# **MỤC LỤC**

# **DANH SÁCH CÁC HÌNH VẼ**

# **DANH SÁCH CÁC BẢNG** BIỂU

# **MỞ ĐẦU**

* 1. **Lý do chọn đề tài:**

Hiện nay, mạng Lan đã có những tiến bộ vượt bậc và ngày càng phổ biến hơn trong đồi sống sinh hoat. Điều này làm cho nhu cầu liên lạc và trao đổi thông tin thông qua mạng Lan ngày càng lớn hơn. Chính vì vậy, chương trình Chat trên mạng Lan được xây dựng để đáp ứng phần nào những nhu cầu cấp thiết đó.

* 1. **Mục đích của đề tài:**

Xây dựng chương trình Chat hoạt động trong mạng Lan với các chức năng cơ bản như: gởi tin nhắn, tạo một nhóm Chat và lưu thông tin bạn bè.

* 1. **Đối tượng và phạm vi nghiên cứu**

### **Đối tượng nghiên cứu**

Tìm hiểu được cơ chế hoạt động của Socket và Thread trong .NET Framwork từ đó viết ứng dụng Chat trong mạng Lan.

### **Phạm vi nghiên cứu**

Chương trình Chat được xây dựng với khả năng gởi các được văn bản qua lại giữa các user thông qua sự điều khiển của một Server trong mạng Lan.

# **KIẾN THỨC ỨNG DỤNG**

* 1. **Sơ lược về lập trình Socket:**

### **Khái niệm Địa chỉ và cổng (Address & Port)**

* ***Nguyên lý:***
  + Trong một máy có rất nhiều ứng dụng muốn trao đổi với các ứng dụng khác thông qua mạng (ví dụ trên có 2 ứng dụng trong máy A muốn trao đổi với với 2 ứng dụng trên máy B).
  + Mỗi máy tính chỉ có duy nhất một đường truyền dữ liệu (để gửi và nhận).
* ***Vấn đề :*** Rất có thể xảy ra "nhầm lẫn" khi dữ liệu từ máy A gửi đến máy B thì không biết là dữ liệu đó gửi cho ứng dụng nào trên máy B?
* ***Giải quyết:*** Mỗi ứng dụng trên máy B sẽ được gán một số hiệu (mà ta vẫn quen gọi là cổng : Port), số hiệu cổng này từ 1..65535. Khi ứng dụng trên máy A muốn gửi cho ứng dụng nào trên máy B thì chỉ việc điền thêm số hiệu cổng (vào trường RemotePort) vào gói tin cần gửi. Trên máy B, các ứng dụng chỉ việc kiểm tra giá trị cổng trên mỗi gói tin xem có trùng với số hiệu cổng của mình (đã được gán – chính là giá trị Localport) hay không? Nếu bằng thì xử lý, còn trái lại thì không làm gì (vì không phải là của mình).
* ***Như vậy***: Khi cần trao đổi dữ liệu cho nhau thì hai ứng dụng cần phải biết thông tin tối thiểu là địa chỉ (Address) và số hiệu cổng (Port) của ứng dụng kia.

### **Lớp IPAddress**

Trên Internet mỗi một trạm (có thể là máy tính, máy in, thiết bị …) đều có một định danh duy nhất, định danh đó thường được gọi là một địa chỉ (Address). Địa chỉ trên Internet là một tập hợp gồm 4 con số có giá trị từ 0-255 và cách nhau bởi dấu chấm.

Để thể hiện địa chỉ này, người ta có thể viết dưới các dạng sau:

* Tên : Ví dụ May01, Server, ….
* Địa chỉ IP nhưng đặt trong một xâu: "192.168.1.1", "127.0.0.1"
* Đặt trong một mảng 4 byte, mỗi byte chứa một số từ 0-255. Ví dụ để biểu diễn địa chỉ 192.168.1.1 với khai báo “byte[] DiaChi = new byte[4];”, ta có thể viết:
* DiaChi(0) = 192;
* DiaChi(1) = 168;
* DiaChi(2) = 1;
* DiaChi(3) = 1;
* Hoặc cũng có thể là một số (long), có độ dài 4 byte. Ví dụ, với địa chỉ 192.168.1.1 ở trên thì giá trị đó sẽ là: 16885952 (đây là số ở hệ thập phân khi xếp liền 4 byte ở trên lại với nhau **00000001** 00000001 **10101000** 11000000
* Như vậy, để đổi một địa chỉ chuẩn ra dạng số ta chỉ việc tính toán cho từng thành phần. Ví dụ: Đổi địa chỉ 192.168.1.2 ra số, ta tính như sau :

***2 \* 256 ^ 3 + 1\* 256 ^ 2 + 168 \* 256 ^ 1 + 192 \* 256 ^ 0***

Trong .NET, IPAddress là một lớp dùng để mô tả địa chỉ này. Đây là lớp rất cơ bản được sử dụng khi chúng ta thao tác (truyền) vào các lớp như IPEndpoint, UDP, TCP, Socket …

**Bảng 21: Các thành phần của lớp IpAddress**

| **Thành viên Static** | **Mô tả** |
| --- | --- |
| ***Any*** | Cung cấp một địa chỉ IP (thường là 0.0.0.0) để chỉ ra rằng Server phải lắng nghe các hoạt động của Client trên tất cả các Card mạng (sử dụng khi xây dựng Server). Thuộc tính này chỉ đọc. |
| ***Broadcast*** | Cung cấp một địa chỉ IP quảng bá (Broadcast, thường là 255.255.255.255), ở dạng số long. |
| ***Loopback*** | Trả về một địa chỉ IP lặp (IP Loopback, ví dụ 127.0.0.1). |
| ***AddressFamily*** | Trả về họ địa chỉ của địa chỉ IP hiện hành. Nếu địa chỉ ở dạng IPv4 thì kết quả là Internetwork, và InternetworkV6 nếu là địa chỉ IPv6. |
| **Phương thức** | **Mô tả** |
| ***IPAddress(Int64)*** | Tạo địa chỉ IP từ một số long. |
| ***IPAddress(Byte[])*** | Tạo địa chỉ IP từ một mảng Byte. |
| ***GetAddressByte ()*** | Chuyển địa chỉ thành mảng Byte. |
| ***HostToNetworkOrder()*** | Đảo thứ tự Byte của một số cho đúng với thứ tự Byte trong địa chỉ IPAddress. |
| ***IsLoopback()*** | Cho biết địa chỉ có phải là địa chỉ lặp hay không? |

* **Ví dụ 1:** Kiểm tra xem 192.168.1.300 có phải là địa chỉ IP hợp lệ không

| **private** void KiemTra()  {  String Ip1 = "127.0.0.1";  String Ip2 = "999.0.0.1";  MessageBox.Show(IPAddress.TryParse(Ip1, new IPAddress(0)));  MessageBox.Show (IPAddress.TryParse(Ip2, new IPAddress(1)));  } |
| --- |

* **Ví dụ 2:** Chuyển địa chỉ hiện hành ra mảng byte và hiển thị từng thành sphần trong mảng đó

| **private** void KiemTra()  {  IpAddress Ip3 = new IPAddress(16885952);  Byte[] b;  b = Ip3.GetAddressBytes();  MessageBox.Show("Address: " & b(0) &"." & b(1) &"." & b(2) & "." & b(3));  } |
| --- |

### **Lớp IPEndpoint**

Trong mạng, để hai trạm có thể trao đổi thông tin được với nhau thì chúng cần phải biết được địa chỉ (IP) của nhau và số hiệu cổng mà hai bên dùng để trao đổi thông tin. Lớp IPAddress mới chỉ cung cấp cho ta một vế là địa chỉ IP (IPAddress), như vậy vẫn còn thiếu vế thứ hai là số hiệu cổng (Port number). Như vậy, lớp IPEndpoint chính là lớp chứa đựng cả IPAddress và Port number.

Đối tượng IPEndpoint sẽ được dùng sau này để truyền trực tiếp cho các đối tượng UDP, TCP…

**Bảng 22: Các thành viên của lớp IpEndPoint**

| **Phương thức khởi tạo** | **Mô tả** |
| --- | --- |
| ***IPEndPoint(Int64, Int32)*** | Tạo một đối tượng mới của lớp **IPEndPoint,** tham số truyền vào là địa **chỉ IP** (ở dạng số) và **cổng** sẽ dùng để giao tiếp. |
| ***IPEndPoint(IPAddress, Int32)*** | Tạo một đối tượng mới của lớp **IPEndPoint,** Tham số truyền vào là một địa chỉ **IPAddress** và số hiệu cổng dùng để giao tiếp. |
| **Thuộc tính** | **Mô tả** |
| ***Address*** | Trả về hoặc thiết lập địa chỉ IP cho Endpoint (trả về một đối tượng IPAddress). |
| ***AddressFamily*** | Lấy về loại giao thức mà Endpoint này đang sử dụng. |
| ***Port*** | Lấy hoặc gán số hiệu cổng của Endpoint. |
| **Phương thức** | **Mô tả** |
| ***Create()*** | Tạo một Endpoint từ một địa chỉ socket (socket address). |
| ***ToString()*** | Trả về địa chỉ IP và số hiệu cổng theo khuôn dạng địa chỉ: cổng. Ví dụ: “192.168.1.1:8080” |

### **Lớp UDP**

Giao thức UDP (User Datagram Protocol hay User Define Protocol) là một giao thức phi kết nối (connectionless) có nghĩa là một bên có thể gửi dữ liệu cho bên kia mà không cần biết là bên đó đã sẵn sàng hay chưa? (Nói cách khác là không cần thiết lập kết nối giữa hai bên khi tiến hành trao đổi thông tin). Giao thức này không tin cậy bằng giao thức TCP nhưng tốc độ lại nhanh và dễ cài đặt. Ngoài ra, với giao thức UDP ta còn có thể gửi các gói tin quảng bá (Broadcast) cho đồng thời nhiều máy.

Trong .NET, lớp **UDPClient** (nằm trong namesapce **System.Net.Sockets**) đóng gói các chức năng của giao thức UDP.

**Bảng 23: Các thành viên của lớp UDPClient**

| **Phương thức khởi tạo** | **Mô tả** |
| --- | --- |
| ***UdpClient ()*** | Tạo một đối tượng (thể hiện) mới của lớp UDPClient. |
| ***UdpClient (AddressFamily)*** | Tạo một đối tượng (thể hiện) mới của lớp UDPClient. Thuộc một dòng địa chỉ (AddressFamily) được chỉ định. |
| ***UdpClient (Int32)*** | Tạo một **UdpClient** và gắn (bind) một cổng cho nó. |
| ***UdpClient (IPEndPoint)*** | Tạo một **UdpClient** và gắn (bind) một IPEndpoint (gán địa chỉ IP và cổng) cho nó. |
| ***UdpClient(Int32, AddressFamily)*** | Tạo một **UdpClient** và gán số hiệu cổng, AddressFamily |
| ***UdpClient(String, Int32)*** | Tạo một **UdpClient** và thiết lập với một trạm từ xa mặc định. |
| **Phương thức** | **Mô tả** |
| [***BeginReceive***](about:blank)***()*** | Nhận dữ liệu Không đồng bộ từ máy ở xa. |
| [***BeginSend***](about:blank)***()*** | Gửi không đồng bộ dữ liệu tới máy ở xa |
| [***Close***](about:blank)***()*** | Đóng kết nối. |
| [***Connect***](about:blank)***()*** | Thiết lập một Default remote host. |
| [***EndReceive***](about:blank)***()*** | Kết thúc nhận dữ liệu không đồng bộ ở trên |
| [***EndSend***](about:blank)***()*** | Kết thúc việc gửi dữ liệu không đồng bộ ở trên |
| ***Receive (ref IPEndPoint)*** | Nhận dữ liệu (đồng bộ) do máy ở xa gửi**.** (Đồng bộ có nghĩa là các lệnh ngay sau lệnh Receive chỉ được thực thi nếu Receive đã nhận được dữ liệu về . Còn nếu nó chưa nhận được – dù chỉ một chút – thì nó vẫn cứ chờ (blocking)) |
| [***Send***](about:blank)***()*** | Gửi dữ liệu (đồng bộ) cho máy ở xa. |

* **Ví dụ 1**: Tạo một UDPClient gắn vào cổng 10 và Gửi một gói tin "Hello" tới một ứng dụng UDP khác đang chạy trên máy có địa chỉ là "127.0.0.1" và cổng 1000.

| **using** System.Net;  **using** System.Net.Sockets;  **public** class UdpTest  {  const LOCAL\_PORT = 10;  const REMOTE\_PORT = 1000;  *// Tạo một UDP và gắn (Bind) vào cổng 10*  UpdCleint Sender = new UdpClient(LOCAL\_PORT);  privte void Gửi\_Dữ\_Liệu()  {  // Chuyển chuỗi "Hello there !" thành mảng byte để gửi đi  Byte[] msg = System.Text.Encoding.UTF8.GetBytes("Hello there !");  *// Gửi vào cổng 1000 của máy 127.0.0.1*  Sender.Send(msg, msg.Length, "127.0.0.1", REMOTE\_PORT);  }  } |
| --- |

* **Ví dụ 2**: Tạo một UDPClient gắn vào cổng 1000 và nhận dữ liệu từ ứng dụng khác gửi đến.

| **using** System.Net;  **using** System.Text;  **using** System.Net.Sockets;  **namespace Test**  {  **public** class **UdpTest**  const LOCAL\_PORT = 1000;  const REMOTE\_PORT = 10;  **UpdClient** Receiver = new **UdpClient**(LOCAL\_PORT);  **private** void **Nhận\_Dữ\_Liệu**()  {  **IPEndPoint** ep = new  **IPEndPoint**(IPAddress.Parse("127.0.0.1"), 100);  **Byte**[] msg = Receiver.**Receive**(ep);  String str;  str = Encoding.UTF8.GetString(msg);  '*/Chuyển byte -> String*  MessageBox.Show(str);  }  }  } |
| --- |

### **Lớp TCP (TCPClient)**

Mục đích của lớp UDPClient ở trên là dùng cho lập trình với giao thức UDP, với giao thức này thì hai bên không cần phải thiết lập kết nối trước khi gửi do vậy mức độ tin cậy không cao. Để đảm bảo độ tin cậy trong các ứng dụng mạng, người ta còn dùng một giao thức khác, gọi là giao thức có kết nối : TCP (Transport Control Protocol). Trên Internet chủ yếu là dùng loại giao thức này, ví dụ như Telnet, HTTP, SMTP, POP3… Để lập trình theo giao thức TCP, MS.NET cung cấp hai lớp có tên là TCPClient và TCPListener.

