an application is required to fill out a
description field that explains the need of the service. The site administrator can then either accept the registration or deny it based on the criteria in the description field. They are also responsible for server maintenance.

2.2 Software Developers

Software Developers are the main users of the system. They will be the ones that register the application, download the P2PComm API and implement it in an application. They also have the ability to view the usage of the application. The P2PComm Web site shows the amount of data consumed by the server.

2.3 Extended Users

Extended users are the people that use the applications that the software developers create. The P2PComm system should be transparent to them. However, software developers that use the P2PComm system need to provide documentation on how to setup the internal network of the extend users. Extended users that use proxy servers or firewalls will need to configure his/her network to pass data forward to the desired clients.

3. Design Protocol

There are three different sections that make up the design protocols of the P2PComm system.

3.1.1 Application Protocols

The application protocols make up a list of message types and structures used by the P2P Communications Server and the P2PComm API. Figure 2 shows all the different protocols that are used by the system. The “Packet Structures” that are shown in Figure 2 display how the messages are broken down into before sending it across the network.
3.2 Component Architecture

The P2P Communications Server has a registered collection of accounts. Each registered account can have multiple devices. The client architecture only handles one device. Figures 3 and 4 show the different processes that are used by the server and
client components. The different colored arrows represent the different threads running in the component.

3.2.1 Server Architecture

Figure 3: Server Architecture
3.2.2 Client Architecture

Client Architecture

![Client Architecture Diagram]

Figure 4: Client Architecture

3.3 P2PComm API Activation

An application that uses the P2PComm API has to be activated before it can communicate with the server. Figures 5-8 show the process of activating the API. The Figures are displayed in the order that they are processed.
3.3.1 Authentication

![Diagram of Authentication Process]

**Figure 5: Authentication**
3.3.2 Registration

**Device Registration**

- Successful login
- Enter device name
- Register device name
- Registration successful
- Secure communications
- Name already registered
- Register device
- No
- Yes

**Figure 6: Device Registration**
3.3.3 Securing Communications (RSA)

Securing Communications (RSA)

Successful registration
Create new private and public encryption keys
Send the device's public key to the server
Store server's public key
Begin communications

Store device's public key
Store device's internet address
Create new private and public encryption keys
Send the server's public key to the device

Figure 7: Securing Communications (RSA)
3.3.4 Communicating (UDP)

**Communicating (UDP)**

![Diagram of data communication process]

**Figure 8: Communicating (UDP)**

### 3.4 Areas of Information Technology

There are three areas of Information Technology being used in this application.

- **Programming:** P2PComm is developed using VB.NET and developed with Microsoft’s Visual Studio 2005 development platform.

- **Database:** Microsoft’s SQL Server 2005 was used as a backend database. Three tables make up the P2PComm system.

- **Networking:** The P2P Communications Server and the P2PComm API uses the User Datagram Protocol (UDP) to communicate at low levels across the Internet.

### 3.5 Database

The backend database consists of three tables. Figure 9 shows their relationships.

The “Accounts” table is used to store the account information and password. It also
stores the username and password that the API uses. The “Devices” table gets populated when the API becomes activated. Once the API is deactivated, it removes the entry that was added upon activation. All messages received by the server are log in the “MessageLog” table.

![Database Diagram](image)

**Figure 9: Database Diagram**

4. Deliverables

   ➢ P2P Communications Server
     
     o Monitors Device Status
     
     o Sends/Receives Messages
     
     o Activates/Deactivates Devices
P2PComm API
- Activates/Deactivates
- Sends/Receives Messages
- Maintains Active Users Status

P2PComm.net Web site
- Account Registration
- Server Usage Reports
- Server Active Device Status

Demo Application
- Whiteboard Online

Implementation guidelines

5. Development

5.1.1 Senior Design I Accomplishments
- Gained high level understanding of concepts behind Peer to Peer Communications.
- Setup server environment.
- Researched communicating using UDP.
- Wrote the proposal
- Presented proposal for development of Senior Design project.

5.1.2 Senior Design II Accomplishments
- Completed authentication web services.
- Completed registration web services.
- Completed securing communications web services.
- Setup database tables.
- Completed P2PComm API.
- Document Design Freeze.
- Present Design Freeze.

5.1.3 Senior Design III Accomplishments

- Completed P2P Server Component
- Completed P2PComm.net Web site
- Completed Demo Application
- Completed Implementation Guidelines
- Presented project at Tech Expo

5.2 Budget

All of the equipment used in the Peer to Peer Communications project was already bought and purchased. A server was setup running Microsoft’s Server 2003. Also installed on the server was Microsoft’s Internet Information Services, Microsoft’s SQL Server 2005. Microsoft’s Visual Studios 2005 was used to develop the software.

