



Adding a Security Layer to Networks



FEBRUARY 14, 2017

SUPERVISOR

Prof. Yousef B. Mahdy

ACKNOWLEDGEMENT

first we should thanks Allah for helping us to complete this project and produce it in this acceptable view .

we would like to thank our faculty management team for provision computers and facilities all the time of developing the project.

We are proud of faculty of computer and information ,assiut university where this work has been performed.

We wish to express our deep gratitude to our supervisor

Prof. Yousef B. Mahdy

For his encouragement and helping us with valuable suggestions through prepration of this project special thanks for.

Finally,our greatest dept to our families for their outstanding support which allowed us to be what we are,they were responsible for getting all of the complex pieces put together instead of us to give us the time to be creative and successful.

Best regards,

Team of the project

ABSTRACT:

Network Security , this is a sub-domain which is the most integral and important part of a network . Basically, the major goal of this project is to implement various security algorithms , or develop a new one. additionally , one can include biometrics , for adding a security layer. This included using the fingerprints , retina scanning , voice , etc for as a password for access to the network or a database on a server.

Cat: Security, Network

Tools:

programming language: C# or java

Project team members:

- **Ahmed Hamdy**
- **Lamiaa Zaghloul**
- **Omnia Abdalla**
- **Mohamed Rashad**
- **Fady Khayrat**
- **Beshoy Mousa**
- **Ahmed Anwar**
- **Marwa Abdalla**

Chapter 1: Network Programming Fundamentals

Concepts of chapter

1.1- Introduction.....	6
1.2- What Is A Computer Network?	6
1.3- Servers & Clients	9
1.4- Application Distribution	10
1.5 Multitiered Applications.....	12

Chapter 2: Networked Client-Server Applications

Concepts of chapter:

2.1 Introduction	18
2.2 Transmission Control Protocol (TCP)	18
2.3 TCP/IP Client-Server Overview	20

Chapter 3: Encrypting Data in Network Connections

Contents of Chapter:

3.1 Introduction.....	31
3.2 What is Encryption.....	31
3.3 History of Encryption.....	32
3.4 How Encryption work.....	33
3.5 A symmetric encryption.....	33
3.6 Symmetric encryption.....	34

3.7 Data Encryption Standard.....	36
3.8 RSA algorithm.....	46
3.9 Creating A symmetric Encryption stream.....	48
3.10 SSL stream class.....	52
3.11 Project.....	60

Chapter 4: Application

Contents of Chapter:

4.1 Overview.....	69
4.2 Getting Start.....	69
4.3 Using SQLite GUI client.....	71
4.4 Interaction with your database.....	72
4.5 Simple Client-Server Application.....	73

Chapter 1: Network Programming Fundamentals

Concepts of chapter

1.1- Introduction.....	6
1.2- What Is A Computer Network?	6
1.3- Servers & Clients	9
1.4- Application Distribution	10
1.5 Multitiered Applications.....	12

Chapter One Network Programming Fundamentals

1.Introduction

Network applications pervade today's modern computing environment. If you use email, a web browser, or a chat program like Windows Live Messenger, you're using software applications powered by network technology. This chapter serves two primary purposes. First, it gives you a broad understanding of key networking concepts and terminology. Here you will learn the difference between server software and server hardware, the meanings of the terms network, packet, datagram, TCP/IP and UDP, and how applications can be physically and logically distributed in a networked environment.

The second purpose of this chapter is to introduce you to the concepts of multitiered , distributed applications. Modern network applications are often logically tiered , with one or more of their logical tiers physically deployed on different computers. It will be important for you to understand the terminology associated with these concepts as you learn to write network-enabled applications.

2. What Is A Computer Network?

A computer network is an interconnected collection of computing devices. A computing device, for the purposes of this rather broad definition, can be any piece of equipment that exists to participate in or support a network in some fashion. Examples of computing devices include general purpose computers, special purpose computers, routers, switches, hubs, printers, etc.

Purpose Of A Network

Computer networks are built with a specific purpose in mind. The primary purpose of a computer network is resource sharing . A resource can be physical (i.e., a printer or a computer) or metaphysical (i.e., knowledge or data). Figure 1 shows a diagram for a simple computer network. This type of simple network is referred to as a local area network (LAN).

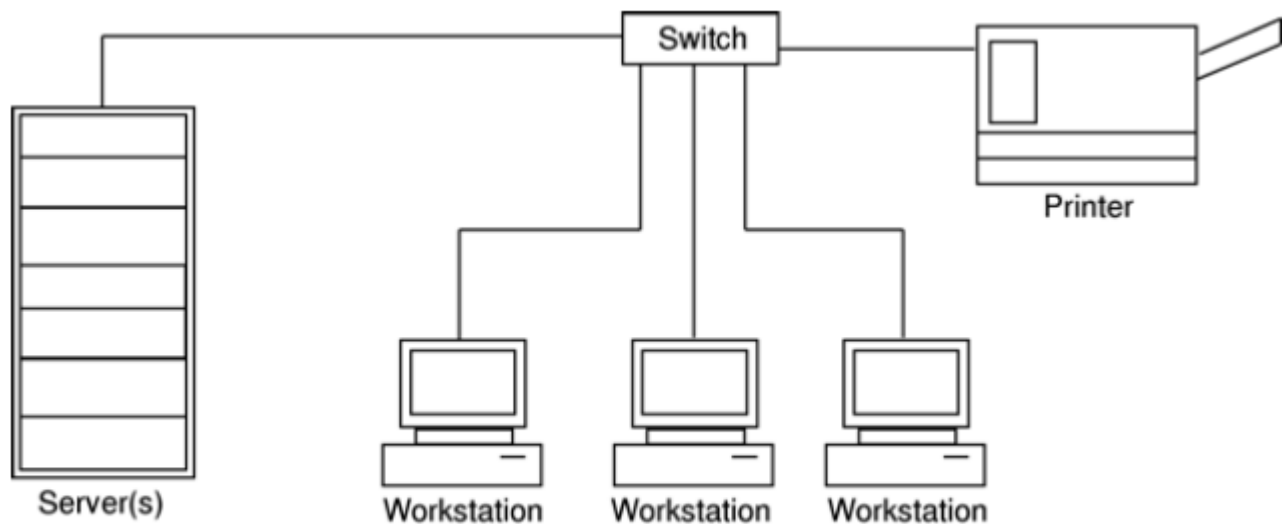


Figure 1: A Simple Computer Network

Referring to Figure 1 — the computing devices participating in this simple network include the workstations, the servers, the printer, and the switch. The switch facilitates network interconnection. In this configuration the work-stations and servers can share the computational resources offered by each computer on the network as well as the printing services offered by the printer. Data can also be offered up for sharing on each computer as well.

What Is a Protocol?

Now that we've got a bit of a feel for what the Internet is, let's consider another important buzzword in computer networking: *protocol*.

What is a protocol? What does a protocol do?

*A **protocol** defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event.*

The Internet, and computer networks in general, make extensive use of protocols. Different protocols are used to accomplish different communication tasks. As you read through this book, you will learn that some protocols are simple and straightforward, while others are complex and intellectually deep. Mastering the field of computer networking is equivalent to understanding the what, why, and how of networking protocols.

The Role Of Network Protocols

A protocol is a specification of rules that govern the conduct of a particular activity. Entities that implement or adhere to the protocol(s) for a given activity can participate in that activity. For example, Robert's Rules of Order

specify a set of protocols for efficiently and effectively conducting meetings. A similar analogy applies to computer networking.

Homogeneous Vs. Heterogeneous Networks

Computers participating in a computer network communicate with each other via a set of networking protocols.