**Bảng 24: Các thành phần của lớp TcpClient**

| **Phương thức khởi tạo** | **Mô tả** | |
| --- | --- | --- |
| ***TcpClient()*** | Tạo một đối tượng **TcpClient**. Chưa đặt thông số gì. | |
| ***TcpClient(IPEndPoint)*** | Tạo một **TcpClient** và gắn cho nó một EndPoint cục bộ. (Gán địa chỉ máy cục bộ và số hiệu cổng để sử dụng trao đổi thông tin về sau) | |
| ***TcpClient(String,Int32)*** | Tạo một đối tượng **TcpClient** và kết nối đến một máy có địa chỉ và số hiệu cổng được truyền vào. RemoteHost có thể là địa chỉ IP chuẩn hoặc tên máy. | |
| **Các thuộc tính** | **Mô tả** | |
| [***Available***](about:blank) | Cho biết số byte đã nhận về từ mạng và có sẵn để đọc. | |
| [***Client***](about:blank) | Trả về Socket ứng với TCPClient hiện hành. | |
| [***Connected***](about:blank) | Trạng thái cho biết đã kết nối được đến Server hay chưa? | |
| **Các hàm thành phần** | **Mô tả** | |
| ***Close()*** | Giải phóng đối tượng **TcpClient** nhưng không đóng kết nối. | |
| ***Connect(RemoteHost, RemotePort)*** | Kết nối đến một máy TCP khác có Tên và số hiệu cổng. | |
| ***GetStream()*** | Trả về **NetworkStream** để từ đó giúp ta gửi hay nhận dữ liệu. (Thường làm tham số khi tạo **StreamReader** và StreamWriter để gửi và nhận dữ liệu dưới dạng xâu ký tự) .  Khi đã gắn vào **StreamReader** và **StreamWriter** rồi thì ta có thể gửi và nhận dữ liệu thông qua các phương thức Readline, writeline tương ứng của các lớp này. | |

* Từ các thành viên của lớp TcpClient ở trên ta thấy rằng, việc kết nối và thực hiện gửi nhận rất đơn giản. Theo các trình tự sau:
  + Bước 1: Tạo một đối tượng TcpClient.
  + Bước 2: Kết nối đến máy chủ (Server) dùng phương thức **Connect.**
  + Bước 3: Tạo 2 đối tượng StreamReader (Receive)và StreamWriter (Send) và **"nối"** với GetStream của cpPClient.
  + Bước 4:
* Dùng đối tượng StreamWriter.Writeline/Write vừa tạo ở trên để gửi dữ liệu đi.
* Dùng đối tượng StreamReader.Readline/Read vừa tạo ở trên để đọc dữ liệu về.
  + Bước 5: Đóng kết nối.
* Nếu muốn gửi/nhận dữ liệu ở mức byte (nhị phân) thì dùng NetworkStream. (truyền GetStream cho NetworkStream).

### **Lớp TcpListener**

TCPListerner là một lớp cho phép người lập trình có thể xây dựng các ứng dụng Server (Ví dụ như SMTP Server, FTP Server, DNS Server, POP3 Server hay server tự định nghĩa ….). Ứng dụng server khác với ứng dụng Client ở chỗ nó luôn luôn thực hiện lắng nghe và chấp nhận các kết nối đến từ Client.

**Bảng 25: Các thành phần của lớp TcpListener**

| **Phương thức khởi tạo** | **Mô tả** |
| --- | --- |
| ***TcpListener ( Int32)*** | Tạo một **TcpListener** và lắng nghe tại cổng chỉ định. |
| ***TcpListener (IPEndPoint)*** | Tạo một **TcpListener** với giá trị Endpoint truyền vào. |
| ***TcpListener(IPAddress,Int32)*** | Tạo một **TcpListener** và lắng nghe các kết nối đến tại địa chỉ IP và cổng chỉ định. |
| **Phương thức** | **Mô tả** |
| ***AcceptSocket( )*** | Chấp nhận một yêu cầu kết nối đang chờ. |
| ***AcceptTcpClient()*** | Chấp nhận một yêu cầu kết nối đang chờ. (Ứng dụng sẽ dừng tại lệnh này cho đến khi nào có một kết nối đến – “Blocking”). |
| ***Pending()*** | Cho biết liệu có kết nối nào đang chờ đợi không |
| ***Start()*** | Bắt đầu lắng nghe các yêu cầu kết nối. |
| ***Stop()*** | Dừng việc nghe. |

* **Ví dụ**: Tạo một server trong đó, khi có một client kết nối đến thì server chuyển xâu đó thành chữ HOA và gửi trả lại cho Client.

| **using** System.Net.Sockets;  **using** System.Net;  **using** System.IO;  **using** System.Net.Sockets;  **using** System.Threading;  **namespace** **TcpListenerTest**  {  **public** class **frmServer**  {  **TcpListener** TCPServer = **new TcpListener**(21);  **bool** Thoat = **false**;  **TcpClient**[] Clients = **new TcpClient**[101];  **int** CurrClient = 0;  **public** void **Xu\_Ly\_Ket\_Noi**()  {  **int** LastClient = CurrClient - 1;  **TcpClient** Con = **Clients**(LastClient);  StreamReader Doc = **new StreamReader**(Con.**GetStream**());  StreamWriter Ghi = **new StreamWriter**(Con.**GetStream**());  string S = **null**;  **while** (Thoat == **false**)  {  Application.**DoEvents**();  **if** (Doc.EndOfStream == **false**)  {  S = Doc.**ReadLine**();  //// Xử lý tại đây  S = S.**ToUpper**();  Ghi.**WriteLine**(S);  ////Gởi lại cho Client...  Ghi.**Flush**();  }  }  }  **public** void **Nghe\_Ket\_Noi**()  {  **while** (Thoat == **false**) {  **Clients**(CurrClient) = TCPServer.**AcceptTcpClient**();  CurrClient += 1;  Thread Th = **new Thread**(Xu\_Ly\_Ket\_Noi);  Th.**Start**();  }  }  **private** void **frmClose**(object s, FormClosingEventArgs e)  {  Thoat = **true**;  }  **private** void **Form1\_Load**(object s, EventArgs e)  {  TCPServer.**Start**();  **Nghe\_Ket\_Noi**  }  **public frmServer**()  {  Load += Form1\_Load;  FormClosing += frmClose;  }  }  } |
| --- |

* 1. **Sơ lược về lập trình đa luồng:**

### **Khái niệm Luồng (Thread)**

Mộtluồng (**Thread)** là một chuỗi liên tiếp những sự thực thi trong chương trình. Trong một chương trình C#, việc thực thi bắt đầu bằng phương thức main() và tiếp tục cho đến khi kết thúc hàm main(). Cấu trúc này rất hay cho những chương trình có một chuỗi xác định những nhiệm vụ liên tiếp. Nhưng thường thì một chương trình cần làm nhiều công việc hơn vào cùng một lúc. Ví dụ trong Internet Explorer khi ta đang tải một trang web thì ta nhấn nút back hay một link nào đó, để làm việc này Internet Explorer sẽ phải làm ít nhất là 3 việc:

* Lấy dữ liệu được trả về từ Internet cùng với các tập tin đi kèm.
* Thể hiện trang Web.
* Xem người dùng có nhập để làm thứ gì khác không.

Để đơn giản vấn đề này ta giả sử Internet Explorer chỉ làm hai công việc:

* Trình bày trang Web.
* Xem người dùng có nhập gì không.

Để thực hành việc này ta sẽ viết một phương thức dùng để lấy và thể hiện trang Web. Giả sử rằng việc trình bày trang Web mất nhiều thời gian (do phải thi hành các đoạn javascript hay các hiệu ứng nào đó …). Vì vậy sau một khoảng thời gian ngắn khoảng 1/12 giây, phương thức sẽ kiểm tra xem người dùng có nhập gì không. Nếu có thì nó sẽ được xử lý, nếu không thì việc trình bày trang sẽ được tiếp tục. Và sau 1/12 giây việc kiểm tra sẽ được lặp lại. Tuy nhiên viết phương thức này thì rất phức tạp do đó ta sẽ dùng kiến trúc event trong Window nghĩa là khi việc nhập xảy ra hệ thống sẽ thông báo cho ứng dụng bằng cách đưa ra một event. Ta sẽ cập nhật phương thức để cho phép dùng các event:

* Ta sẽ viết một bộ xử lí event để đáp ứng đối với việc nhập của người dùng.
* Ta sẽ viết một phương thức để lấy và trình bày dữ liệu. Phương thức này được thực thi khi ta không làm bất cứ điều gì khác.

Ta hãy xem cách phương thức lấy và trình bày trang web làm việc: đầu tiên nó sẽ tự định thời gian. Trong khi nó đang chạy, máy tính không thể đáp ứng việc nhập của người dùng . Do đó nó phải chú ý đến việc định thời gian để gọi phương thức kiểm tra việc nhập của người dùng, nghĩa là phương thức vừa chạy vừa quan sát thời gian. Bên cạnh đó nó còn phải quan tâm đến việc lưu trữ trạng thái trước khi nó gọi phương thức khác để sau khi phương thức khác thực hiện xong nó sẽ trả về đúng chỗ nó đã dừng. Vào thời Window 3.1 đây thực sự là những gì phải làm để xử lí tình huống này. Tuy nhiên ở NT3.1 và sau đó là Windows 95 trở đi đã có việc xử lí đa luồng điều này làm việc giải quyết vấn đề tiện lợi hơn. Dưới đây chúng ta sẽ tìm hiểu một vài lớp cơ bản trong ngôn ngữ lập trình C# và vấn đề đồng bộ hóa (Synchronization) trong lập trình đa luồng.

### **Khảo sát namespace System.Threading**

Namespace System.Threading cung cấp một số kiểu dữ liệu cho phép bạn thực hiện lập trình đa luồng. Ngoài việc cung cấp những kiểu dữ liệu tượng trưng cho một luồng cụ thể nào đó, namespace này còn định nghĩa những lớp có thể quản lý một collection các luồng (ThreadPool), một lớp Timer đơn giản (không dựa vào GUI) và các lớp cung cấp truy cập được đồng bộ vào dữ liệu được chia sẻ sử dụng.

**Bảng 26: Một số lớp của namespace System.Threading**

| **Các lớp thành viên** | **Mô tả** |
| --- | --- |
| ***Interlocked*** | Lớp này dùng cung cấp truy cập đồng bộ hóa vào dữ liệu được chia sẽ sử dụng (shared data). |
| ***Moniter*** | Lớp này cung cấp việc đồng bộ hóa các đối tượng luồng sử dụng khóa chốt (lock) và tín hiệu chờ (wait signal). |
| ***Mutex*** | Lớp này cung cấp việc đồng bộ hóa sơ đẳng có thể được dùng đối với inter process synchronization. |
| ***Thread*** | Lớp này tượng trưng cho một luồng được thi hành trong lòng Common Language Runtime. Sử dụng lớp này bạn có khả năng bổ sung những luồng khác trong cùng AppDomain. |
| ***ThreadPool*** | Lớp này quản lý những luồng có liên hệ với nhau trong cùng một Process nào đó. |
| ***Timer*** | Cho biết một delegate có thể được triệu gọi vào một lúc được khai báo nào đó. Tác vụ wait được thi hành bởi luồng trong thread pool. |
| ***WaitHandle*** | Lớp này tượng trưng cho tất cả các đối tượng đồng bộ hóa (cho phép multiple wait) vào lúc chạy. |
| ***ThreadStart*** | Lớp này là một delegate chỉ về hàm hành sự nào đó phải được thi hành đầu tiên khi một luồng bắt đầu. |
| ***TimerCallBack*** | Delegate đối với Timer. |
| ***WaitCallBack*** | Lớp này là một delegate định nghĩa hàm hành sự kêu gọi lại (callback) đối với ThreadPool user work item. |

#### **Lớp Thread**

Lớp sơ đẳng nhất trong tất cả các lớp thuộc Namespace System.Threading là lớp Thread. Lớp này tượng trưng cho một vỏ bọc hướng đối tượng bao quanh một lộ trình thi hành trong lòng một AppDomain nào đó. Lớp này định nghĩa một số hàm thực thi (cả static lẫn shared) cho phép bạn tạo mới những luồng từ luồng hiện hành, cũng như cho Sleep, Stop hay Kill một luồng nào đó.