Figure 10 shows a detailed list of associated cost that went with the project

<table>
<thead>
<tr>
<th>Hardware / Software</th>
<th>Retail</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2PComm.net domain registration</td>
<td>$10</td>
<td>$10</td>
</tr>
<tr>
<td>High speed Internet connection</td>
<td>$50/m</td>
<td>$50/m</td>
</tr>
<tr>
<td>Server Hardware</td>
<td>$700</td>
<td>$700</td>
</tr>
<tr>
<td>Microsoft’s SQL Server 2005</td>
<td>$709</td>
<td>Used MSDN</td>
</tr>
<tr>
<td>Microsoft’s Visual Studio.Net 2005</td>
<td>$689.99</td>
<td>Used MSDN</td>
</tr>
<tr>
<td>Microsoft’s Server 2003</td>
<td>$384.99</td>
<td>Used MSDN</td>
</tr>
<tr>
<td>Total</td>
<td><strong>Actual: $2,543.98</strong></td>
<td><strong>Spent: $760</strong></td>
</tr>
</tbody>
</table>
5.3 **Hardware and Software**

The Peer to Peer Communications project was built using the following hardware and software.

- **Hardware**
  - **Server**: Required to process communications and host website

- **Software**
  - **Windows Server 2003**: Required for Internet Information Services.
  - **Internet Information Services**: Used for hosting P2PComm.net
  - **Microsoft’s SQL Server 2005**: Relational database for project.
  - **Microsoft Visual Studio .Net 2005**: Development platform for VB.net code

6. **Proof of Design**

There were four main components that were design to facilitate the Peer to Peer Communications system.

6.1 **P2PComm API**

Before the P2PComm API starts communicating with the server it must first go through an activation process. The first step of activation is the “LoginDevice” routine. Figure 11 shows the “LoginDevice” routine. If the validation of any of the parameters passed into the login method fail, the method returns a blank global unique identifier (GUID). After the validation of the parameters pass, the next step is to call the stored procedure “sp_CheckMobilePassword.” This stored procedure checks the registered username and password that is being passed into the routine. If it is successful, the
routine returns a new GUID for the session identifier and returns the account GUID of the username being passed into the routine. The “LoginDevice” routine resides on the P2P Communications Server. There is a wrapper class called “WSSSLAuthentication” that makes the “LoginDevice” call.

```
<WebMethod()>  
    Public Function LoginDevice(ByVal strUsername As String, ByVal strPassword As String) As Guid()
        Dim AccountGuid As New Guid("00000000-0000-0000-0000-000000000000")
        Dim PasswordParam As SqlParameter
        Dim SessionGuid As New Guid("00000000-0000-0000-0000-000000000000")
        Dim SqlConn As New SqlConnection(ConfigurationManager.ConnectionStrings("p2pcommDBPublic").ConnectionString)
        Dim SqlCmd As New SqlCommand
        Dim SqlDr As SqlDataReader
        Dim Encoder As New System.Text.ASCIIEncoding
        If strUsername = Nothing Then Return New Guid() {SessionGuid, AccountGuid}
        If strPassword = Nothing Then Return New Guid() {SessionGuid, AccountGuid}
        If Trim(strUsername) = "" Then Return New Guid() {SessionGuid, AccountGuid}
        If Trim(strPassword) = "" Then Return New Guid() {SessionGuid, AccountGuid}
        SqlCmd.Connection = SqlConn
        SqlCmd.CommandType = Data.CommandType.StoredProcedure
        SqlCmd.CommandText = "sp_CheckMobilePassword"
        SqlCmd.Parameters.Add(New SqlParameter("MobileUsername", SqlSafe(strUsername)))
        PasswordParam = New SqlParameter("MobilePassword", SqlDbType.Binary, 16)
        PasswordParam.Value = Convert.FromBase64String(strPassword)
        SqlCmd.Parameters.Add(PasswordParam)
        SqlCmd.Connection.Open()
       SqlDr = SqlCmd.ExecuteReader()
        If SqlDr.Read() Then
            AccountGuid = New Guid(SqlDr.Item("AccountGuid").ToString)
            SessionGuid = CreateSessionGuid(DeviceGuid, AccountGuid)
        End If
```
The second step in the activation process is the device registration. It also runs a stored procedure that checks the name being passed into the routine. If the name being passed to the routine is unique it returns a new GUID. Otherwise, it returns a blank GUID. The “RegisterDevice” routine call is made from the “WSSSLDeviceRegistry” class.