There are generally two types of network environments: 1) homogeneous - where all the computers are built by the same company and can talk to each other via that company's proprietary networking protocol, or 2) heterogeneous-where the computers are built by different companies, have different operating systems, and therefore different proprietary networking protocols. An example of a homogenous network would be one comprised entirely of Apple Macintosh computers and Apple peripherals. The Macintosh computers could communicate perfectly fine with each other via AppleTalk which is an Apple networking protocol. In a perfect world, we would all use Apple Macintosh computers but the world is, alas, imperfect, and almost every

network in existence is heterogeneous in nature. Apple Macs running OS X must communicate with computers running Sun Solaris, Microsoft Windows, Linux, and a host of other hardware and operating system combinations.

The Unifying Network Protocols: TCP/IP

In today's heterogeneous computer network environment, the protocols that power the Internet — Transmission Control Protocol (TCP) and Internet Protocol (IP) — collectively referred to as TCP/IP, have emerged as the standard network protocols through which different types of computers can talk to each other. Figure 2 shows the local area network connected to the Internet via a router. So long as the computers on the LAN utilize an operating system that implements TCP/IP, then they can access the computational and data resources made available both internally and via the Internet. If the LAN does not utilize TCP/IP, then a bridge or gateway device would be required to perform the necessary internetwork protocol translation.

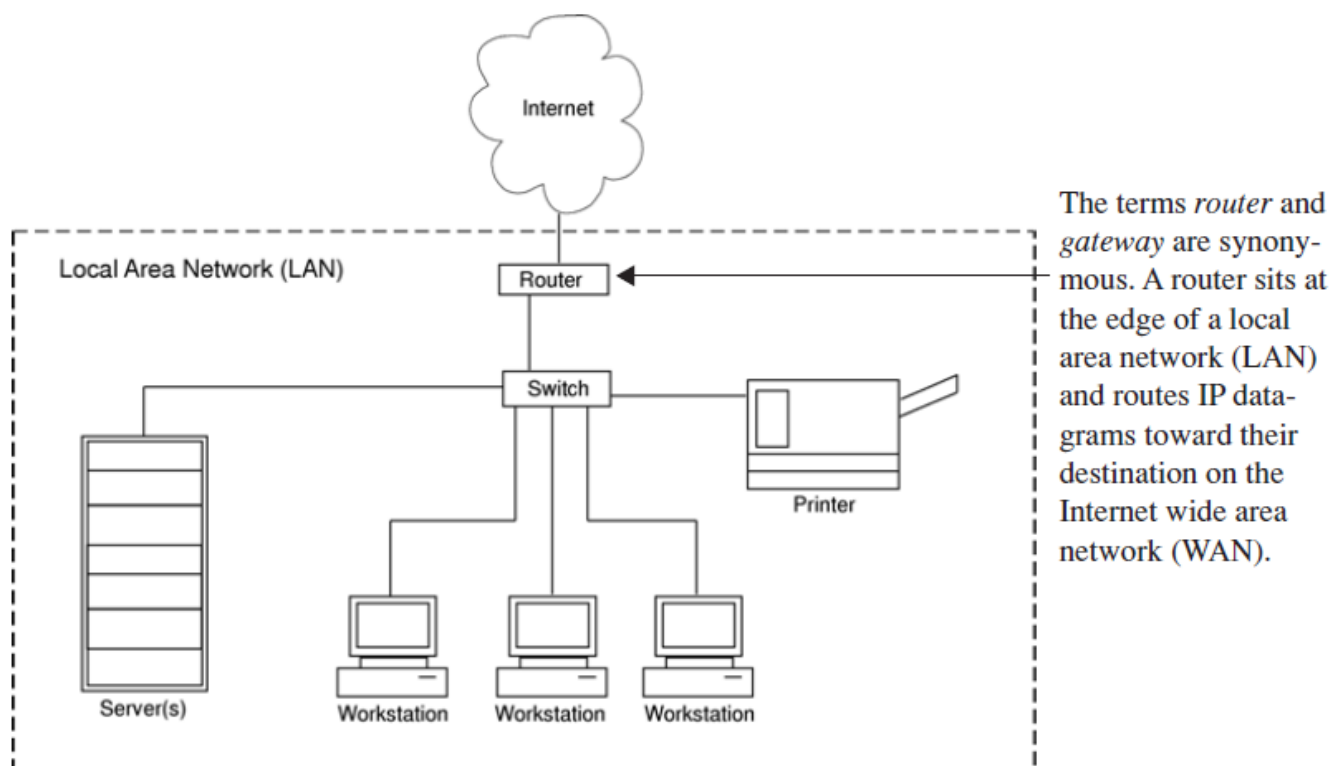


Figure 2: Local Area Network Connected to the Internet

What's So Special About The Internet?

What makes the Internet so special? The answer is — TCP/IP. The Internet is a vast network of computer networks. All of the networks on the Internet communicate with each other via TCP/IP. The TCP/IP protocols were developed with Department of Defense (DoD) funding. What the DoD wanted was a computer and communications network that was resilient to attack. If a piece of the Internet was destroyed by a nuclear blast, then data would be automatically routed through the surviving network connections. When one computer communicates with another computer via the Internet, the data it sends is separated into packets and transmitted one packet at a time to the designated computer. TCP/IP provides packet routing and guaranteed packet delivery. Because of the functionality provided by the TCP/IP protocols, the Internet is considered to be a robust and reliable way to transmit and receive data.

Figure 3 shows how the simple network of Agency A can share resources with other agencies via the Internet.