**Bảng 27: Các thành phần static của lớp Thread**

| **Các thành phần Static** | **Mô tả** |
| --- | --- |
| ***CurrentThread*** | Thuộc tính read-only này trả về một quy chiếu về luồng hiện đang chạy. |
| ***GetData()*** | Đi lấy vị trí từ slot được khai báo trên luồng hiện hành đối với domain hiện hành trong luồng. |
| ***SetData()*** | Cho đặt để trị lên slot được khai báo trên luồng hiện hành đối với domain hiện hành trong luồng |
| ***GetDomain()***  ***GetDomainID()*** | Đi lấy một qui chiếu về AppDomain hiện hành (hoặc mã nhận diện ID của domain này) mà luồng hiện đang chạy trên đó. |
| ***Sleep()*** | Cho ngưng luồng hiện hành trong một thời gian nhất định được khai báo. |

Ngoài ra lớp **Thread** cũng hổ trợ các thành viên cấp đối tượng.

**Bảng 28: Các thành viên cấp đối tượng của lớp Thread**

| **Các lớp thành viên** | **Mô tả** |
| --- | --- |
| ***IsAlive*** | Thuộc tính này trả về một trị boolean cho biết liệu xem luồng đã khởi đông hay chưa. |
| ***IsBackground*** | Đi lấy hoặc đặt để giá trị cho biết liệu xem luồng là một luồng nền hay không. |
| ***Name*** | Thuộc tính này cho phép bạn thiết lập một tên văn bản mang tính thân thiện đối với luồng. |
| ***Priority*** | Đi lấy hoặc đặt để ưu tiên của một luồng. Có thể được gán một trị lấy từ enumeration **ThreadPriority** (chẳng hạn **Normal, Lowest, Highest, BelowNormal, AboveNormal**). |
| ***ThreadState*** | Đi lấy hoặc đặt để tình trạng của luồng. Có thế được gán từ enumeration **ThreadState** (chẳng hạn **Unstarted, Running, WaitSleepJoin, Suspended, SuspendRequested, AbortRequested, Stopped**). |
| ***Interrup()*** | Cho ngưng chạy luồng hiện hành. |
| ***Join()*** | Yêu cầu luồng chờ đối với luồng bị ngưng chạy. |
| ***Resume()*** | Tiếp tục lại đối với một luồng bị ngưng chạy. |
| ***Start()*** | Cho bắt đầu thi hành luồng được khai báo bởi delegate ThreadStart. |
| ***Suspend()*** | Cho ngưng chạy một luồng. Nếu luồng đã bị ngưng rồi, một triệu gọi hàm **Suspend()** sẽ không có tác dụng. |

#### **Thao tác với luồng**

Luồng được thao tác bằng cách dùng lớp **Thread** nằm trong Namespace System.Threading. Một thể hiện của luồng đại diện cho một luồng. Ta có thể tạo các luồng khác bằng cách khởi tạo một đối tượng Thread.

Giả sử rằng ta đang viết 1 trình biên tập hình ảnh đồ hoạ, và người dùng yêu cầu thay đổi độ sâu của màu trong ảnh. Ta bắt đầu khởi tạo một đối tượng luồng như sau:

// entryPoint được khai báo trước là 1 delegate kiểu ThreadStart

Thread depthChangeThread = new Thread(entryPoint);

Đoạn mã trên biểu diễn một hàm khởi tạo của **Thread** với một thông số chỉ định điểm nhập của một luồng. Đó là phương thức nơi luồng bắt đầu thi hành. Trong tình huống này ta dùng thông số là delegate, môt delegate đã được định nghĩa trong System.Threading gọi là **ThreadStart**, chữ kí của nó như sau:

public delegate void ThreadStart();

Thông số ta truyền cho hàm dựng phải là 1 delegate kiểu này. Ta bắt đầu luồng bằng cách gọi phương thức Thread.Start() , giả sử rằng ta có phương thức ChangeColorDepth():

void ChangeColorDepth()

{

// xử lí để thay đổi màu

}

Sắp xếp lại ta có đoạn mã sau :

ThreadStart entryPoint = new ThreadStart(ChangeColorDepth);

Thread depthChangeThread = new Thread(entryPoint);

depthChangeThread.Name = “Depth Change Thread”;

depthChangeThread.Start();

Sau điểm này, cả hai luồng sẽ chạy đồng bộ với nhau.

Trong đoạn mã này ta đăng kí tên cho luồng bằng cách dùng thuộc tính Thread.Name. Không cần thiết làm điều này nhưng nó có thể hữu ích.

Lưu ý rằng bởi vì điểm đột nhập vào luồng (trong ví dụ này là ChangeColorDepth() ) không thể lấy bất kì thông số nào. Ta sẽ phải tìm một cách nào đó để truyền thông số cho phương thức nếu cần. Cách tốt nhất là dùng các trường thành viên của lớp mà phương thức này là thành viên. Cũng vậy phương thức không thể trả về bất cứ thứ gì .

Mỗi lần ta bắt đầu một luồng khác, ta cũng có thể đình chỉ, hồi phục hay bỏ qua nó. Đình chỉ nghĩa là cho luồng đó ngủ (sleep) - nghĩa là không chạy trong 1 khoảng thời gian. Sau đó nó thể đưọc phục hồi, nghĩa là trả nó về thời diểm mà nó bị định chỉ. Nếu luồng đưọc bỏ, nó dừng chạy. Window sẽ huỷ tất cả dữ liệu mà liên hệ đến luồng đó, để luồng không thể được bắt đầu lại. Tiếp tục ví dụ trên, ta giả sử vì lí do nào đó luồng giao diện người dùng trình bày một hộp thoại cho người dùng cơ hội để đình chỉ tạm thời sự đổi tiến trình. Ta sẽ soạn mã đáp ứng trong luồng main :

depthChangeThread.Suspend();

Và nếu người dùng được yêu cầu cho tiến trình được phục hồi:

depthChangeThread.Resume();

Cuối cùng nếu người dùng muốn huỷ luồng :

depthChangeThread.Abort();

Phương thức Suspend() có thể không làm cho luồng bị định chỉ tức thời mà có thể là sau một vài lệnh, điều này là để luồng được đình chỉ an toàn. Đối với phương thức Abort() nó làm việc bằng cách tung ra biệt lệ ThreadAbortException. ThreadAbortException là một lớp biệt lệ đặc biệt mà không bao giờ được xử lí. Nếu luồng đó thực thi mã bên trong khối try, bất kì khối finally sẽ được thực thi trước khi luồng bị huỷ. Sau khi huỷ luồng ta có thể muốn đợi cho đến khi luồng thực sự bị huỷ trước khi tiếp tục luồng khác ta có thể đợi bằng cách dùng phương thức join() :

depthChangeThread.Abort();

depthChangeThread.Join();

Join() cũng có một số overload khác chỉ định thời gian đợi. Nếu hết thời gian này việc thi hành sẽ được tiếp tục. Nếu một luồng chính muốn thi hành một vài hành động trên nó, nó cần một tham chiếu đến đối tượng luồng mà đại diện cho luồng riêng. Nó có thể lấy một tham chiếu sử dụng thuộc tính static -**CurrentThread**- của lớp Thread:

Thread myOwnThread = Thread.CurrentThread;

Có hai cách khác nhau mà ta có thể thao tác lớp Thread:

* Ta có thể khởi tạo 1 đối tượng luồng , mà sẽ đại diện cho luồng đang chạy và các thành viên thể hiện của nó áp dụng đến luồng đang chạy
* Ta có thể gọi 1 số phương thức static . những phương thức này sẽ áp dụng đến luồng mà ta thực sự đang gọi phương thức từ nó.  
  một phương thức static mà ta muốn gọi là Sleep(), đơn giản đặt luồng đang chạy ngủ một khoảng thời gian, sau đó nó sẽ tiếp tục.

### **Đồng bộ hóa (Synchronization) trong lập trình đa luồng:**

#### **Đồng bộ hóa**

Đôi khim có thể bạn muốn điều khiển việc truy cập vào một nguồn lực, chẳng hạn các thuộc tính hoặc các hàm của một đối tượng, làm thế nào chỉ một mạch trình được phép thay đổi hoặc sử dụng nguồn lực đó mà thôi. Việc đồng bộ hóa được thể hiện thông qua một cái khóa được thiết lập trên đối tượng, ngăn không cho luồng nào đó truy cập khi mạch trình đi trước chưa xong công việc.

Trong phần này, ta sẽ là quen với cơ chế đồng bộ hóa mà Common Language Runtime cung cấp: lệnh **lock**. Nhưng trước tiên, ta cần mô phỏng một nguồn lực được chia sẽ sử dụng bằng cách sử dụng một biến số nguyên đơn giản: **counter**.

Để bắt đầu, ta khai báo biến thành viên và khởi gán về zero:

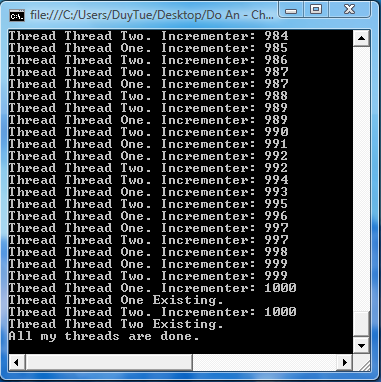
int counter = 0;

Bài toán được đặt ra ở đây như sau: luồng thứ nhất sẽ đọc trị counter (0) rồi gán giá trị này cho biến trung gian (temp). Tiếp đó tăng trị của temp rồi Sleep một khoảng thời gian. Luồng thứ nhất xong việc thì gán trị của temp trả về cho counter và cho hiển thị trị này. Trong khi nó làm công việc, thì luồng thứ hai cũng thực hiện một công việc giống như vậy. Ta cho việc này lập này khoảng 1000 lần. Kết quả mà ta chờ đợi là hai luồng trên đếm lần lượt tăng biến counter lên 1 và in ra kết quả 1, 2, 3, 4 … tuy nhiên ta sẽ xét đoạn chương trình dưới đây và thấy rằng kết quả hoàn toàn khác với những gì mà chúng ta mong đợi.

Đoạn mã của chương trình như sau:

| **using** System;  **using** System.Threading;  **namespace** TestThread  {  **public** class **Tester**  {  **private** int counter = 0;  **static** void **Main**(string[] args)  {  Tester t = new Tester();  t.DoTest();  Console.ReadLine();  }  **public** void **DoTest**()  {  Thread t1 = new Thread(new ThreadStart(Incrementer));  t1.IsBackground = true;  t1.Name = "Thread One";  t1.Start();  Console.WriteLine("Start thread {0}", t1.Name);  Thread t2 = new Thread(new ThreadStart(Incrementer));  t2.IsBackground = true;  t2.Name = "Thread Two";  t2.Start();  Console.WriteLine("Start thread {0}", t2.Name);  t1.Join();  t2.Join();  Console.WriteLine("All my threads are done.");  }  **public** void **Incrementer()**  {  **try**  {  **while** (counter < 1000)  {  int temp = counter;  temp++;  Thread.Sleep(1);  counter = temp;  Console.WriteLine("Thread {0}. Incrementer: {1}",  Thread.CurrentThread.Name, counter);  }  }  **catch** (ThreadInterruptedException)  {  Console.WriteLine("Thread {0} interrupted! Cleaning up...", Thread.CurrentThread.Name);  }  **finally**  {  Console.WriteLine("Thread {0} Existing.",  Thread.CurrentThread.Name);  }  }  }  } |
| --- |

Kết quả đạt được là:



**Hình 21: Kết quả chương trình không sử dụng đồng bộ hóa**

Do đó ta cần phải đồng bộ hóa việc truy cập đối tượng counter.