The final step of activation comes from a class called “UDPCommunicator.” This is the class responsible for sending/receiving and encrypting/decrypting the data being passed to it. When the “SecureCommunications” routine is called, a new encryption key pair is created for the device using the API. Figure 12 shows a portion of the “SecureCommunications” routine. After the “RSAEncryptionServiceProvider” is created, it is passed into a new instance of “MessageSerializer.” The “MessageSerializer” is the class responsible for converting the raw information received by the network into a format that the API understands. The “MessageSerialier” class can be viewed in Appendix A. Figure 13 shows the server-side “SecureCommunications.” It also creates its encryption key pair and passes its public key back to the client. After this routine is finished, both the client and server can encrypt and decrypt each others messages.
_LocalPort = LocalPort
_EncryptionServiceProvider = New RSAEncryptionServiceProvider()
_MessageSerializer = New MessageSerializer(_EncryptionServiceProvider)

Try
    bytRemotePublicKey = _DeviceSecurity.SetEncryptionKeys(_Authenticator.SessionGuid,
               _EncryptionServiceProvider.LocalPublic, _LocalPort)
    _EncryptionServiceProvider.RemotePublic = bytRemotePublicKey
    If Not _EncryptionServiceProvider.IsRemoteKeySet Then
        Select Case bytRemotePublicKey(0)
        Case 0
            RaiseEvent SecureCommunicationFailed("Securing communication failed: No SessionGuid")
        Case 1
            RaiseEvent SecureCommunicationFailed("Securing communication failed: Sent Invalid crypto key size")
        Case 2
            RaiseEvent SecureCommunicationFailed("Securing communication failed: Device port out of range")
        Case Else
            RaiseEvent SecureCommunicationFailed("Securing communication failed: Received Invalid crypto key size")
        End Select
    Else
        _IsSecure = True
        RaiseEvent CommunicationSecured(_UDPCommunicator)
    End If
Catch ex As Exception
    _IsSecure = False
    RaiseEvent SecureCommunicationFailed("Securing communication failed: " & ex.Message)
End Try

Figure 12: Securing Communications Client-Side
<WebMethod()> _
Public Function SetEncryptionKeys(ByVal SessionGuid As Guid, ByVal DevicePublicKey() As Byte, ByVal DevicePort As Integer) As Byte()
    If SessionGuid = Nothing Then Return New Byte() {0}
    If Not DevicePublicKey.Length = 116 Then Return New Byte() {1}
    If DevicePort > 65535 Or DevicePort < 1 Then Return New Byte() {2}
    Dim ServerKeyPair() As Byte
    Dim ServerPublicKey() As Byte
    Dim ParamServerKeyPair As New SqlParameter("ServerKeyPair", SqlDbType.Binary, 452)
    Dim ParamDevicePublicKey As New SqlParameter("DevicePublicKey", SqlDbType.Binary, 116)
    Dim SqlConn As New SqlConnection(ConfigurationManager.ConnectionStrings("p2pCommDBUser").ConnectionString)
    Dim SqlCmd As New SqlCommand
    Dim RowsAffected As Integer
    _RSACSP = New RSACryptoServiceProvider(768)
    ServerKeyPair = _RSACSP.ExportCspBlob(True)
    ServerPublicKey = _RSACSP.ExportCspBlob(False)
    ParamServerKeyPair.Value = ServerKeyPair
    ParamDevicePublicKey.Value = DevicePublicKey
    SqlCmd.Connection = SqlConn
    SqlCmd.CommandType = CommandType.StoredProcedure
    SqlCmd.CommandText = "sp_AssignCryptoKeys"
    SqlCmd.Parameters.Add(New SqlParameter("SessionGuid", SqlDbType.VarChar, 50))
    SqlCmd.Parameters.Add(New SqlParameter("DeviceHost", SqlDbType.VarChar, 50))
    SqlCmd.Parameters.Add(New SqlParameter("DevicePort", SqlDbType.VarChar, 50))
    SqlCmd.Parameters.Add(ParamServerKeyPair)
    SqlCmd.Parameters.Add(ParamDevicePublicKey)
    SqlCmd.Connection.Open()
    RowsAffected = SqlCmd.ExecuteNonQuery()
    SqlCmd.Dispose()
    Return ServerPublicKey
End Function