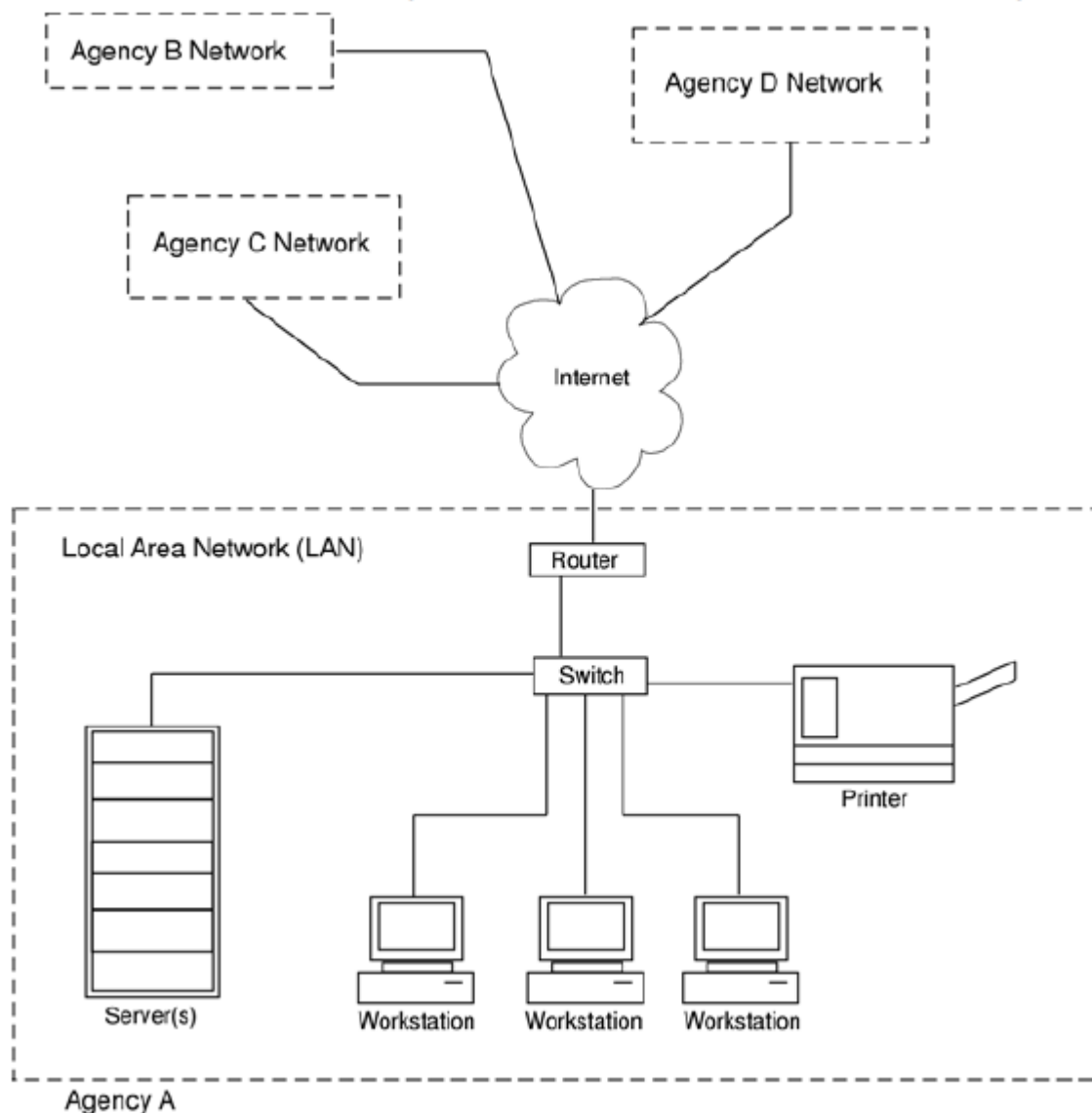


Figure 3: The Internet — A Network of Networks Communicating via Internet Protocols

3. Servers & Clients

The terms server and client each have both a hardware and software connotation. This section briefly discusses these terms in both aspects in greater detail to provide you with a foundation for the material presented in the next section.

Server Hardware And Software:

The term server is often used to refer both to a piece of computing hardware on which a server application runs and to the server application itself. I will use the term server to refer to hardware. I will use the term server application to refer to a software component whose job is to provide some level of service to another entity.

As Figure 4 illustrates, it is the job of a server to host server applications. However, as desktop computing power increases, the lines between client and server hardware become increasingly blurry. A good definition for a server then is any computer used to host one or more server applications as its primary job. A server is

usually (should be) treated as a critical piece of capital equipment within an organization. Server operating requirements are used to specify air conditioning, electrical, and flooring requirements for data centers. Servers are supported by data backup and recovery procedures and, if they are truly agency critical, will have some form of fault tolerance and redundancy designed in as well. A server running a server application is also referred to as a host . The term host extends to any computer running any application.

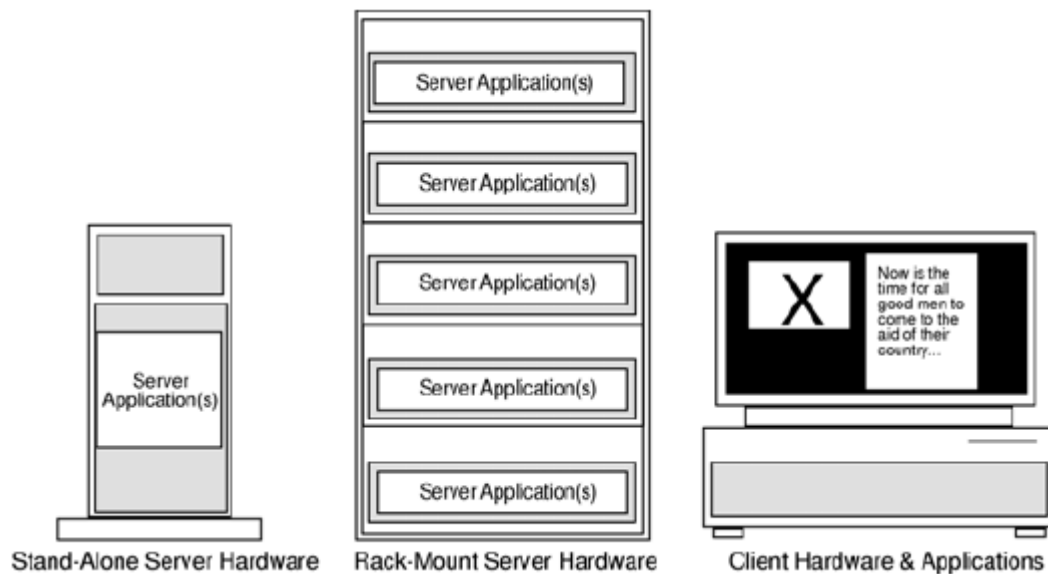


Figure 4: Client and Server Hardware and Applications

Client Hardware And Software:

The term client is also used to describe both hardware and software. Client hardware is any computing device that hosts an application that requires or uses the services of a server application. Client software is any application that requires or uses the services provided by a server application. For example, when you run Microsoft Internet Explorer on your home computer, you are running a client application. You use Internet Explorer to access web sites via the Internet. These web sites are served up by a web server (i.e ., an HTTP server), which is a server application hosted on a server somewhere out there in Internet land.

4.Application Distribution:

The term application distribution refers to where (i.e., on what physical computer) one or more pieces of a network application reside. This section discusses the concepts of physically distributing client and server applications. Server applications themselves can be further divided into multiple application layers with each distinct application layer being physically deployed to one or more computers. The concepts associated with multilayered applications are presented and discussed in the next section.

Physical Distribution On One Computer:

Client and server applications can both be deployed on the same computer. This is most often done for the purposes of testing during development. When you write client-server applications in Chapter 2, you will test them on your development machine. If you are fortunate enough to have a home network that includes multiple computers, you can test your client-server applications in a more real world setting. Figure 5 illustrates the concept of running client and server applications on the same physical hardware.

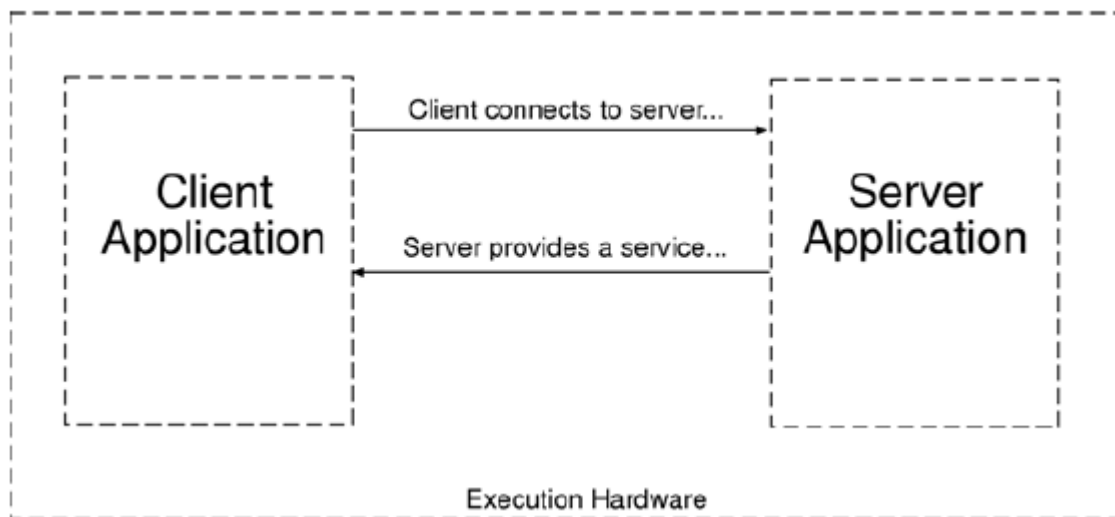


Figure 5: Client and Server Applications Physically Deployed to the Same Computer

Running Multiple Clients On The Same Computer

You can run multiple client applications on the same computer. To do this, your server application must be capable of handling multiple concurrent client requests for service. A server application with this capability is generally referred to as being multithreaded. Each incoming client connection is passed off to a unique thread for processing. The execution of multiple client applications, in addition to the server application, on the same hardware, is common practice during a software project's development and testing phases. Figure 6 illustrates the concept of running multiple client applications and the server application on the same hardware.