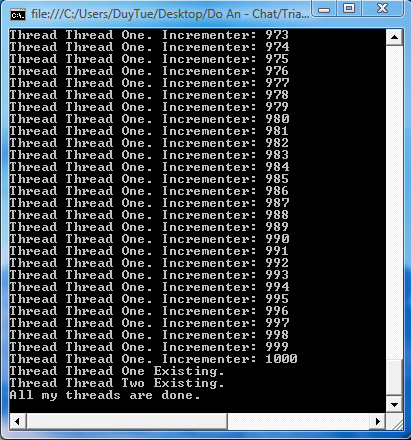
C# cung cấp đối tượng **Lock** để thưc hiện công việc đồng bộ hóa này. Một lock sẽ đánh dấu một critical section trên đoạn mã đồng thời cung cấp việc đồng bộ hóa đối với đối tượng được chỉ định khi lock có hiệu lực. Cú pháp sử dụng một Lock yêu cầu khóa chặt một đối tượng rồi thi hành một câu lệnh hoặc một khối lệnh rồi sẽ mở khóa ở cuối câu hoặc khối lệnh đó. C# cung cấp hổ trợ trực tiếp khóa chặt thông qua từ chốt lock. Ta sẽ tra qua theo một đối tượng qui chiếu và theo sau từ chốt là một khối lệnh

*lock(expression) statement-block*

Trong ví dụ trên, để có được kết quả như mong muốn, ta sẽ sửa hàm **Incrementer** lại như sau:

| try  {  lock (this)  {  while (counter < 1000)  {  int temp = counter;  temp++;  Thread.Sleep(1);  counter = temp;  Console.WriteLine("Thread {0}. Incrementer: {1}",  Thread.CurrentThread.Name, counter);  }  }  }  // Các khối catch và finally không thay đổi |
| --- |

Kết quả thu được sẽ là:



**Hình 22: Kết quả chương trình sử dụng đồng bộ hóa**

Việc đồng bộ các luồng là quan trọng trong các ứng dụng đa luồng. Tuy nhiên có một số lỗi tinh vi và khó kiểm soát có thể xuất hiện cụ thể là **deadlock** và **race condition.**

#### **Deadlock**

#### Deadlock là một lỗi mà có thể xuất hiện khi hai luồng cần truy nhập vào các tài nguyên bị khoá lẫn nhau. Giả sử một luồng đang chạy theo đoạn mã sau, trong đó **A**, **B** là hai đối tượng tham chiếu mà cả hai luồng cần truy nhập :

| lock (A)  {  // do something  lock (B)  {  // do something  }  } |
| --- |

Vào cùng lúc đó 1 luồng khác đang chạy :

| lock (B)  {  // do something  lock (A)  {  // do something  }  } |
| --- |

Có thể xảy ra biến cố sau: luồng đầu tiên yêu cầu một lock trên **A**, trong khi vào cùng thời điểm đó luồng thứ hai yêu cầu lock trên **B**. Một khoảng thời gian ngắn sau, luồng **A** gặp câu lệnh **lock(B)**, và ngay lập tức bước vào trạng thái ngủ, đợi cho lock trên **B** được giải phóng. Và tương tự sau đó, luồng thứ hai gặp câu lệnh **lock(A)** và cũng rơi vào trạng thái ngủ chờ cho đến khi lock trên A được giải phóng . Không may, lock trên **A** sẽ không bao giờ được giải phóng bởi vì luồng đầu tiên mà đã lock trên **A** đang ngủ và không thức dậy cho đến khi lock trên **B** được giải phóng điều này cũng không thể xảy ra cho đến khi nào luồng thứ hai thức dậy. Kết quả là deadlock. Cả hai luồng đều không làm gì cả, đợi lẫn nhau để giải phóng lock. Loại lỗi này làm toàn ứng dụng bị treo, ta phải dùng Task Manager để hủy nó.

Deadlock có thể được tránh nếu cả hai luồng yêu cầu lock trên đối tượng theo cùng thứ tự . Trong ví dụ trên nếu luồng thứ hai yêu cầu lock cùng thứ tự với luồng đầu, **A** đầu tiên rồi tới b thì những luồng mà lock trên a đầu sẽ hoàn thành nhiệm vụ của nó sau đó các luồng khác sẽ bắt đầu.

#### **Race condition**

**Race condition** là cái cái gì đó tinh vi hơn deadlock. Nó hiếm khi nào dừng việc thực thi của tiến trình , nhưng nó có thể dẫn đến việc dữ liệu bị lỗi. Nói chung nó xuất hiện khi vài luồng cố gắng truy nhập vào cùng một dữ liệu và không quan tâm đến các luồng khác làm gì để hiểu ta xem ví dụ sau :

Giả sử ta có một mảng các đối tượng, mỗi phần tử cần được xử lí bằng một cách nào đó, và ta có một số luồng giữa chúng làm tiến trình này. Ta có thể có một đối tuợng gọi là ArrayController chứa mảng đối tượng và một số int chỉ định số phẩn tử được xử lí .tacó phương thức:

int GetObject(int index)

{

// trả về đối tượng với chỉ mục được cho

}

Và thuộc tính read/write

int ObjectsProcessed

{

// chỉ định bao nhiêu đối tượng được xử lí

}

Bây giờ mỗi luồng mà dùng để xử lí các đối tượng có thể thi hành đoạn mã sau:

lock(ArrayController)  
{

int nextIndex = ArrayController.ObjectsProcessed;

Console.WriteLine(”Object to be processed next is ” + NextIndex);

++ArrayController.ObjectsProcessed;

object next = ArrayController.GetObject();

}

ProcessObject(next);

Nếu ta muốn tài nguyên không bị giữ quá lâu , ta có thể không giữ lock trên ArrayController trong khi ta đang trình bày thông điệp người dùng . Do đó ta viết lại đoạn mã trên:

lock(ArrayController)  
{

int nextIndex = ArrayController.ObjectsProcessed;

}

Console.WriteLine(”Object to be processed next is ” + nextIndex);

lock(ArrayController)

{

++ArrayController.ObjectsProcessed;

object next = ArrayController.GetObject();

}

ProcessObject(next);

Ta có thể gặp một vấn đề. Nếu một luồng lấy lấy đối tưọng (đối tượng thứ 11 trong mảng) và đi tới trình bày thông điệp nói về việc xử lí đối tượng này. Trong khi đó luồng thứ hai cũng bắt đầu thi hành cũng đoạn mã gọi ObjectProcessed, và quyết định đối tượng xử lí kế tiếp là đối tượng thứ 11, bởi vì luồng đầu tiên vẫn chưa được cập nhật.

ArrayController.ObjectsProcessed trong khi luồng thứ hai đang viết đến màn hình rằng bây giờ nó sẽ xử lí đối tượng thứ 11, luồng đầu tiên yêu cầu một lock khác trên ArrayController và bên trong lock này tăng ObjectsProcessed. Không may, nó quá trễ. Cả hai luồng đều đang xử lí cùng một đối tượng và loại tình huống này ta gọi là Race Condition.

# **PHÂN TÍCH THIẾT KẾ CHƯƠNG TRÌNH**

* 1. **Phân tích**

### **Phân tích nhu cầu thực tiễn:**

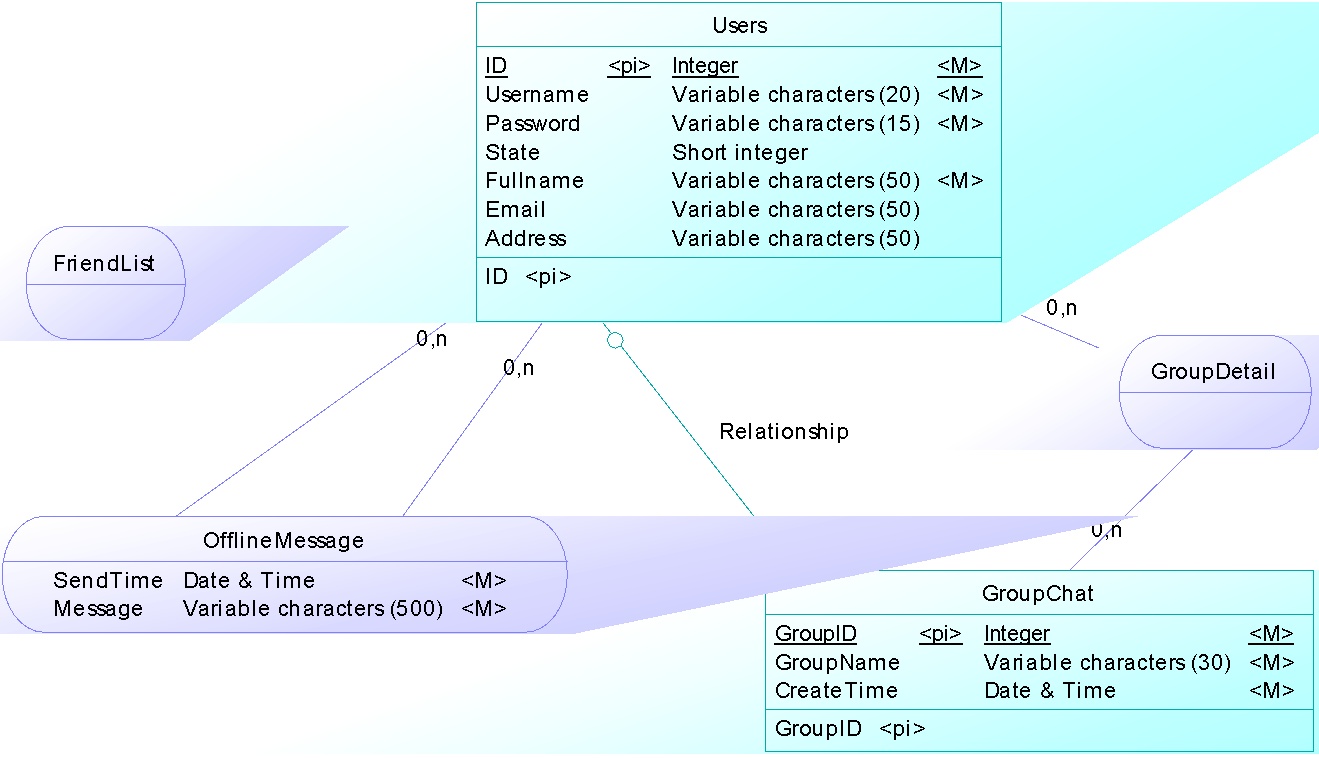
Hiện nay, mạng Lan phát triển rất mạnh cả trong trường học, các cơ quan tổ chức và ở cả các hộ gia đình. Chính điều đó kéo theo nhu cầu liên lạc trao đổi thông tin trong mạng Lan cũng phát triển theo. Chính vì vậy, một chương trình Chat phục vụ cho nhu cầu liên lạc, trao đổi thông tin trong mạng Lan là rất cần thiết.

### **Yêu cầu đề ra:**

Yêu cầu đặt ra là xây dựng chương trình Chat hoạt động trong mạng Lan sử dụng Socket và Multithreading đòi hỏi các chức năng nghiệp vụ sau:

* Chat giữa hai người với nhau: Hai người gởi thông điệp qua lại cho nhau.
* Chat giữa một nhóm người: Một người đứng ra tạo một nhóm Chat và mời các thành viên khác tham gia thảo luận.
* Một User có khả năng thêm và xóa một người vào Friend List của mình để có thể liên lạc một cách dể dàng.

### **Mô hình dữ liệu ở mức quan niệm:**

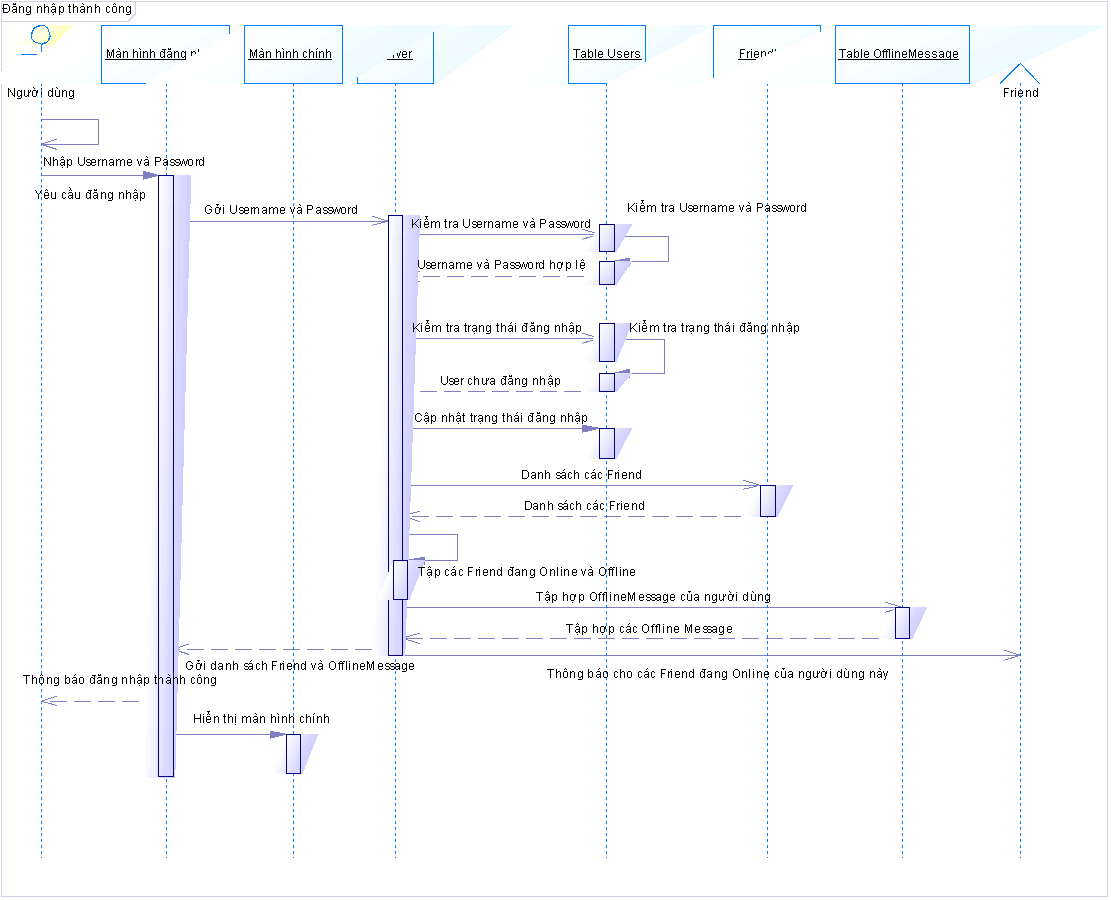


**Hình 31: Mô hình dữ liệu ở mức quan niệm**

### **Phân tích các thành phần xữ lý:**

#### **Các xử lý liên quan đến đăng nhập**

* Xử lý đăng nhập:
  + Mô hình xử lý:

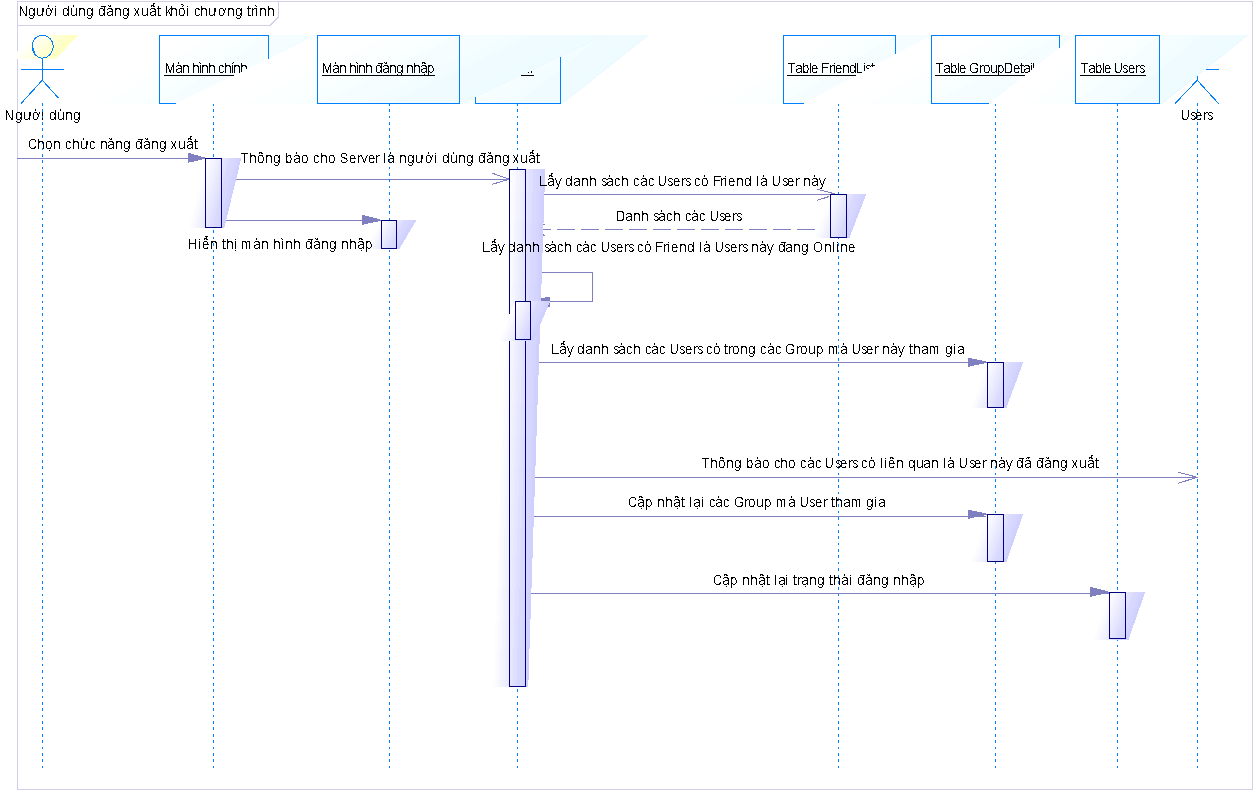


**Hình 32: Mô hình xử lý đăng nhập**

* + Mô tả:

Khi một User yêu cầu đăng nhập hệ thống, Client sẽ gởi Username và Password cho Server. Server sẽ kiểm tra Username và Password có hợp lệ hay không. Sau đó, Server sẽ kiểm tra Username này đã đăng nhập chưa. Nếu đăng nhập thành công, Server sẽ lấy danh sách các Friend đang Offline và Online của User kèm theo danh sách các tin nhắn Offline (nếu có) và gởi cho User và cập nhật lại trạng thái đăng nhập của User. Đồng thời cũng gởi thông báo đến các Users khác có Friend là User này mới Online.

* Xử lý đăng xuất:
  + Mô hình xử lý:



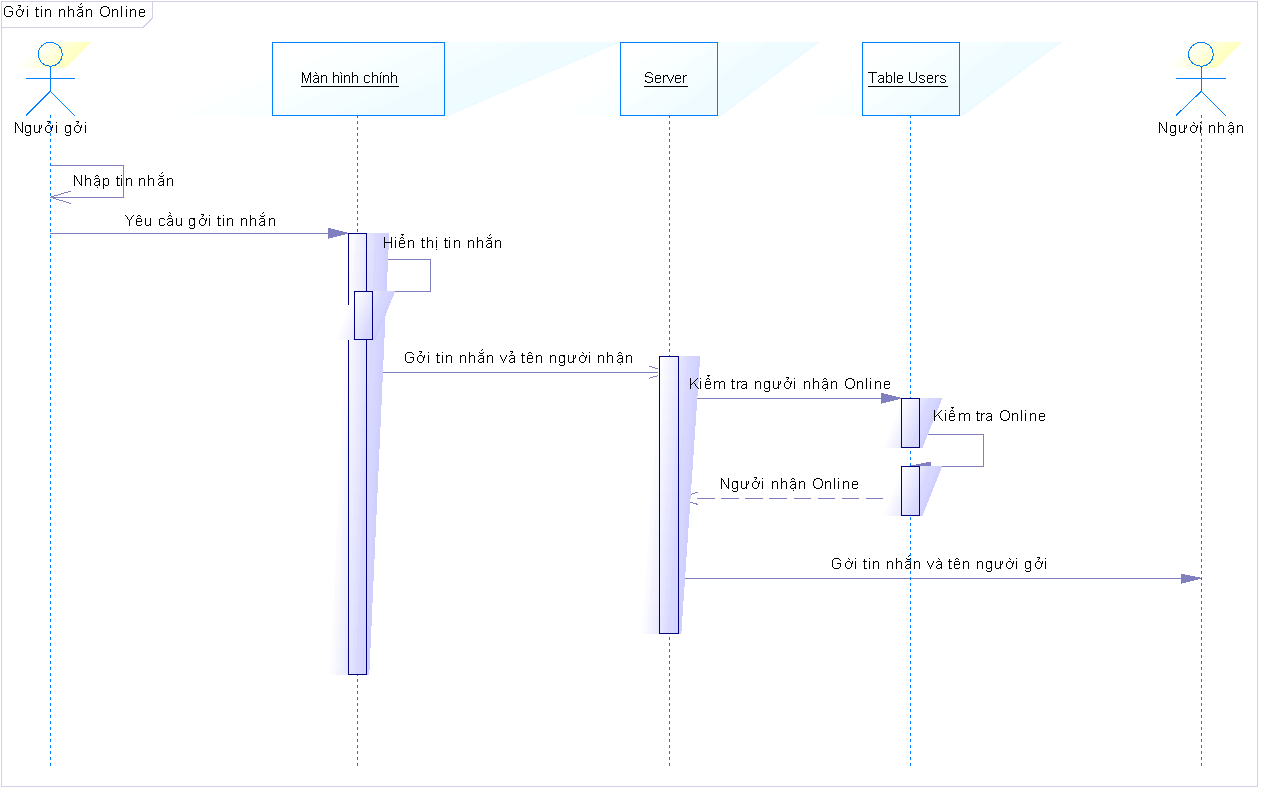
**Hình 33: Mô hình xử lý đăng xuất**

* + Mô tả:

Khi một User yêu cầu đăng xuất lại hệ thống, Client sẽ hiển thị lại màn hình đăng nhập và đồng thời gởi thông báo đang xuất đến Server. Server sẽ gởi thông báo đăng xuất tới các Users có Friend là User này. Bên cạnh đó, Server kiểm tra tất cả các Groups mà User này đang tham gia. Server sẽ gởi thông báo hủy đến các Group mà User này là người khởi tạo và sẽ gởi thông báo đăng xuất đến các Group mà User này chỉ tham gia với tư cách là thành viên. Cuối cùng, Server sẽ cập nhật lại trạng thái đăng nhập của User.

#### **Các xử lý đối với việc gởi tin nhắn**

* Gởi tin nhắn Online:
  + Mô hình xử lý:

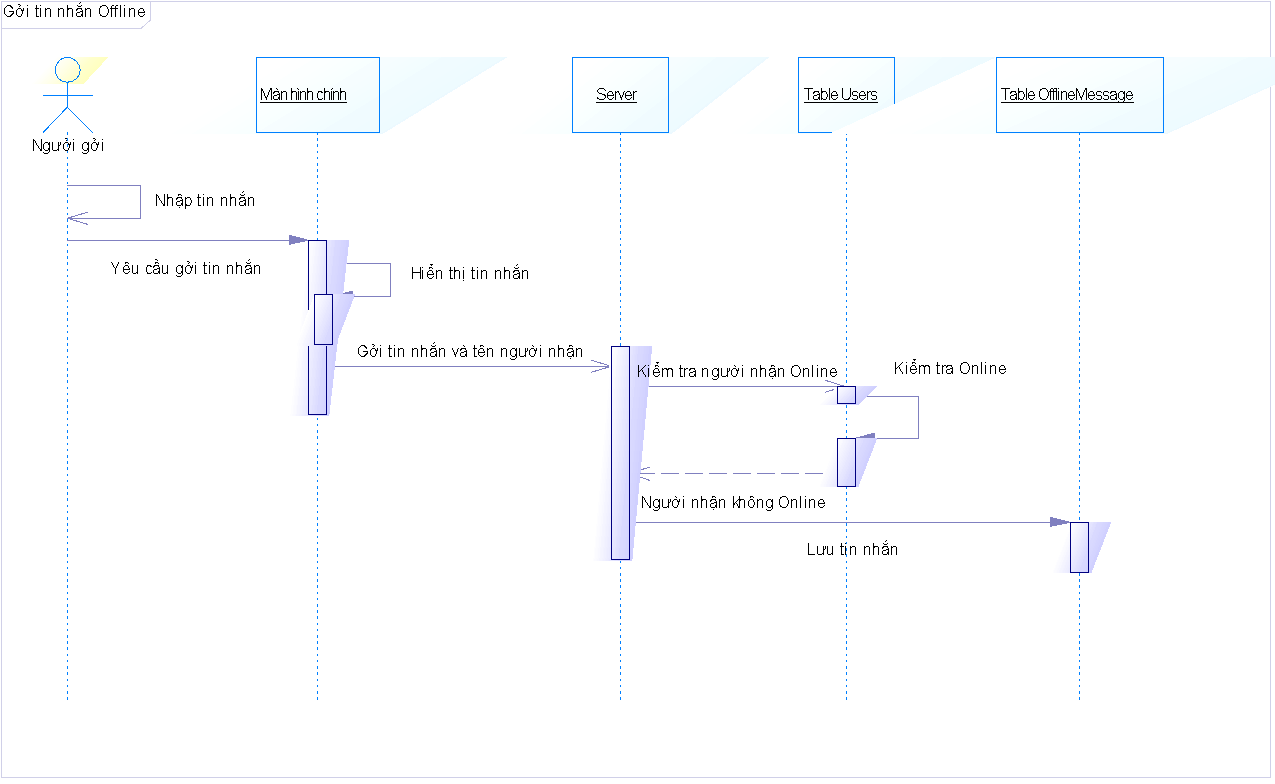


**Hình 34: Mô hình xử lý gởi tin nhắn Online**

* + Mô tả:

Khi người dùng nhập tin nhắn và nhấn phím Enter (hoặc button Send), Client sẽ hiển thị tin nhắn trên Tab Chat đồng thời gởi tin nhắn và tên người nhận lên cho Server. Server sẽ kiểm tra xem người nhận có phải đang Online hay không. Nếu người nhận đang Online, Server sẽ gởi tin nhắn và tên người gởi. Khi nhận được tin nhắn, Client sẽ hiển thị tin nhắn thông qua một Tab Chat (Tab này sẽ được mở nếu nó chưa có trước đó).