Figure 13: Securing Communications Server-Side
After both the client and server has exchanged their encryption keys, the API will startup its infinite routines that keep the API communicating with the server. Figure 14 and 15 shows the routines that run infinitely until the API is deactivated. The “Receive,” routine receives the raw bytes of data that come from the network.

```vbnet
Private Sub Receive()
    Dim b() As Byte
    Dim listenEP As IPEndPoint

    listenEP = New IPEndPoint(IPAddress.Any, 0)

    Try
        While True
            If _bStop Then Exit While

            Try
                'Set endpoint
                listenEP.Address = _listenEP.Address
                listenEP.Port = _listenEP.Port

                'Get data
                b = _udpClient.Receive(listenEP)

                'Add data to buffer
                _rBuffer.Enqueue(b)
            Catch ex As SocketException
                'Receive routine timed out
                End Try
            End While
        Catch ex As Exception
        Finally
            'Increment udp flag
            _intUdpClientFlag += 1
        End Try
    End Sub

    'Set UDPWrapper to inactive
    _bIsActive = False
End Sub
```

**Figure 14: Receive**
The send routine is constantly runs until the API is deactivated. It runs every 100 milliseconds. “_sBuffer” is the send buffer. When it has items to send it sends then.

When no items are present in the buffer routine waits 100 milliseconds and runs again.

```vbnet
Private Sub Send()
    Dim b() As Byte
    Try
        While True
            If bStop Then Exit While
            While _sBuffer.Count > 0
                'Dequeue send buffer
                b = _sBuffer.Dequeue()
                'Send Bytes
                udpClient.Send(b, b.Length, _remoteEP)
            End While
            Thread.Sleep(TRANSCEIVER_SENDING_TICK_RATE) ' wait for tick rate.
        End While
    Catch ex As Exception
    Finally
        'Increment flag by one.
        _intUdpClientFlag += 1
        'When flag is two then it means we have got out of loops in Send and Receive.
        While (Not _intUdpClientFlag = 2) : Thread.Sleep(10) :
    End While
End Sub
```

**Figure 15: Send**

Once a message received and de-serialized, it is passed to the “ClientProcessor” class which determines how to process the message. Figures 16, 17 and 18 show all the different types of messages that the “ClientProcessor” knows how to process. The “ClientProcessor” passes messages to the “UDPCommunicator” which passes messages to the “CommDevice”
Public Sub ProcessDataMessage(ByVal dm As DataMessage, ByRef Close As Boolean)
    Select Case dm.Type
        Case MESSAGETYPES.SERVER_ACKNOWLEDGEMENT
            MessageAcknowledged(dm)
            RaiseEvent LogExtendedEvent("ClientProcessor", "(R) SERVER_ACKNOWLEDGEMENT (s=" & dm.Seq & ") (t=" & dm.Type & ") (txt=" & dm.Text & ")")
        Case MESSAGETYPES.FRIEND_DISCONNECTED
            Dim str() As String = dm.Text.Split("|")
            RaiseEvent ActiveDevicesUpdate(str(0), New Guid(str(1)), MESSAGETYPES.FRIEND_DISCONNECTED)
            RaiseEvent LogExtendedEvent("ClientProcessor", "(R) FRIEND_DISCONNECTED (s=" & dm.Seq & ") (t=" & dm.Type & ") (txt=" & dm.Text & ")")
        Case MESSAGETYPES.FRIEND_LOGGED_IN
            Dim str() As String = dm.Text.Split("|")
            RaiseEvent ActiveDevicesUpdate(str(0), New Guid(str(1)), MESSAGETYPES.FRIEND_LOGGED_IN)
            RaiseEvent LogExtendedEvent("ClientProcessor", "(R) FRIEND_LOGGED_IN (s=" & dm.Seq & ") (t=" & dm.Type & ") (txt=" & dm.Text & ")")
        Case MESSAGETYPES.FRIEND_IDOL
            Dim str() As String = dm.Text.Split("|")
            RaiseEvent ActiveDevicesUpdate(str(0), New Guid(str(1)), MESSAGETYPES.FRIEND_IDOL)
            RaiseEvent LogExtendedEvent("ClientProcessor", "(R) FRIEND_IDOL (s=" & dm.Seq & ") (t=" & dm.Type & ") (txt=" & dm.Text & ")")
        Case MESSAGETYPES.FRIEND_LOGGED_OUT
            Dim str() As String = dm.Text.Split("|")
            RaiseEvent ActiveDevicesUpdate(str(0), New Guid(str(1)), MESSAGETYPES.FRIEND_LOGGED_OUT)
            RaiseEvent LogExtendedEvent("ClientProcessor", "(R) FRIEND_LOGGED_OUT (s=" & dm.Seq & ") (t=" & dm.Type & ") (txt=" & dm.Text & ")")
        Case Else
            RaiseEvent LogExtendedEvent("ClientProcessor", "(R) DATA_MESSAGE type " & dm.Type & ") is not supported (s=" & dm.Seq & ") (txt=" & dm.Text & ")")
            Exit Sub 'Unknown Data Message Type
    End Select
End Sub
The "CommDevice" class is a user control that can be added to windows forms. This is the item that software developers will be able to interact with. The software
developer will need to activate the “CommDevice.” The “CommDevice” can start communicating after the activation process has occurred. Because the P2PComm API runs on separate threads, all events that get sent to the user interface (UI) must be checked to see if an “Invoke” is required. Routines running on separate threads can not access the main UI thread. All of the events that get raised to the UI must be executed by the main thread. Figure 19 shows how this is done.

```vbnet
Private Delegate Sub dActiveDevicesUpdate(ByVal DeviceName As String, ByVal DeviceGuid As Guid, ByVal MessageType As MESSAGETYPES)

Private Sub Communicator_ActiveDevicesUpdate(ByVal DeviceName As String, ByVal DeviceGuid As Guid, ByVal MessageType As MESSAGETYPES) Handles _Communicator.ActiveDevicesUpdate
    If Me.InvokeRequired Then
        Dim d As New dActiveDevicesUpdate(AddressOf Communicator_ActiveDevicesUpdate)
        Me.Invoke(d, New Object() {DeviceName, DeviceGuid, MessageType})
    Else
        Select Case MessageType
            Case MESSAGETYPES.FRIEND_DISCONNETED
                _ActiveDevices.Remove(_ActiveDevices(DeviceName))
            Case MESSAGETYPES.FRIEND_LOGGED_IN
                Dim d As New Device
                d.DeviceGuid = DeviceGuid
                d.Account = _Authenticator.AccountGuid
                d.Name = DeviceName
                _ActiveDevices.Add(d)
            Case MESSAGETYPES.FRIEND_LOGGED_OUT
                _ActiveDevices.Remove(_ActiveDevices(DeviceName))
            Case Else
                'Do nothing
        End Select
        RaiseEvent ActiveDevicesUpdate(DeviceName, MessageType)
    End If
End Sub
```