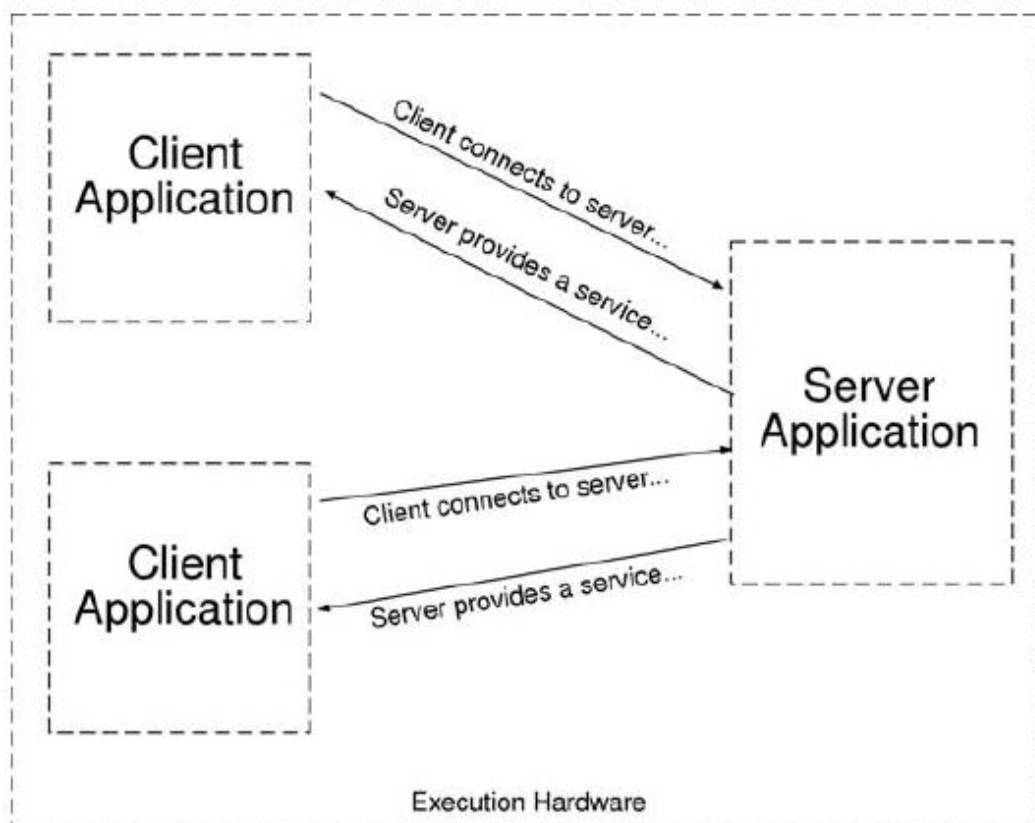


Figure 6: Running Multiple Clients on Same Hardware

Addressing The Local Machine

When testing client-server applications on your local machine, you can use the localhost IP address of 127.0.0.1 as the server application's host address.

Physical Distribution Across Multiple Computers

Although client and server applications can be co-located on the same hardware, it is more often the case that they are physically deployed on different machines geographically separated by great distance. Figure 7 illustrates this concept.

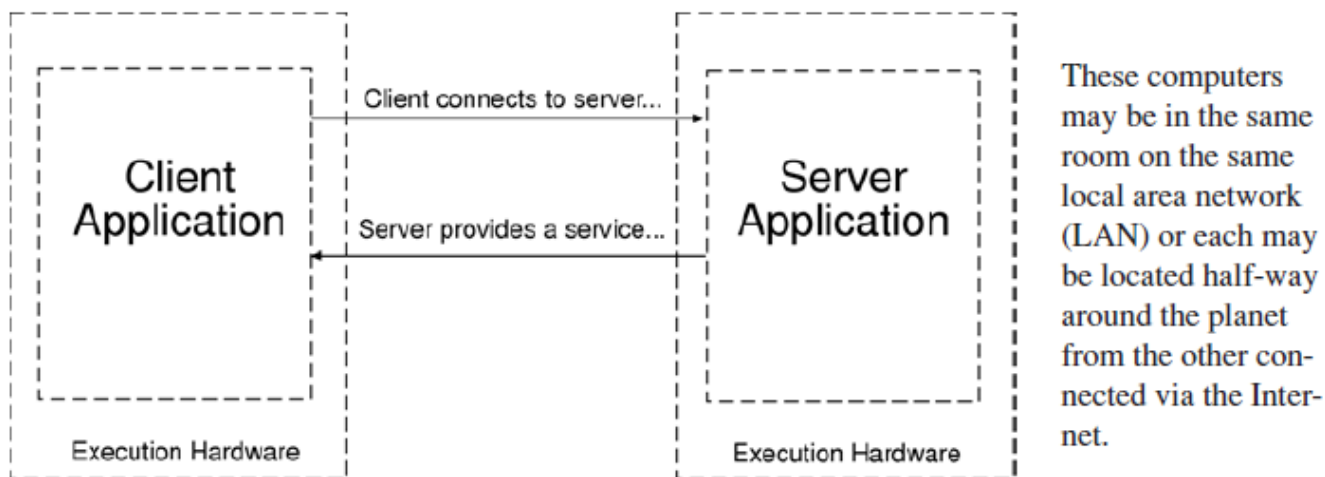


Figure 7: Client and Server Applications Deployed on Different Computers

6.Multitiered Applications

Up until now I have referred to client and server applications as if they were monolithic components. In reality, modern client-server applications are logically segmented into functional layers. These layers are also referred to as application tiers . An application composed of more than one tier is referred to as a multitiered application. This section discusses the concepts related to multitiered applications in greater detail.

Logical Application Tiers

Figure 8 illustrates the concept of a multitiered application.

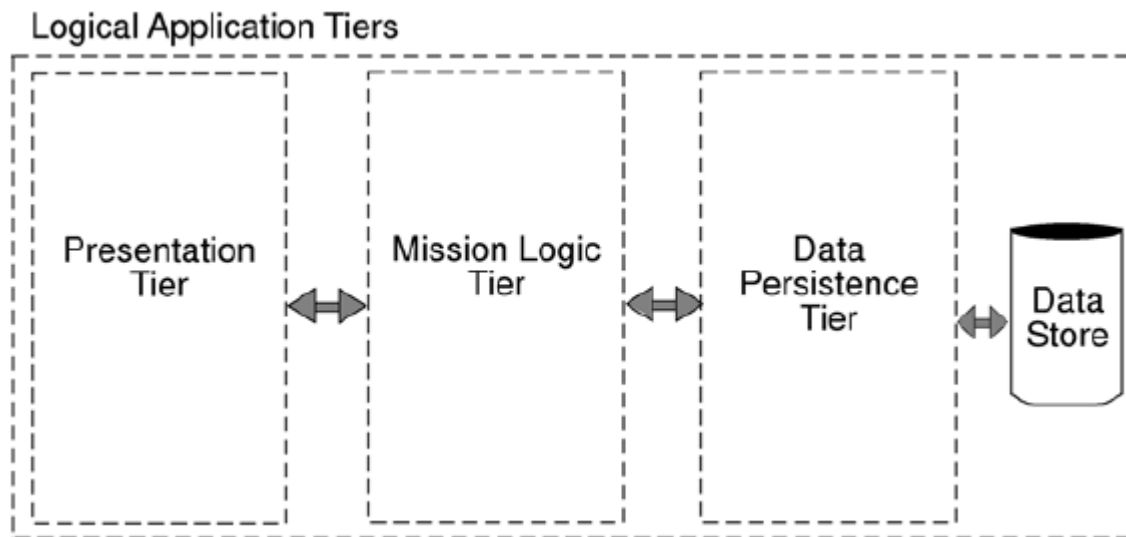


Figure 8: A Multitiered Application

Referring to Figure 8 — in this example the application comprises three functional tiers: 1) presentation tier 2) mission logic tier , and 3) data persistence tier.