* Gởi tin nhắn Offline:
  + Mô hình xử lý:



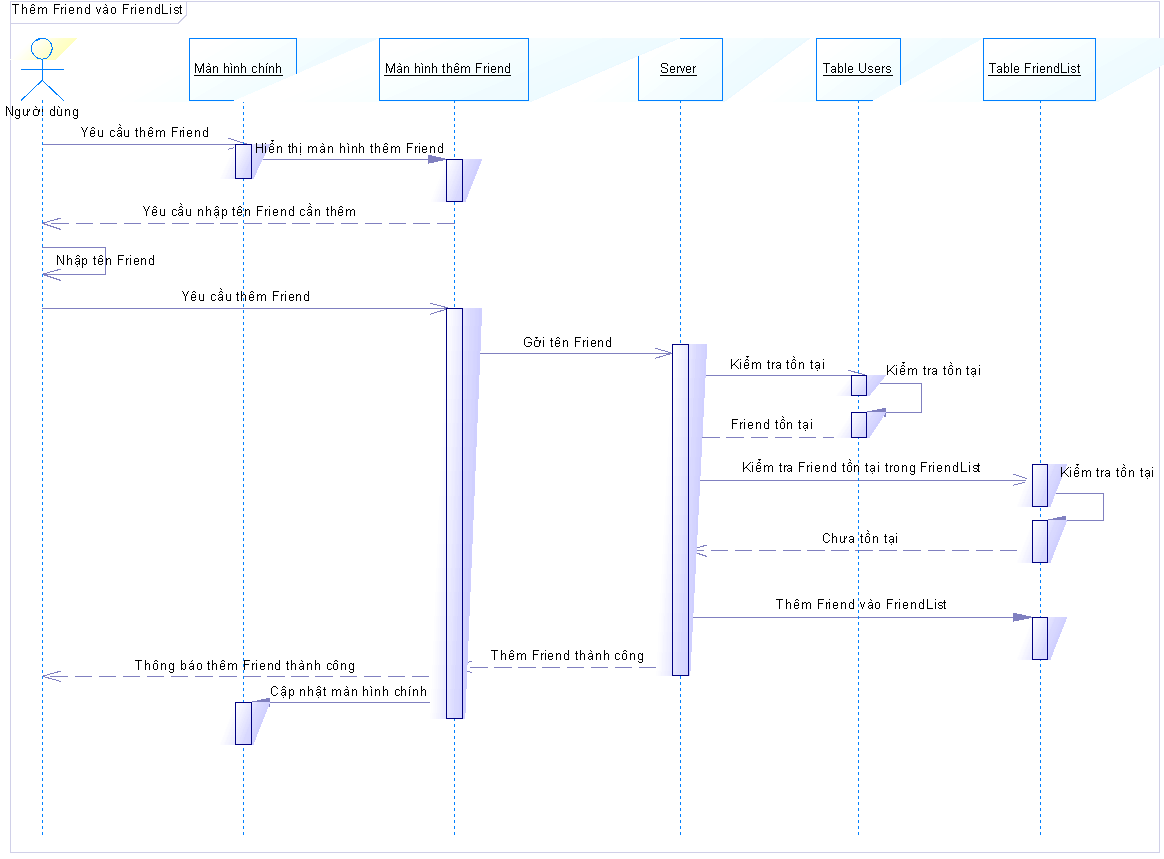
**Hình 35: Mô hình xử lý gởi tin nhắn Offline**

* + Mô tả:

Khi người dùng nhập tin nhắn và nhấn phím Enter (hoặc button Send), Client sẽ hiển thị tin nhắn trên Tab Chat đồng thời gởi tin nhắn và tên người nhận lên cho Server. Server sẽ kiểm tra xem người nhận có phải đang Online hay không. Nếu người nhận đang Offline, Server sẽ lưu tin nhắn cùng tên người gởi vào bảng OfflineMessage để gởi cho người nhận ở lần đăng nhập tiếp theo.

#### **Các xử lý đối với FriendList**

* Thêm một Friend vào FriendList:
  + Mô hình xử lý:

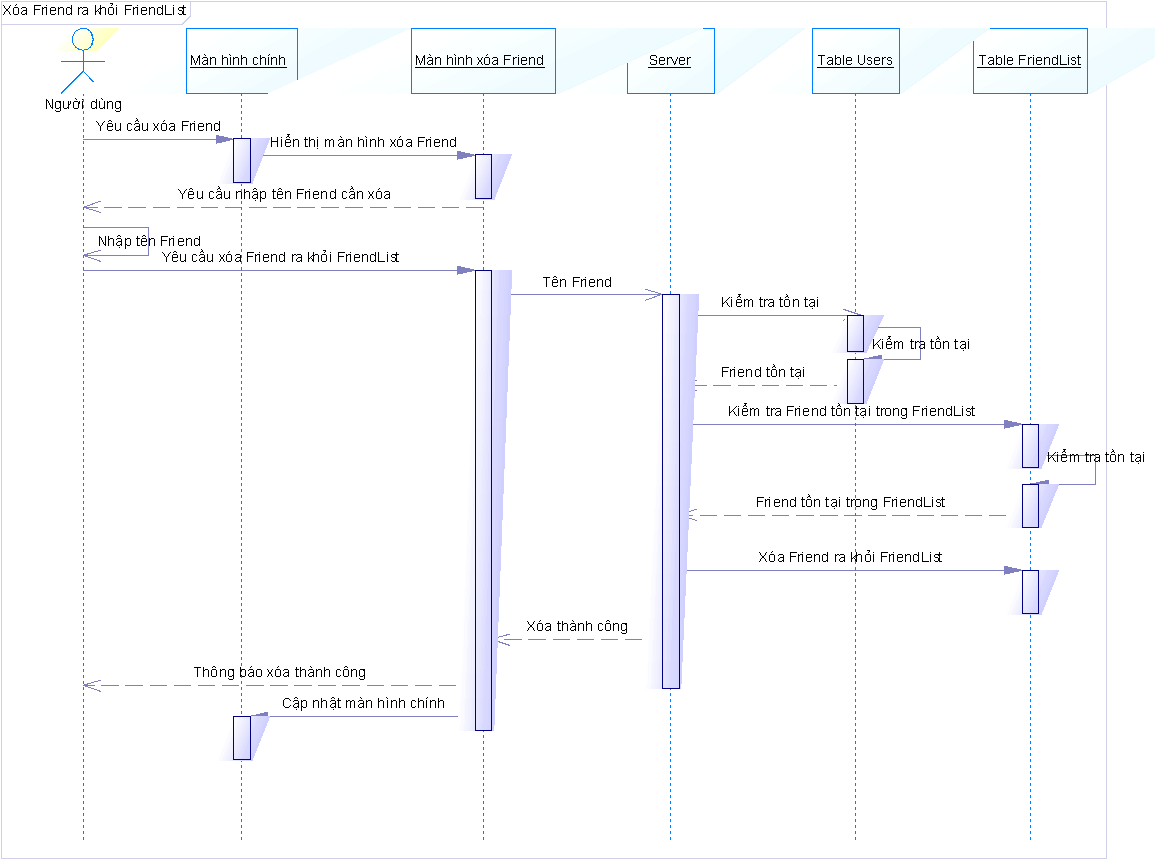


**Hình 36: Mô hình xử lý thêm Friend**

* + Mô tả:

Khi người dùng chọn chức năng thêm một Friend vào FriendList, màn hình thêm FriendList sẽ được mở ra. Người dùng sẽ nhập tên của Friend và sau đó Client sẽ gởi tên Friend này lên cho Server. Trước tiên, Server sẽ kiểm tra Friend này có tồn tại hay không. Tiếp theo, sẽ kiểm tra Friend này đã được thêm vào FriendList trước đó hay chưa. Nếu Friend này chưa có trong FriendList, Server sẽ thêm Friend này vào FriendList của người dùng. Cuối cùng, Server sẽ gởi kết quả của công việc về cho Client. Dựa vào kết quả nhận được, Client sẽ thông báo cho người dùng biết là việc thêm thành công hay thất bại (có hai nguyên nhân thất bại là Friend không tồn tại và Friend đã có trong FriendList rồi).

* Xóa một Friend ra khỏi FriendList:
  + Mô hình xử lý:



**Hình 37: Mô hình xử lý xóa Friend**

* + Mô tả:

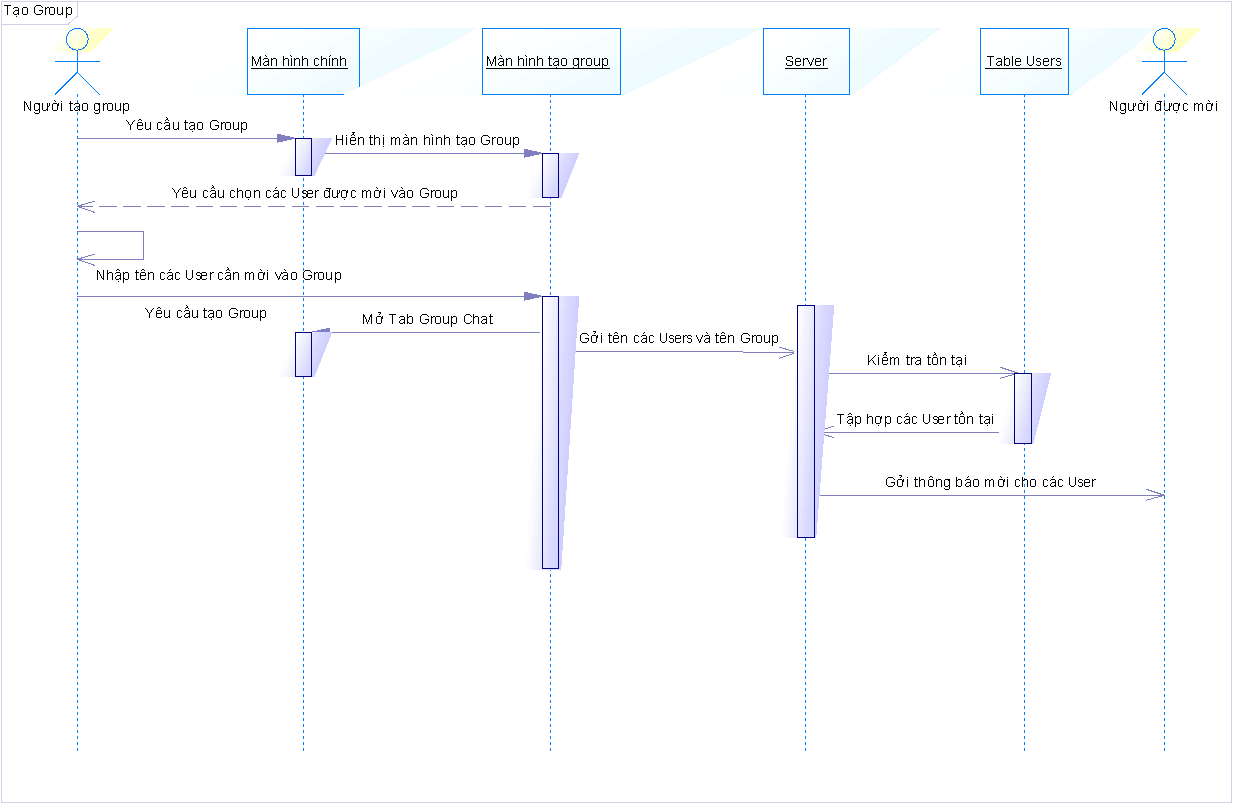
Khi người dùng chọn chức năng xóa một Friend vào FriendList, màn hình thêm FriendList sẽ được mở ra. Người dùng sẽ nhập tên của Friend và sau đó Client sẽ gởi tên Friend này lên cho Server. Trước tiên, Server sẽ kiểm tra Friend này có tồn tại hay không. Tiếp theo, sẽ kiểm tra Friend này đã được thêm vào FriendList trước đó hay chưa. Nếu Friend này đã có trong FriendList, Server sẽ xóa Friend này ra khỏi FriendList của người dùng. Cuối cùng, Server sẽ gởi kết quả của công việc về cho Client. Dựa vào kết quả nhận được, Client sẽ thông báo cho người dùng biết là việc Xóa thành công hay thất bại (có hai nguyên nhân thất bại là Friend không tồn tại và Friend chưa có trong FriendList) đồng thời cập nhật lại màn hình chính nếu cần.

* + Ghi chú:

Chức năng này có thể được gọi khi người dùng nhấn phím Delete trong Listbox FriendList.

#### **Các xử lý đối với Group Chat**

* Khi User tạo Group:
  + Mô hình xử lý:

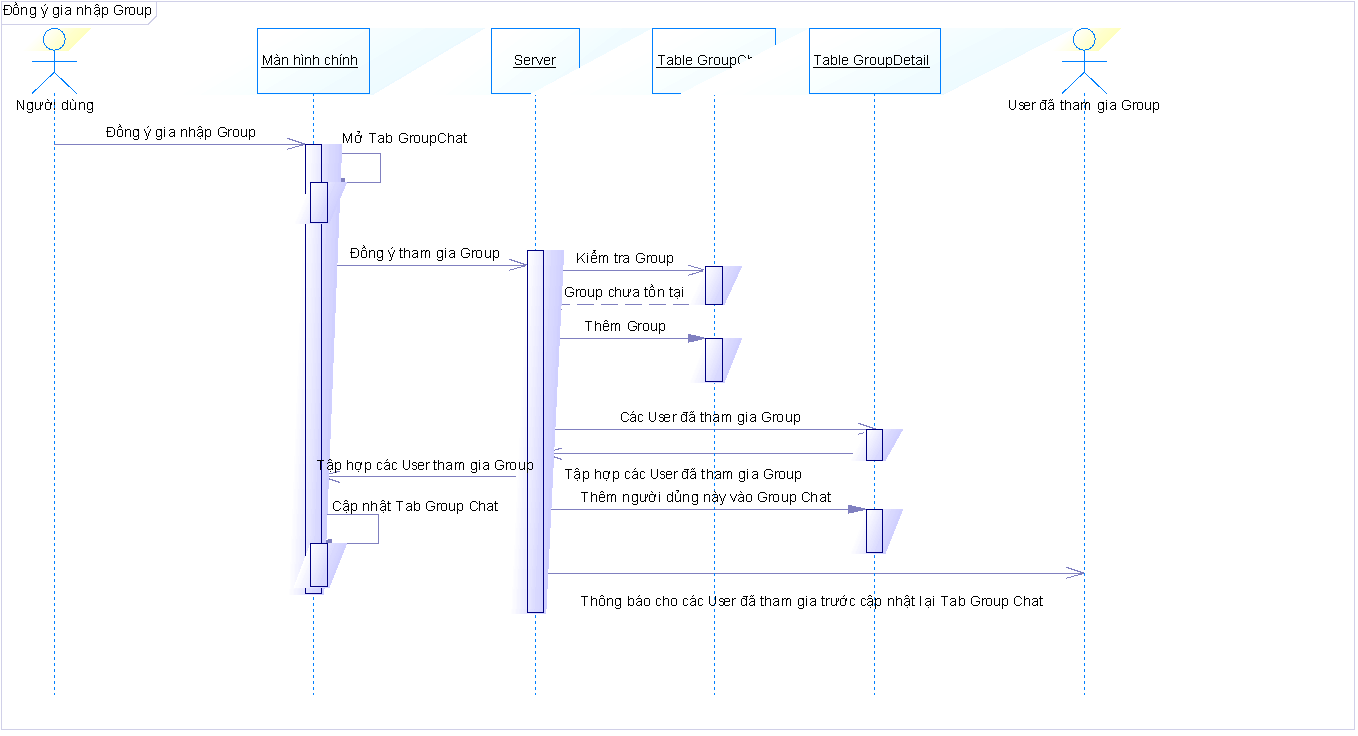


**Hình 38:** **Mô hình xử lý tạo Group**

* + Mô tả:

Khi người dùng chọn chức năng Invite Group thì màn hình tạo Group sẽ xuất hiện. Người dùng nhập danh sách các Users muốn mời và có thể nhập thêm Invite Message rồi nhấn Invite. Client sẽ tự phát sinh ra tên Group và sẽ gởi kèm nó với danh sách các Users muốn mời đồng thời cũng mở thêm một Tab Group Chat. Server sẽ lọc ra các Users đang Online trong danh sách các Users được mời. Sau đó, Server sẽ gởi lời mời kèm theo tên người mời đến các Users được mời.