Figure 19: Raising Events on the Main Thread
6.2 P2P Communications Server

The P2P Communications Server functions similar to the “ClientProcessor.” The main difference is that it holds a collection of “ServerProcessors.” Each collection of “ServerProcessors” is held in a collection. This collection is held in a class called “AccountProcessor.” All “AccountProcessors” are held in a class that is held by the root element of the server. The “ServerProcessor” main responsibility is to send messages to all devices are logged in to the server. Figure 20 shows the how the “ServerProcessor” collection enumerates through all the active devices using the server. The server runs through this same process for all the different types of messages. When a broadcast is received, it must enumerate through all the active devices and send the broadcast to each one of the active devices.

```vbnet
Public Sub LogOutDevice(ByVal Source As Guid)
    If Not Contains(Source.ToString) Then Exit Sub
    Dim DeviceName As String = Item(Source.ToString).Device.Name
    For Each SP As ServerProcessor In _Col
        If SP.Device.DeviceGuid = Source Then Continue For
        SP.SendDeviceUpdate(DeviceName, Source, MESSAGETYPES.FRIEND_LOGGED_OUT)
    Next
    RaiseEvent OnDeviceRemoval(Source)
    DeviceMonitor.RemoveDevice(Source)
    Item(Source.ToString).Dispose()
    Remove(Source.ToString)
End Sub
```