. As their names suggest, each tier has a distinct responsibility for delivering specific application functionality. The presentation tier is concerned with rendering the user interface. The mission logic tier (a.k.a. business logic tier) contains the code that implements the application's services. (i.e., data processing algorithms, mission-oriented processes, etc.) (I use the term mission logic tier interchangeably with the term business logic

tier when referring to multitiered applications written for Department of Defense clients.) The data persistence tier is responsible for servicing the data needs (i.e., data storage and retrieval) of the mission logic layer as quickly and reliably as possible.

Another way to think about each tier's responsibilities is as a separation of concerns:

- the presentation tier is concerned with how a user interacts with an application
- the mission logic tier is concerned with implementing mission support processes
- the data persistence tier is concerned with reliable data storage and retrieval in support of mission processes

Physical Tier Distribution

The logical application tiers may be physically deployed on the same computer, as is illustrated in Figure 9. It is more likely the case, however, that logical application tiers are physically deployed to separate and distinct computing nodes located some distance apart. Figure 10 illustrates this concept by showing each logical tier deployed to a different computer. In between this extreme lies any combination of logical tier deployments as best

supports an agency's mission requirements.

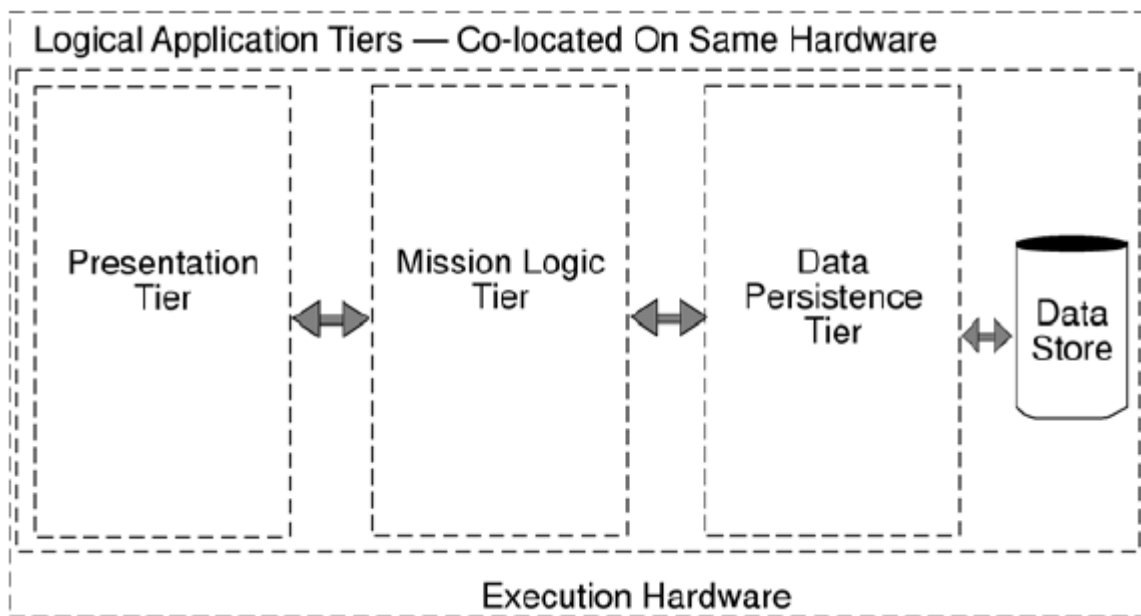


Figure 9: Physically Deploying Logical Application Tiers on Same Computer

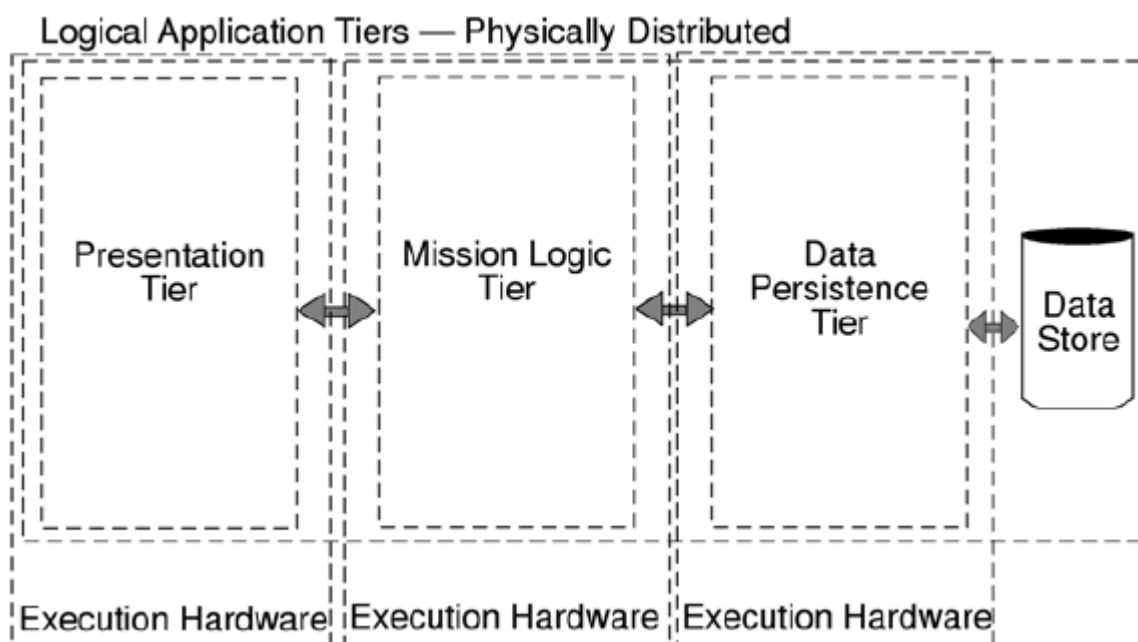


Figure 10: Logical Application Tiers Physically Deployed to Different Computers

Internet Networking Protocols: Nuts & Bolts

This section discusses the concepts associated with the Internet protocols and related terminology in greater detail. You'll find this background information helpful when navigating your way through the System.Net namespace looking for a solution to your network programming problem.

The Internet Protocols: TCP, UDP, And IP

The Internet protocols facilitate the transmission and reception of data between participating client and server applications in a packet-switched network environment. The term packet-switched network means that data traveling along network pathways is divided into small, routable packages referred to as packets. If a communication link between two points on a network goes down, the packets are routed through remaining network connections to their intended destination.

The Internet protocols work together as a layered protocol stack as is shown in Figure 11.