* Khi User đồng ý gia nhập Group:
  + Mô hình xử lý:

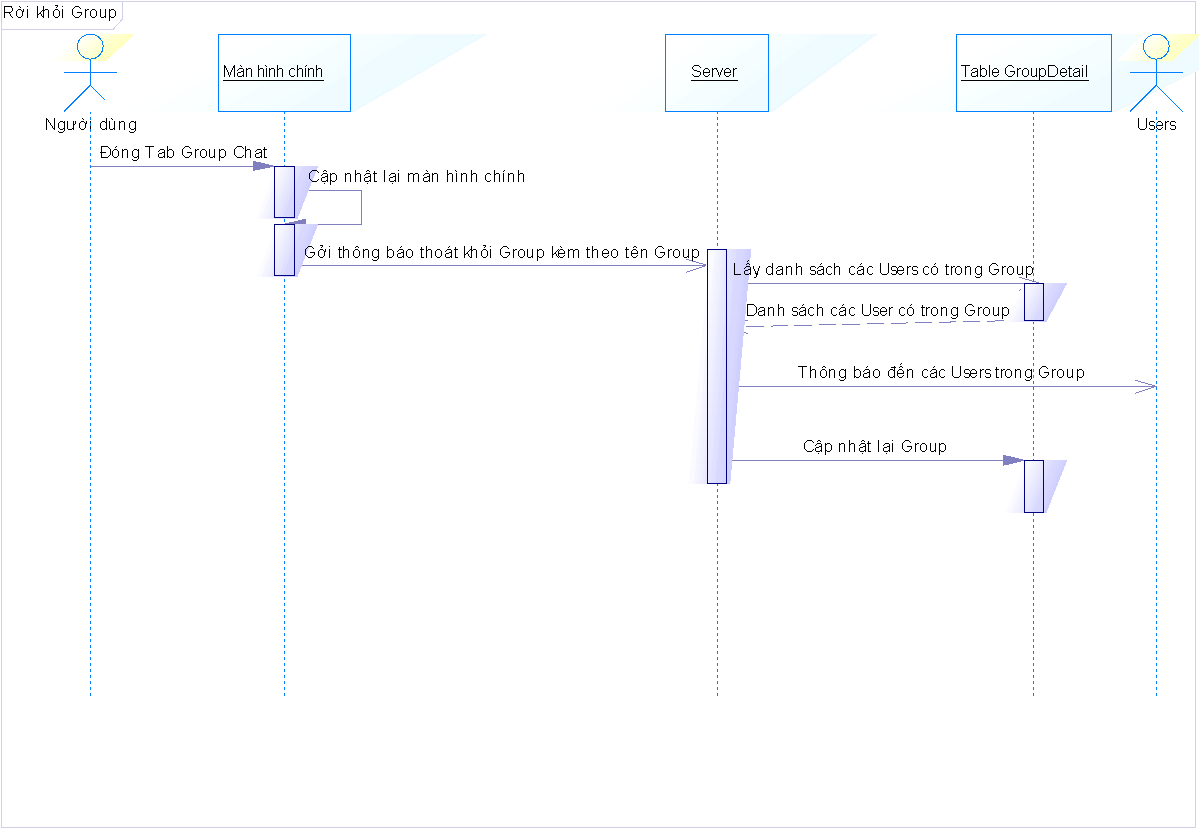


**Hình 39: Mô hình xử lý đồng ý gia nhập Group**

* + Mô tả:

Khi người dùng đồng ý lời mời gia nhập nhóm, Client sẽ gởi thông báo đồng ý cho Server. Server sẽ kiểm tra xem Group này đã được lưu hay chưa (Group chưa được lưu nếu chưa có người dùng nào đồng ý tham gia). Sau đó, Server sẽ lưu người dùng này vào GroupDetail. Cuối cùng, Server sẽ gởi tên tất cả các Users đã tham gia Group cho người dùng, đồng thời cũng gởi thông báo đến cho các Users trong Group là người dùng này đã đồng ý gia nhập nhóm.

* Khi User thoát khỏi Group:
  + Mô hình xử lý:

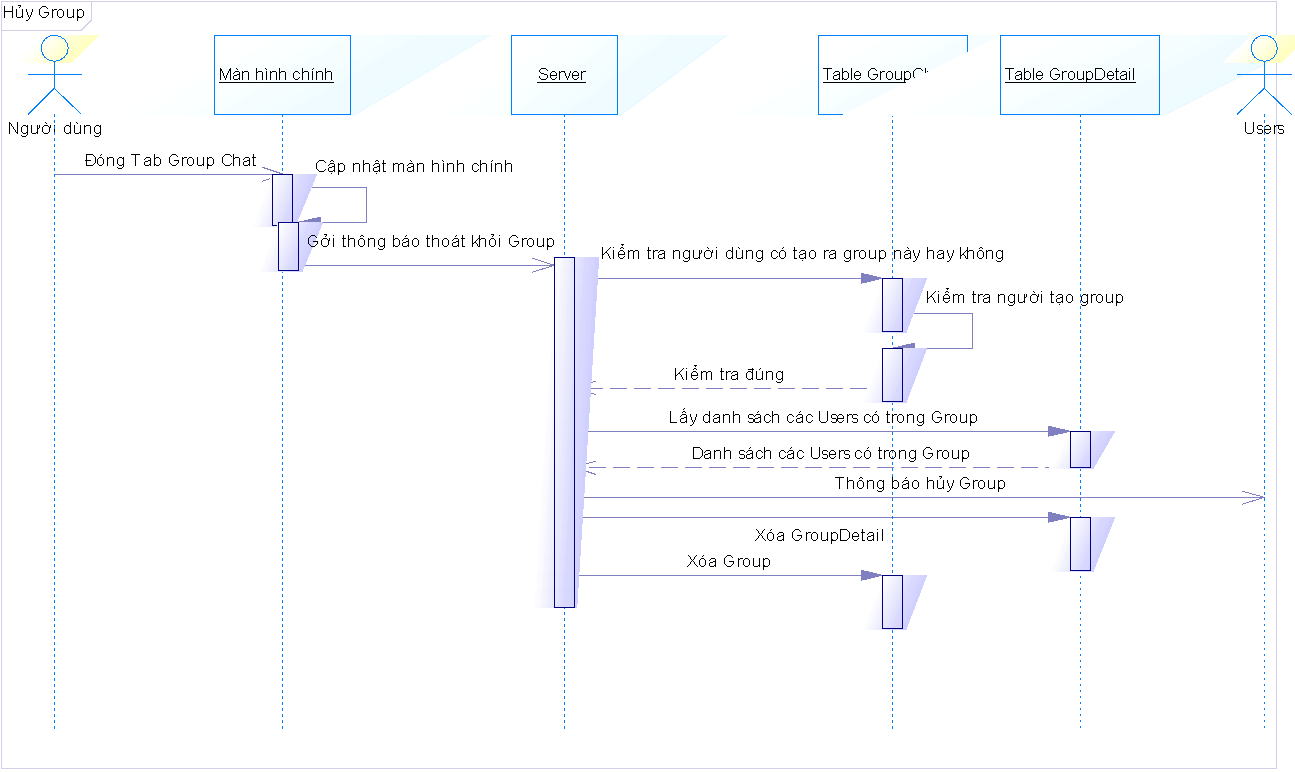


**Hình 310: Mô hình xử lý thoát khỏi Group**

* + Mô tả:

Khi một người dùng tắt Tab Group Chat, Client sẽ gởi thông báo thoát khỏi Group cho Server. Server trước tiên kiểm tra người dùng này có phải là người đã tạo Group hay không. Nếu không phải, Server sẽ gởi thông báo tới các Users khác trong Group. Cuối cùng, Server sẽ cập nhật lại GroupDetail.

* Khi User hủy Group:
  + Mô hình xử lý:



**Hình 311: Mô hình xử lý hủy Group**

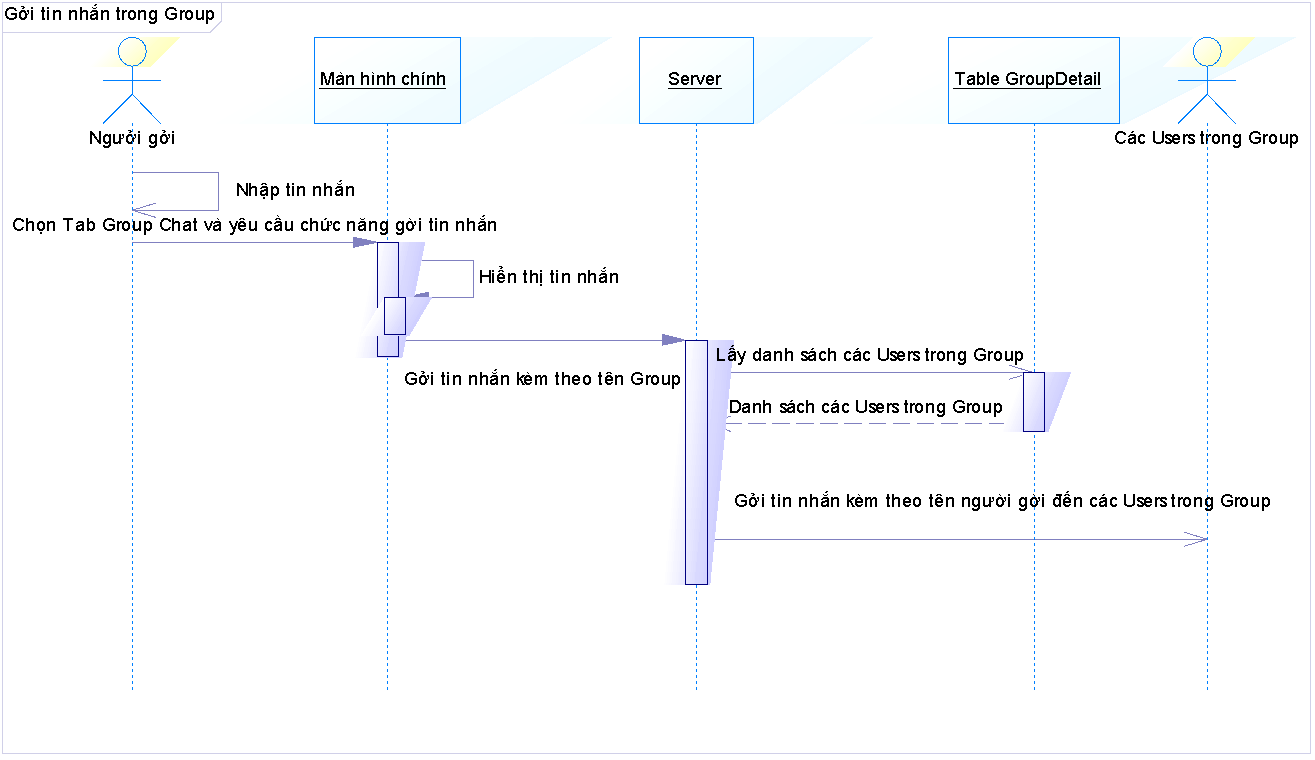
* + Mô tả:

Khi một người dùng tắt Tab Group Chat, Client sẽ gởi thông báo thoát khỏi Group cho Server. Server trước tiên kiểm tra người dùng này có phải là người đã tạo Group hay không. Nếu đúng, Server sẽ gởi thông báo hủy Group tới các Users khác trong Group. Cuối cùng, Server sẽ xóa Group này khỏi GroupDetail và GroupChat.