Figure 20: ServerProcessor Enumeration
The P2P Communications Server is also responsible for keeping all active devices updated on device statuses. The server uses a class called “DeviceMonitor” that has an infinite routine that constantly checks when the last heartbeat of a device was. If the last heartbeat is greater than 30 seconds, the server will disconnect the device from the server. Figure 21 shows the infinite loop that runs in the “DeviceMonitor” class.

```vbnet
While SqlDr.Read
    i += 1
    If TypeOf SqlDr("LastHeartbeat") Is System.DBNull Then
        Continue While
    End If
    dt = SqlDr("LastHeartbeat")
    If TypeOf SqlDr("DeviceName") Is System.DBNull Then
        If DateDiff(DateInterval.Second, dt, Now()) > 30 Then
            RemoveDevice(New Guid(SqlDr("DeviceGuid").ToString))
            RaiseEvent LogEvent("DeviceMonitor", "Unregistered Device " & SqlDr("DeviceGuid").ToString & " got disconnection")
        Else
            d.Account = New Guid(SqlDr("AccountGuid").ToString)
            d.DeviceGuid = New Guid(SqlDr("DeviceGuid").ToString)
            Select Case True
                Case DateDiff(DateInterval.Second, dt, Now()) > DEFAULT_DISCONNECT_TIME
                    RemoveDevice(d.DeviceGuid)
                    RaiseEvent OnDeviceDisconnect(d)
                Case DateDiff(DateInterval.Second, dt, Now()) > DEFAULT_IDOL_TIME
                    If SqlDr("DeviceStatus") = "IDOL" Then
                        Continue While
                    End If
                    SetDeviceStatus(d.DeviceGuid, "IDOL")
                    RaiseEvent OnDeviceIdol(d)
                Case Else
                    'Do Nothing
            End Select
            If Not SqlDr("DeviceStatus") = "ACTIVE" Then
                SetDeviceStatus(d.DeviceGuid, "ACTIVE")
            End If
        End If
    End While
```

**Figure 21: DeviceMonitor Infinite Routine**
6.3 P2PComm.net Web site

The P2PComm.net Web site holds many purposes. Figure 22 shows the main page.

Figure 22: P2PComm.net Home page

The main purpose for the P2PComm.net Web site is registration. Figure 23 shows the registration and login page.
After a user is logged in to the website, he/she has the ability to view the current servers usage for the account logged into. In Figure 24, the server usage totals are displayed. These values are updated every time the API is used. Figure 25 breaks down the data by message type and displays the usage per each message. Figure 26 is same view as Figure 25 but breaks the data down even farther by group by account. Figure 26 is only available by the site administrator.
### Figure 24: P2PComm.net Server Usage Total

<table>
<thead>
<tr>
<th>total messages processed</th>
<th>total data processed (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>253</td>
<td>7302</td>
</tr>
<tr>
<td>19717</td>
<td>736305</td>
</tr>
</tbody>
</table>

### Figure 25: P2PComm.net Server Usage Summaries

<table>
<thead>
<tr>
<th>message type</th>
<th>messages processed</th>
<th>data processed (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROADCAST_MESSAGE</td>
<td>394</td>
<td>13539</td>
</tr>
<tr>
<td>FRIEND_DISCONNECTED</td>
<td>84</td>
<td>3024</td>
</tr>
<tr>
<td>FRIEND_IDOL</td>
<td>97</td>
<td>3492</td>
</tr>
<tr>
<td>FRIEND_LOGGED_IN</td>
<td>237</td>
<td>8532</td>
</tr>
<tr>
<td>FRIEND_LOGGED_OUT</td>
<td>116</td>
<td>4176</td>
</tr>
<tr>
<td>PRIVATE_MESSAGE</td>
<td>11</td>
<td>66</td>
</tr>
<tr>
<td>UNKNOWN_MESSAGE_TYPE</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>UNSECURE_BROADCAST_MESSAGE</td>
<td>19027</td>
<td>710746</td>
</tr>
</tbody>
</table>
The site administrator is also responsible for activating and deactivating accounts. There is a link on the left navigation panel that says “Account Activation.” Figure 27 shows how this page looks.
6.4 Demo Application

The demo application was created to demonstrate the different ways that the API can be used. The demo application was named Whiteboard Online. It was a drawing program that allows others to view the same digital whiteboard across the Internet. Everyone that was logged into the application saw what the others were drawing. Figure 28 shows how the application looks.

Figure 28: Whiteboard Online
8. Conclusion and Recommendations

8.1 Conclusion

In conclusion, The P2PComm Project is up and working. The system allows for easy Internet communications between applications. Software developers can use this system to build real-time Internet based applications without any knowledge of programming applications to communicate across the Web.
Appendix A