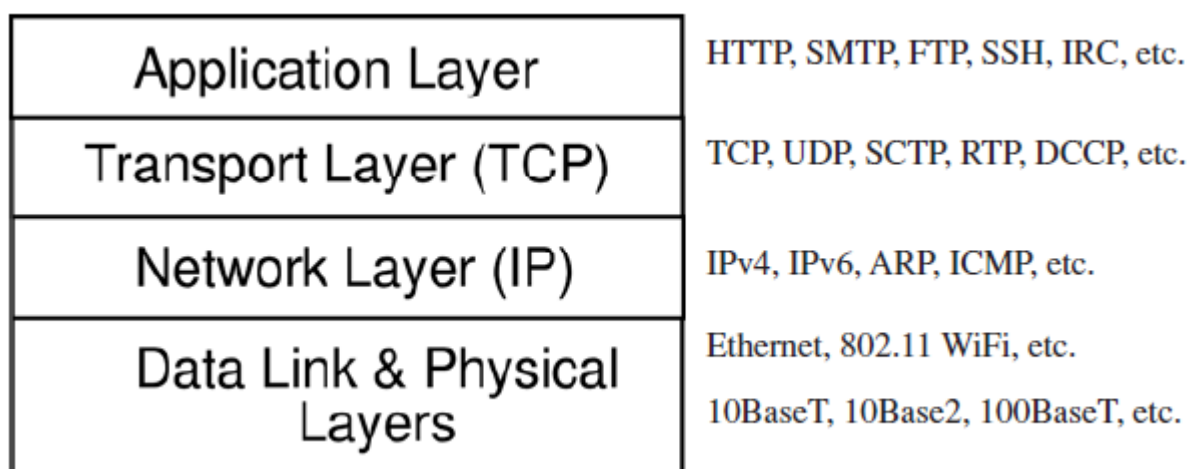


Figure 11: TCP/IP Protocol Stack

The layered protocol stack consists of the application layer, the transport layer, the network layer, the data link layer, and the physical layer. Each protocol stack layer provides a set of services to the layer above it. Several examples of protocols that may be employed at each level in the application stack are also shown in Figure 11. For more information on protocols not discussed in this chapter, please consult the sources listed in the references section.

The Application Layer

The application layer represents any internet enabled application that requires the services of the transport layer. Typical applications you may be familiar with include File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), TELNET, or a custom internet application such as one you might write. The application layer relies on services provided by the transport layer.

Transport layer

The purpose of the transport layer is to provide host-to-host, connection-oriented, data transmission service to the application layer. Two internet protocols that function at the transport layer include the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). The .NET Framework supports both of these protocols directly in that normal network communication takes place using TCP, but UDP can be utilized if required.

Transmission Control Protocol (TCP)

The purpose of TCP is to provide highly reliable, host-to-host communication. To achieve this, TCP manages several important issues including basic data transfer, reliability, flow control, multiplexing, connections, and precedence and security. The sending and receiving TCP modules work together to achieve the level of service mandated by the TCP protocol. The sending TCP module packages octets of data into segments, which it forwards to the network layer and the Internet Protocol (IP) for further transmission. TCP tags each octet with a sequence number. The receiving TCP module signals an acknowledgement when it receives each segment and orders the octets according to sequence number, eliminating duplicates and properly handling those that may have been received out of order. In short — TCP guarantees data delivery and saves you the worry.

User Datagram Protocol (UDP)

UDP is used to send and receive data as quickly as possible without the overhead incurred when using TCP. UDP is an extremely lightweight protocol when compared with TCP. It provides direct access to the IP datagram level. However, the quick data transmission provided by UDP comes at a price. Data is not guaranteed to arrive at its intended destination when sent via UDP.

Now, you might ask yourself, “Self, what’s UDP good for?” Generally speaking, any application that needs to send data quickly and doesn’t particularly care about lost datagrams might stand to benefit from using UDP. Examples include data streams where previously sent data is of little or no use because of its age. (i.e., stock market quote streams, voice transmissions, etc.)

Network Layer

The network layer is responsible for the routing of data traffic between internet hosts. These hosts may be located on a local area network or on another network somewhere on the Internet. The Internet Protocol (IP) resides at this layer and provides data routing services to the transport layer protocols TCP or UDP.

Internet Protocol (IP)

The Internet Protocol (IP) is a connectionless service that permits the exchange of data between hosts without a prior call setup. (Hence the term connectionless.) It packages data submitted by TCP or UDP into blocks called datagrams. IP uses IP addresses and routing tables to properly route datagrams to their intended destination networks.

Data Link And Physical Layers

The data link and physical layers are the lowest layers of the networking protocol stack. It is here that data is placed “on the wire” for transmission across the LAN or across the world.

The Data Link Layer

The data link layer sits below the network layer and is responsible for the transmission of data across a particular communications link. It provides for flow control and error correction of transmitted data. An example protocol that operates at the data link layer is Ethernet.

The Physical Layer

The physical layer is responsible for the actual transmission of data across the physical communication lines. Physical layer protocols concern themselves with the types of signals used to transmit data. (i.e., electrical, optical, etc.) and the type of media used to convey the signals (i.e., fiber optic, twisted pair, coaxial, etc.).

Putting It All Together

Computers that participate in a TCP/IP networking environment must be running an instance of the TCP/IP protocol stack as is illustrated in Figure 12.

Referring to Figure 12 — when Host Computer A sends data to Host Computer B via the Internet, the data is passed from the application layer to the physical layer on Host Computer A and sent to the gateway that links the two subnetworks. At the gateway the packets are passed back up to the network layer to determine the forwarding address, then repackaged and sent to the destination computer. When the packets arrive at Host Computer B they are passed back up the protocol stack and the original data is presented to the application layer.

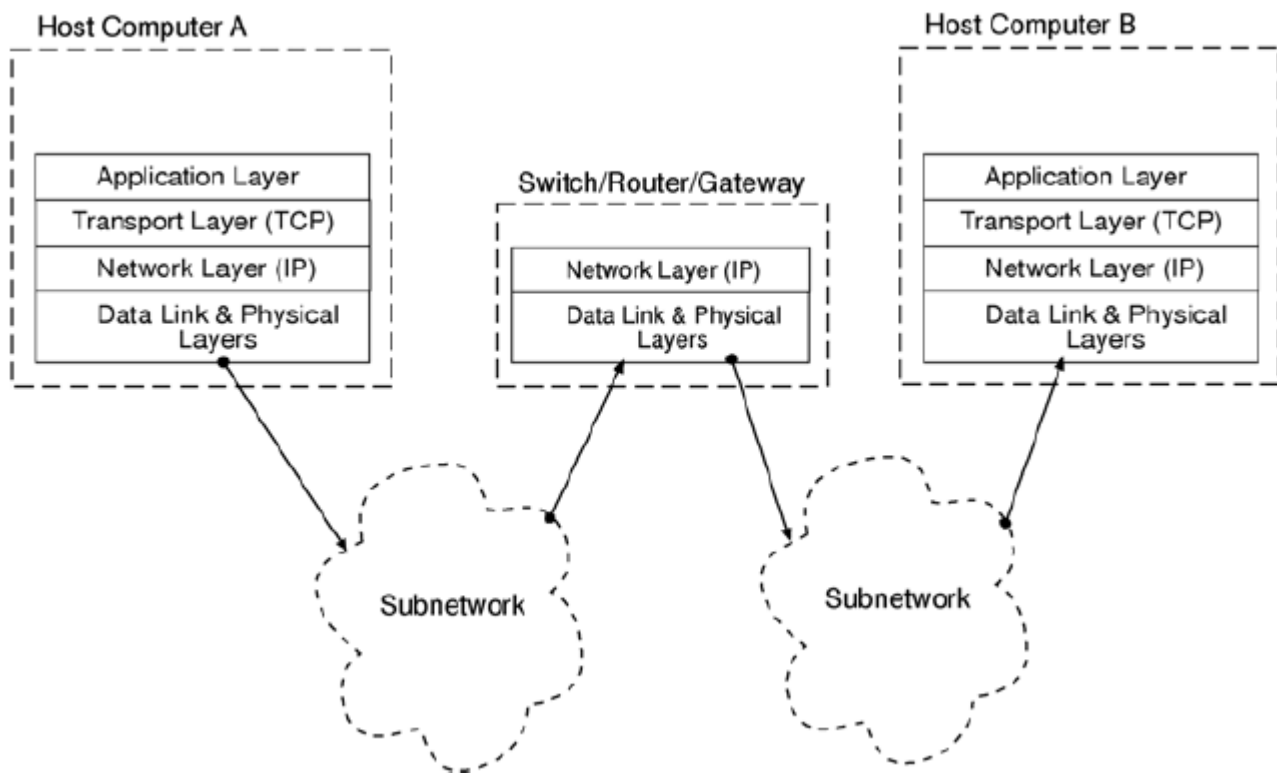


Figure 12: Internet Protocol Stack Operations

What You Need To Know

Now that you have some idea of what's involved with moving data between host computers on the Internet or on a local area network using the Internet protocols, you can pretty much forget about all these nasty details. The .NET Framework provides a set of classes in the System.Net namespace that makes network programming easy.