* + Ghi chú:

Khi nhận được thông báo hủy Group từ Users, Client sẽ tắt Tab Group Chat tương ứng đồng thời thông báo cho người dùng.

* Khi User gởi tin nhắn tới các thành viên trong Group:
  + Mô hình xử lý:



**Hình 312: Mô hình xử lý gởi tin nhắn trong Group**

* + Mô tả:

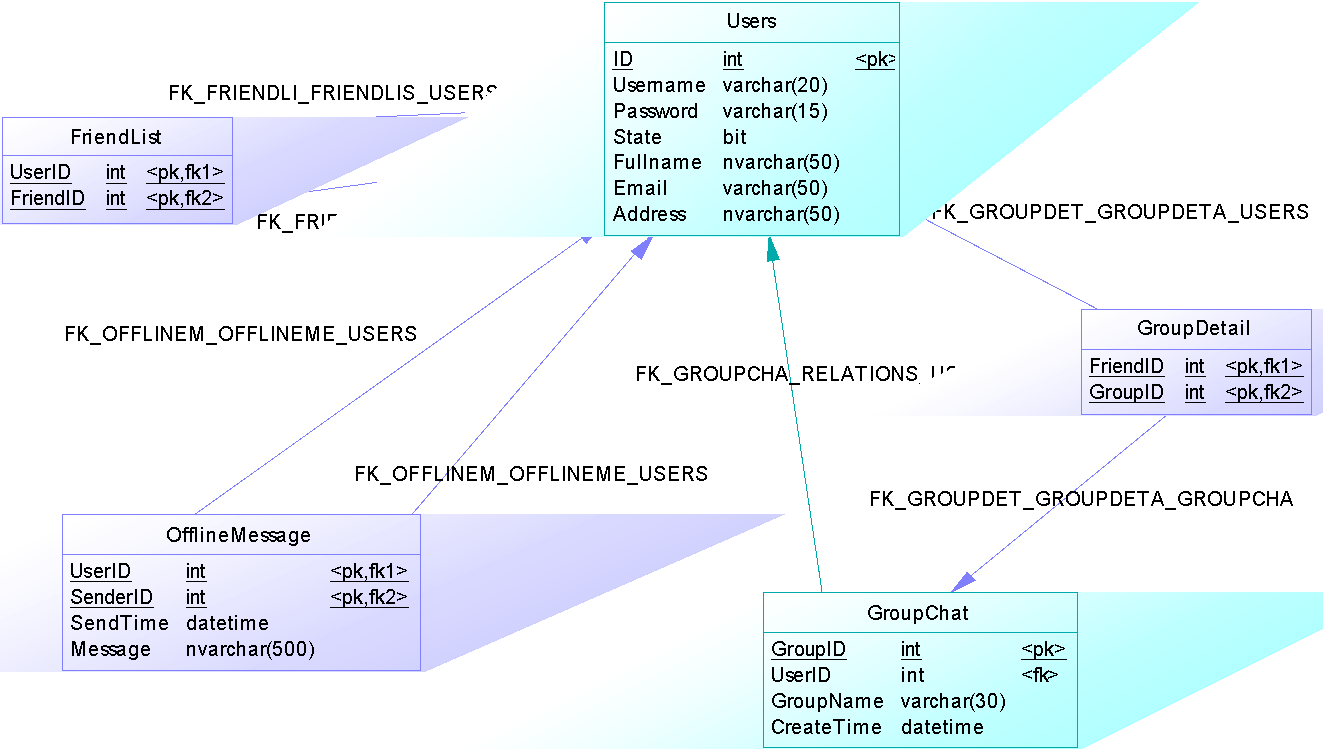
Khi người dùng nhập tin nhắn và nhấn button Send trong Tab Group Chat, Client sẽ gởi tin nhắn và tên Group cho Server. Dựa vào tên Group nhận được, Server sẽ tìm tất cả các Users đã tham gia Group và gởi tin nhắn kèm theo tên người gởi đến các Users này.

* 1. **Thiết kế dữ liệu**

### **Chuẩn hóa dữ liệu:**

* Users (ID, Username, Password, State, Fullname, Email, Address)
* FriendList(UserID, FriendID)
* OfflineMessage(UserID, SenderID, SendTime, Message)
* GroupChat(GroupID, UserID, GroupName, CreateTime)
* GroupDetail(FriendID, GroupID)

### **Mô hình dữ liệu ở mức vật lý:**



**Hình 313: Mô hình dữ liệu ở mức vật lý**

### **Thiết kế dữ liệu:**

Gồm có các bảng Users, FriendList, OfflineMessage, GroupChat, GroupDetail để lưu trử các thông tin về user để đăng nhập, quan hệ giữa các user, lưu tin nhắn offline của các user và lưu tạm thời thông tin các nhóm Chat.

* Table Users
* Chức năng: Lưu thông tin về User
* Danh sách các thuộc tính:

**Bảng 31: Table Users**

| **STT** | **Tên thuộc tính** | **Kiểu dữ liệu** | **Diễn giải** |
| --- | --- | --- | --- |
| 1 | ID | int (4) | Tự động tăng, Khóa chính |
| 2 | Username | varchar (20) | Tên đăng nhập |
| 3 | Password | varchar (15) | Mật khẩu đăng nhập |
| 4 | State | bit (1) | Trạng thái của User |
| 5 | Fullname | nvarchar (50) | Tên đầy đủ của User |
| 6 | Email | varchar (30) | Email của User |
| 7 | Address | nvarchar (50) | Địa chỉ của User |

* Table FriendList
* Chức năng: Lưu thông tin về quan hệ giữa các User
* Danh sách các thuộc tính:

**Bảng 32: Table FriendList**

| **STT** | **Tên thuộc tính** | **Kiểu dữ liệu** | **Diễn giải** |
| --- | --- | --- | --- |
| 1 | UserID | int (4) | ID của User, Khóa chính |
| 2 | FriendID | int (4) | ID của Friend, Khóa chính |

* Table OfflineMessage
* Chức năng: Lưu các tin nhắn offline của User
* Danh sách các thuộc tính:

**Bảng 33: Table OfflineMessage**

| **STT** | **Tên thuộc tính** | **Kiểu dữ liệu** | **Diễn giải** |
| --- | --- | --- | --- |
| 1 | UserID | int (4) | ID của người nhận, Khóa chính |
| 2 | SenderID | int (4) | ID của người gởi, Khóa chính |
| 3 | SendTime | datetime (8) | Thời điểm gởi tin nhắn, Khóa chính |
| 4 | Message | nvarchar (500) | Nội dung tin nhắn |

* Table GroupChat
* Chức năng: Lưu tạm thời thông tin của các nhóm Chat
* Danh sách các thuộc tính:

**Bảng 34: Table GroupChat**

| **STT** | **Tên thuộc tính** | **Kiểu dữ liệu** | **Diễn giải** |
| --- | --- | --- | --- |
| 1 | GroupID | int (4) | Tự động tăng, Khóa chính |
| 2 | UserID | int (4) | ID của User tạo nhóm |
| 3 | GroupName | varchar (20) | Tên của nhóm |
| 4 | CreateTime | datetime (8) | Thời điểm tạo nhóm |

* Table GroupDetail
* Chức năng: Lưu thông tin chi tiết của các nhóm Chat
* Danh sách các thuộc tính:

**Bảng 35: Table GroupDetail**

| **STT** | **Tên thuộc tính** | **Kiểu dữ liệu** | **Diễn giải** |
| --- | --- | --- | --- |
| 1 | GroupID | int (4) | ID của Group, Khóa chính |
| 2 | FriendID | int (4) | ID của của User thành viên trong Group, Khóa chính |

### **Mô tả các ràng buộc toàn vẹn:**

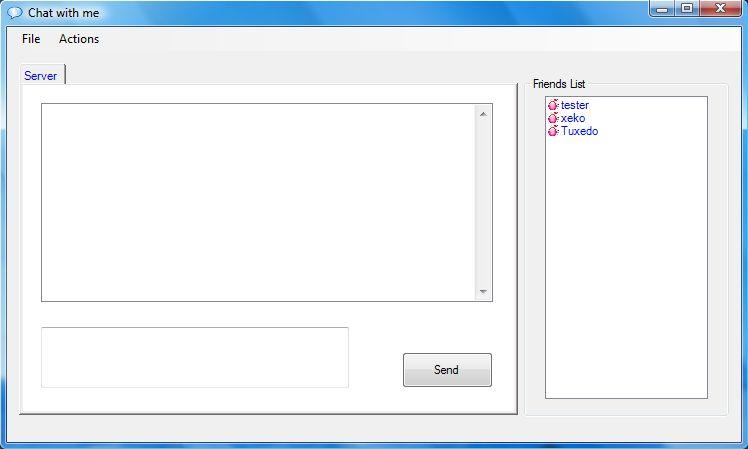
* ***Ràng buộc 1***: Username trong Table Users không được trùng nhau.
* ***Ràng buộc 2***: UserID và FriendID trong Table FriendList phải tồn tại trong Table Users.
* ***Ràng buộc 3***: UsersID trong Table GroupChat phải tồn tại trong Table Users.
* ***Ràng buộc 4***: GroupID trong Table GroupDetail phải tồn tại trong Table GroupChat.
* ***Ràng buộc 5***: FriendID trong Table GroupDetail phải tồn tại trong Table Users.
* ***Ràng buộc 6***: SenderID và UserID trong Table OfflineMessage phải tồn tại trong Table Users.
  1. **Thiết kế giao diện**

### **Màn hình đăng nhập**



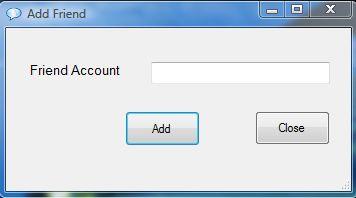
**Hình 314: Màn hình đăng nhập**

### **Màn hình chính**



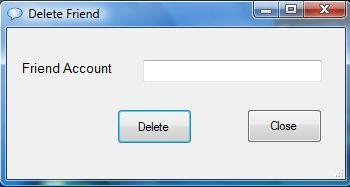
**Hình 315: Màn hình chính**

### **Màn hình thêm Friend**



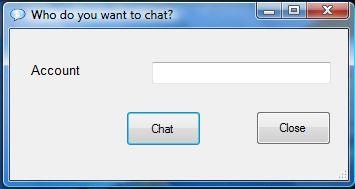
**Hình 316: Màn hình thêm Friend**

### **Màn hình xóa Friend**



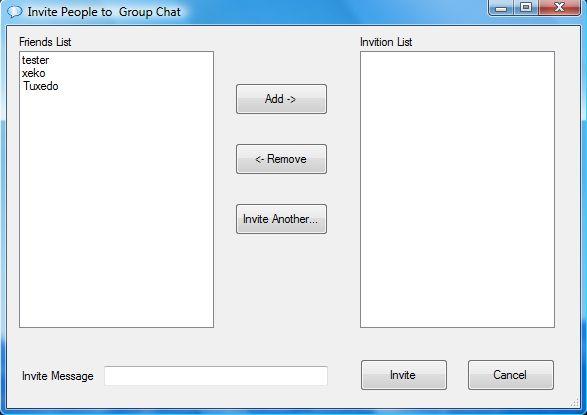
**Hình 317: Màn hình xóa Friend**

### **Màn hình Chat With**



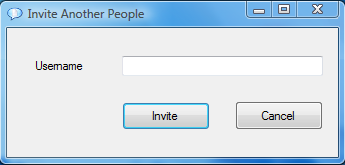
**Hình 318: Màn hình Chat With**

### **Màn hình Invite Group**



**Hình 319: Màn hình Invite Group**

### **Màn hình Invite Another**



**Hình 320: Màn hình Invite Another**

# **CÀI ĐẶT – THỬ NGHIỆM**

* 1. **Cài đặt chương trình**
  2. **Hướng dẫn sử dụng**

# **KẾT LUẬN**

* 1. **Kết quả đạt được**
* Đề tài “Chương trình Chat” đã thực hiện được các nội dung sau:
* Tìm hiểu được cách thức lập trình Socket và lập trình đa luồng trên môi trường .NET
* Tìm hiểu được cách thức hoạt động của một chương trình Chat đơn giản.
* Ưu và nhược điểm của chương trình:
* Ưu điểm:
* Nhược điểm:
  + Chưa gửi kèm các hình ảnh khi Chat.
  + Chưa có chức năng gửi File giữa các Users.
  + Chỉ hỗ trợ Chat Text đơn thuần.
  1. **Hướng phát triển**

Về cơ bản, nghiên cứu đã đạt được những yêu cầu đã đặt ra. Tuy nhên nếu có điều kiện, đồ án sẽ cố gắng phát triển thêm các chức năng sau:

* Hỗ trợ gửi hình ảnh khi Chat
* Hỗ trợ chức năng gởi file.
* Hỗ trợ chức năng Voice Chat và Webcam.

# **TÀI LIỆU THAM KHẢO**

Tài liệu:

1. Dương Quang Thiện, 2005, .NET toàn tập.
2. Nguyễn Ngọc Bình Phương, 2005, Các giải pháp lập trình C#.
3. Sybex, 2003, C-Sharp Network Programming

Các trang Web

1. http://www.codeproject.com
2. http://www.codeguru.com