The class “MessageSerializer” is used by both the P2P Communications Server and the P2PComm API. It is responsible for taking raw bytes of data and converting them into the appropriate type. It must be created with an “IEncryptionServiceProvider” so it knows how to encrypt and decrypt the data it processes. It raises an event every time it serializes or de-serializes a message. It also raises the “LogEvent” event so any program that uses the “MessageSerializer,” can log its activity.

```vbnet
Public Class MessageSerializer
    Implements IDisposable

    Private _ESP As IEncryptionServiceProvider
    Private _ESPLockObj As New Object

    Public Event LogEvent(ByVal Type As String, ByVal Desc As String)
    Public Event OnMessageDeSerialized(ByVal m As Message)
    Public Event OnMessageSerialized(ByVal b() As Byte)
    Public Event OnDataMessageDeSerialized(ByVal dm As DataMessage)
    Public Event OnDataMessageSerialized(ByVal b() As Byte)
    Public Event OnSourceMessageDeSerialized(ByVal sm As SourceMessage)
    Public Event OnSourceMessageSerialized(ByVal b() As Byte)
    Public Event OnSourceDestinationMessageDeSerialized(ByVal sdm As SourceDestinationMessage)
    Public Event OnSourceDestinationMessageSerialized(ByVal b() As Byte)
    Public Event OnUnsecureSourceMessageDeSerialized(ByVal usm As UnsecureSourceMessage)
    Public Event OnUnsecureSourceMessageSerialized(ByVal b() As Byte)

    Public Sub New(ByVal ESP As IEncryptionServiceProvider)
        _ESP = ESP
    End Sub

    Public Sub ProcessMessage(ByVal b() As Byte)
        If _ESP Is Nothing Then Throw New Exception("Encryption Service Provider is not set")
        Select Case MessageUtility.GetMessageType(b)
            Case MESSAGETYPES.MESSAGE
```
RaiseEvent LogEvent("MessageSerializer", "Message type is MESSAGE")
    Dim m As New Message(b)
m.DeSerialize(Nothing)
    If Not m.IsValidMessage Then Exit Sub
    ProcessReceivedMessage(m)
Case MESSAGETYPES.DATA_MESSAGE
    RaiseEvent LogEvent("MessageSerializer", "Message type is DATA_MESSAGE")
    Dim m As New DataMessage(b)
m.DeSerialize(_ESP)
    If Not m.IsValidMessage Then Exit Sub
    ProcessReceivedDataMessage(m)
Case MESSAGETYPES.SOURCE_MESSAGE
    RaiseEvent LogEvent("MessageSerializer", "Message type is SOURCE_MESSAGE")
    Dim m As New SourceMessage(b)
m.DeSerialize(_ESP)
    If Not m.IsValidMessage Then Exit Sub
    ProcessReceivedSourceMessage(m)
Case MESSAGETYPES.SOURCE_DESTINATION_MESSAGE
    RaiseEvent LogEvent("MessageSerializer", "Message type is SOURCE_DESTINATION MESSAGE")
    Dim m As New SourceDestinationMessage(b)
m.DeSerialize(_ESP)
    If Not m.IsValidMessage Then Exit Sub
    ProcessReceivedSourceDestinationMessage(m)
Case MESSAGETYPES.UNSECURE_SOURCE_MESSAGE
    RaiseEvent LogEvent("MessageSerializer", "Message type is UNSECURE_SOURCE MESSAGE")
    Dim m As New UnsecureSourceMessage(b)
m.DeSerialize(_ESP)
    If Not m.IsValidMessage Then Exit Sub
    ProcessReceivedUnsecureSourceMessage(m)
Case Else 'Disregard message
    RaiseEvent LogEvent("MessageSerializer", "Disregard message. Invalid format")
End Select
End Sub
Public Overridable Sub ProcessReceivedMessage(ByVal m As Message)
    RaiseEvent OnMessageDeSerialized(m)
    RaiseEvent LogEvent("MessageSerializer", "MESSAGE (seq=" & m.Seq & ") (ack=" & m.Acknowledgement & ")(type=" & m.Type & ") DeSerialized")
End Sub
Public Overridable Sub ProcessReceivedDataMessage(ByVal dm As DataMessage)
    RaiseEvent OnDataMessageDeSerialized(dm)
    RaiseEvent LogEvent("MessageSerializer", "DATA_MESSAGE (seq=" & dm.Seq & ") (ack=" & dm.Acknowledgement & ")(type=" & dm.Type & ")(txt=" & dm.Text & ") DeSerialized")
End Sub
Public Overridable Sub ProcessReceivedSourceMessage(ByVal sm As SourceMessage)
    RaiseEvent OnSourceMessageDeSerialized(sm)
    RaiseEvent LogEvent("MessageSerializer", "SOURCE MESSAGE (seq=" & sm.Seq & ") (ack=" & sm.Acknowledgement & ")(type=" & sm.Type & ") DeSerialized")
End Sub
Public Overridable Sub ProcessReceivedSourceDestinationMessage(ByVal sdm As SourceDestinationMessage)
    RaiseEvent OnSourceDestinationMessageDeSerialized(sdm)
End Sub