Chapter 2: Networked Client-Server Applications

Concepts of chapter:

2.1 Introduction	18
2.2 Transmission Control Protocol (TCP)	18
2.3 TCP/IP Client-Server Overview	20

Chapter two Networked Client-Server Applications

Introduction:

You're going to have a lot of fun in this chapter. It is here that you'll put into practical use many of the networking concepts discussed in the previous chapter. you'll learn how to create client-server applications using the `TcpListener`, `TcpClient`, and other classes found in the `System.Net` namespace. I'll show you how to create a multithreaded server application that can service requests from multiple clients. I'll also show you how to create a custom application protocol so your client-server applications can talk to each other.

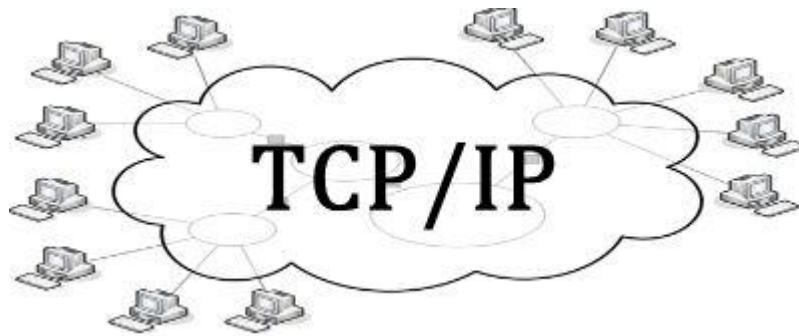
Client-Server Applications With `TcpListener` And `TcpClient`

In this section you'll get a little more down in the weeds with network programming by using the `TcpListener` and `TcpClient` classes to create client-server applications. Unlike .NET remoting, you'll need to know how to handle the details of establishing a network connection between client and server applications, how to send data between the client and server so they can perform useful work, and how to use threads to enable a server to handle multiple client requests simultaneously.

Transmission Control Protocol(TCP):

TCP (Transmission Control Protocol) is a transport layer protocol in the TCP/IP protocol suite (another common transport protocol in TCP/IP is UDP (User Datagram Protocol)). It is a layered protocol suite that uses a strong DoD model. DoD is a four layered model that consists of Link Layer, Internet Layer, Transport Layer, and Application Layer. TCP and IP protocols are the core protocols of TCP/IP (therefore its name TCP/IP). Here is a list of some common protocols in TCP/IP layers from top to down:

- Application Layer: HTTP, FTP, SMTP...
- Transport Layer: TCP, UDP...
- Internet Layer: IP, ARP, ICMP...
- Link Layer: Ethernet, Token Ring...



Also, all your TCP based applications stands on Application Layer. There are many other protocols on each layer. TCP provides reliable, ordered delivery of a stream of bytes from a program on one computer to another program on another computer. TCP is a connection-oriented, asynchronous, and double-way communication protocol (see figure below). Let's explain these concepts:

- **Ordered delivery of a stream of bytes:** In TCP, applications communicate over byte streams. Minimum data transfer size is one byte. The first sent byte first reaches the destination (FIFO queue, ordered delivery). There are two independent communication streams (double-way communication). Applications can send data anytime, independent from each other (asynchronous communication).
- **Reliability:** TCP uses a sequence number to identify each byte of data. The sequence number identifies the order of the bytes sent from each computer so that the data can be reconstructed in order, regardless of any fragmentation, disordering, or packet loss that may occur during transmission. So, if the physical connection with the remote application does not fail and you don't get an exception, your data is almost **guaranteed** to be delivered. Even if a single byte of your data is not delivered (because of an error), subsequent bytes are not delivered before it. If your data reaches destination, you can be sure that it is not corrupted and not deficient.

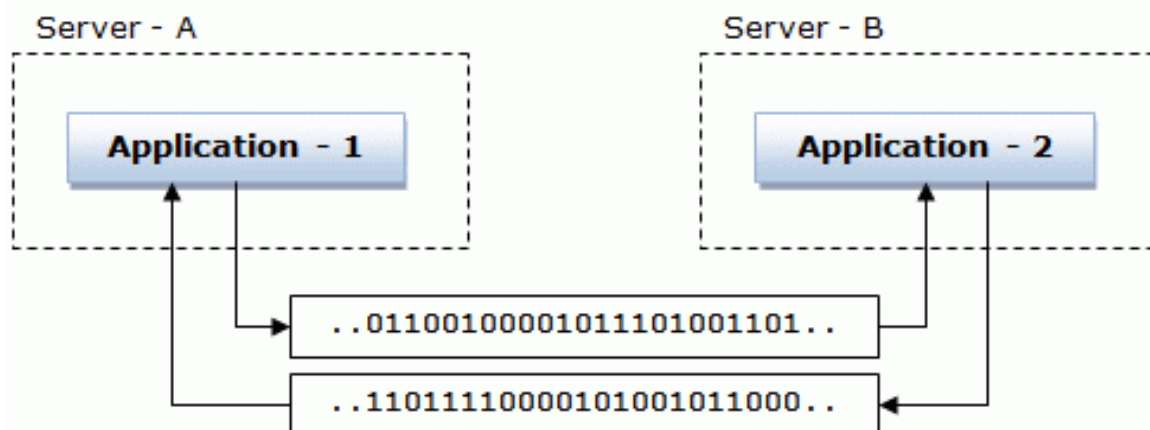


Fig. 1:Stream based double-way TCP communication

Like all communication protocols, in TCP, each application has a unique address. This address consists of an **IP address and a TCP port**.

IP address (for IPv4) is a four octet number separated by dots, like 85.112.23.219. All computers that are connected to the internet have an IP addresses. (If you are connected to the internet behind a router etc., you may not have a public IP address, but your router has one and your computer has a local network IP, thus the router dynamically uses NAT to allow you to use TCP/IP. Anyway, a remote application can send IP datagrams to your computer.)

TCP port is just a number (2 byte unsigned integer, therefore a TCP port can be 0 to 65536) that allows TCP to distinguish your application from other applications in the same computer.

The last TCP concept that I will mention about is **the socket**. A socket is an interface to send data to and receive data from the TCP/IP stack that is provided by the Operating System. In .NET, a socket is represented by a **Socket** object. A Socket object is used to send/receive data between two TCP endpoints. A special kind of socket is the **Listener Socket**. In server/client architecture, the server first opens a **TCP listener socket** to

allow clients to connect to the server. After communication is established with the client, a dedicated socket object is created with the client and all communication with that client is made on this new socket.

Why TCP?

Communication between two applications is generally called as **IPC** (Inter-Process Communication). There are many ways of IPC. TCP/IP is one of them. The advantage of TCP/IP is that the communicating applications can be running on different computers and different locations. Since Internet also works on TCP/IP, remote applications can communicate over the **internet**. TCP/IP is completely **platform independent**, standard, and implemented by all operating systems (and many other devices). Also, it can be used for communication of applications that is on the same computer. Surely, other techniques such as Named Pipes and Shared Memory can be used for communication of two applications but they have an important restriction that the two applications must run on the same computer (or on same network at the best case) and can not communicate over the internet.