Public Overridable Sub ProcessReceivedUnsecureSourceMessage(ByVal usm As SourceMessage)
    RaiseEvent OnUnsecureSourceMessageDeSerialized(usm)
    RaiseEvent LogEvent("MessageSerializer", "UNSECURE_SOURCE_MESSAGE (seq=" & usm.Seq & ") (ack=" & usm.Acknowledgement & ") (type=" & usm.Type & ") (txt=" & usm.Text & ") (src=" & usm.Source.ToString & ") DeSerialized")
End Sub

Public Overridable Sub SendMessage(ByVal Seq As Integer, ByVal Ack As Integer, ByVal MessageType As MESSAGETYPES)
    If Not MessageType And MESSAGETYPES.MESSAGE Then Exit Sub
    Dim m As New Message
    m.Type = MessageType
    m.Seq = Seq
    m.Acknowledgement = Ack
    m.Serialize(Nothing)
    If Not m.IsValidMessage() Then Exit Sub
    RaiseEvent OnMessageSerialized(m.Blob)
    RaiseEvent LogEvent("MessageSerializer", "MESSAGE (seq=" & m.Seq & ") (ack=" & m.Acknowledgement & ") (type=" & m.Type & ") Serialized")
End Sub

Public Overridable Sub SendDataMessage(ByVal Seq As Integer, ByVal Ack As Integer, ByVal Text As String, ByVal MessageType As MESSAGETYPES)
    If Not MessageType And MESSAGETYPES.DATA_MESSAGE Then Exit Sub
    Dim dm As New DataMessage
    dm.Type = MessageType
    dm.Seq = Seq
    dm.Acknowledgement = Ack
    dm.Text = Text
    dm.Serialize(_ESP)
    If Not dm.IsValidMessage() Then Exit Sub
    RaiseEvent OnDataMessageSerialized(dm.Blob)
    RaiseEvent LogEvent("MessageSerializer", "DATA_MESSAGE (seq=" & dm.Seq & ")) (ack=" & dm.Acknowledgement & ") (type=" & dm.Type & ") (txt=" & dm.Text & ") DeSerialized")
End Sub
Public Overridable Sub SendSourceMessage(ByVal Seq As Integer, ByVal Ack As Integer, ByVal Source As Guid, ByVal Text As String, ByVal MessageType As MESSAGETYPES)
If Not MessageType And MESSAGETYPES.SOURCE_MESSAGE Then
Exit Sub
Dim sm As New SourceMessage
sm.Type = MessageType
sm.Source = Source
sm.Seq = Seq
sm.Acknowledgement = Ack
sm.Text = Text
sm.Serialize(_ESP)
If Not sm.IsValidMessage() Then Exit Sub
RaiseEvent OnSourceMessageSerialized(sm.Blob)
End Sub
Public Overridable Sub SendUnsecureSourceMessage(ByVal Seq As Integer, ByVal Ack As Integer, ByVal Source As Guid, ByVal Text As String, ByVal MessageType As MESSAGETYPES)
If Not MessageType And MESSAGETYPES.UNSECURE_BROADCAST_MESSAGE Then Exit Sub
Dim usm As New UnsecureSourceMessage
usm.Type = MessageType
usm.Source = Source
usm.Seq = Seq
usm.Acknowledgement = Ack
usm.Text = Text
usm.Serialize(_ESP)
If Not usm.IsValidMessage() Then Exit Sub
RaiseEvent OnUnsecureSourceMessageSerialized(usm.Blob)
End Sub
Public Overridable Sub SendSourceDestinationMessage(ByVal Seq As Integer, ByVal Ack As Integer, ByVal Source As Guid, ByVal Destination As Guid, ByVal Text As String, ByVal MessageType As MESSAGETYPES)
If Not MessageType And MESSAGETYPES.SOURCE_DESTINATION_MESSAGE Then Exit Sub
Dim sdm As New SourceDestinationMessage

sdm.Type = MESSAGETYPES.PRIVATE_MESSAGE
sdm.Source = Source
sdm.Destination = Destination
sdm.Seq = Seq
sdm.Acknowledgement = Ack
sdm.Text = Text
sdm.Serialize(_ESP)

If Not sdm.IsValidMessage() Then Exit Sub

RaiseEvent OnSourceDestinationMessageSerialized(sdm.Blob)
End Sub
References