What are the stages of building a communication framework upon TCP?

In this section, I will discuss how to make a plan to build a communication framework over TCP and the problems to be considered.

Stream based data transfer

From the perspective of TCP, everything that is being sent from an application to another are **bytes of a stream**. TCP does not know if it is bytes of a picture, a file, a string or a serialized object. Also TCP does not know if a data (say a serialized object) has been sent and we are now sending another data (I mention about **message boundaries**).

Messaging over streams

From the application perspective, the data transferred over a TCP stream are **separated packets**. I call it **messages**. An application wants to send messages to another application when needed. Also, an application wants to be **informed** (via some callback mechanism) when a new message is received from a remote application.

So, we must build a mechanism over TCP streams to write a message to a stream and read a message from a stream. We must provide a way of separating messages from each other.

Wire protocol (message to stream protocol)

The way of writing a message to a stream and reading a message from a stream is called **wire protocol**. A wire protocol has some knowledge about the message that is being sent and received. In .NET, if we represent a message by an object, we can use **serialization** to build a **byte array** from the object. We can use **binary serialization**, **XML serialization** (and then string to byte array encoding), or a **custom serialization**. Anyway, after creating a byte array from a message, we can write it to a TCP stream. Also, we can use a kind of **deserialization** to re-create an object from a TCP stream.

If we create a custom wire protocol that can be implemented in any other programming language or platform, we can build a **platform independent** system. But if we use .NET binary serialization, our application can only be used in .NET world.

Request/Reply style messaging

In most scenarios, an application **sends a message** to a remote application and waits for a **response message**. Because of the **asynchronous** nature of TCP, this is an important implementation detail. In this case, we must use **multithreading** tools (such as ManualResetEvent class in .NET) to block/wait a sender thread until a response is received and to notify when a response is received. We also must provide a way of matching **request and reply messages**. Last but not least, we must provide a **timeout mechanism** if the remote application does not send a response message within a specified time.

Mapping messages with classes and methods

This is the most important and distinctive feature of a TCP based communication library from a user (that uses the framework to build applications) perspective. Many different approaches can be preferred according to the needs of the user, capabilities of the programming language or platform etc... Let's discuss a few approaches here.

The first approach is **message-oriented communication**. In this way, the sender application sends an object from a special type of class using serialization (as mentioned before), and the receiver application deserializes the object, and enters a switch or if statement to call a method according to the properties of the object. Message classes can be derived from a base class and can define common methods. Even, message objects can be a simple string. Applications parse this string and performs the needed operations. The disadvantage of this approach is that the user of the library must define his own messaging logic and write switch statements. If a new message type is added to the system, many classes, methods, or code blocks must be changed. Also, it is not type safe, and messages are resolved at runtime. So an application can send unknown message types etc.

3.TCP/IP Client-Server Overview

The steps required to write a TCP/IP client-server application using the System.Net.TcpListener and System.Net.TcpClient classes are highlighted in the following illustrations.

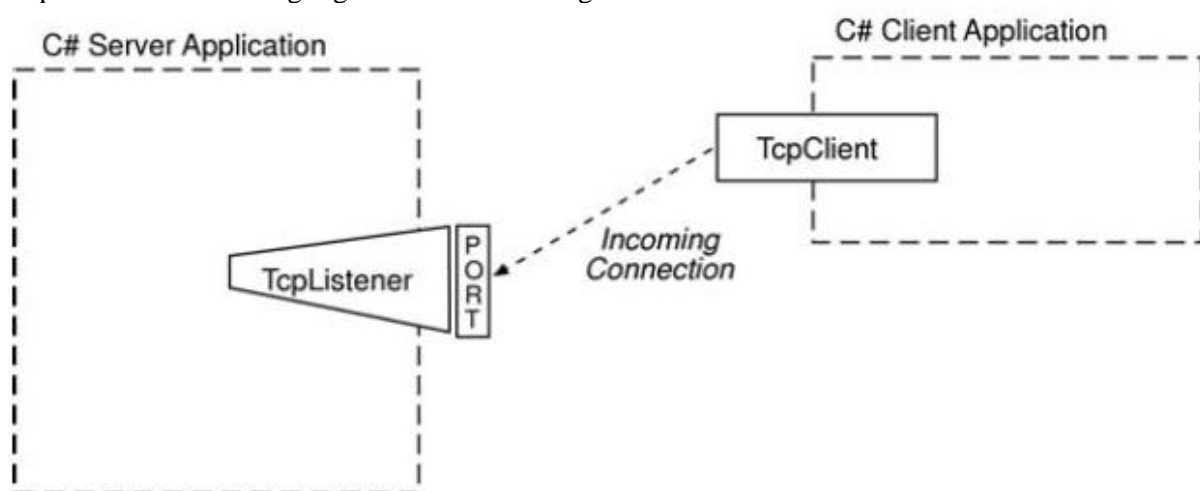


Figure 2: Server Application Listens on a Host and Port for Incoming TcpClient Connections

Referring to Figure 2 — a server application uses a TcpListener object to listen for incoming TcpClient connections on a particular IP address (or multiple IP addresses) and port number. The client application uses a TcpClient object to connect to a particular machine, given its IP address or DNS name (i.e., www.warrenworks.com) which is then mapped to an IP address, and specified port number.

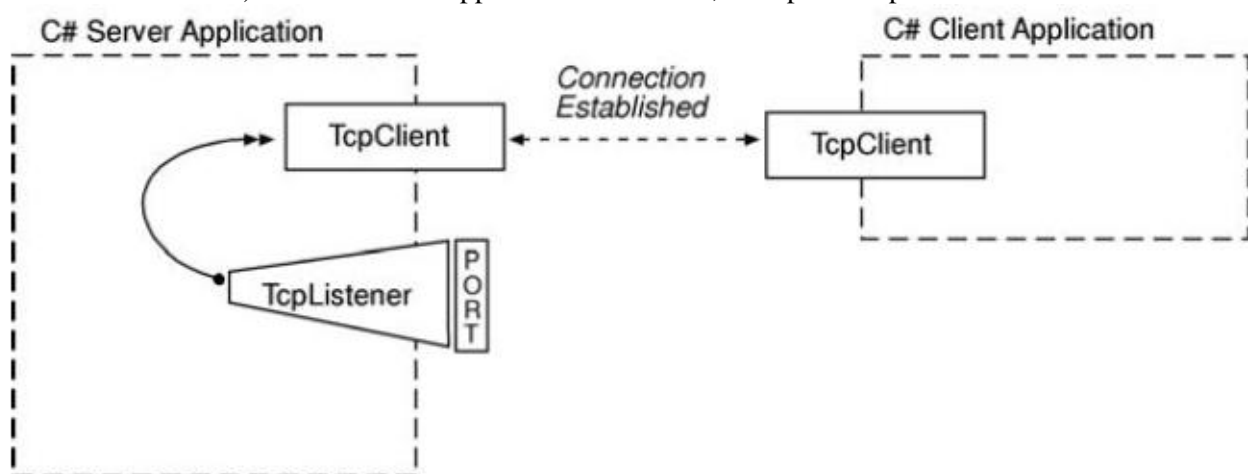


Figure 3: TcpListener Accepts Incoming TcpClient Connection

Referring to Figure 3 — when the TcpListener object detects an incoming TcpClient connection, it “accepts” the connection, which results in the creation of a server-side Tcpclient object. Client-server communication

takes place between the server-side and client-side `TcpClient` objects, as is shown in Figure 4. Both the `TcpListener` and `TcpClient` objects provide wrappers around socket objects. You could use socket objects directly to conduct client-server communication.

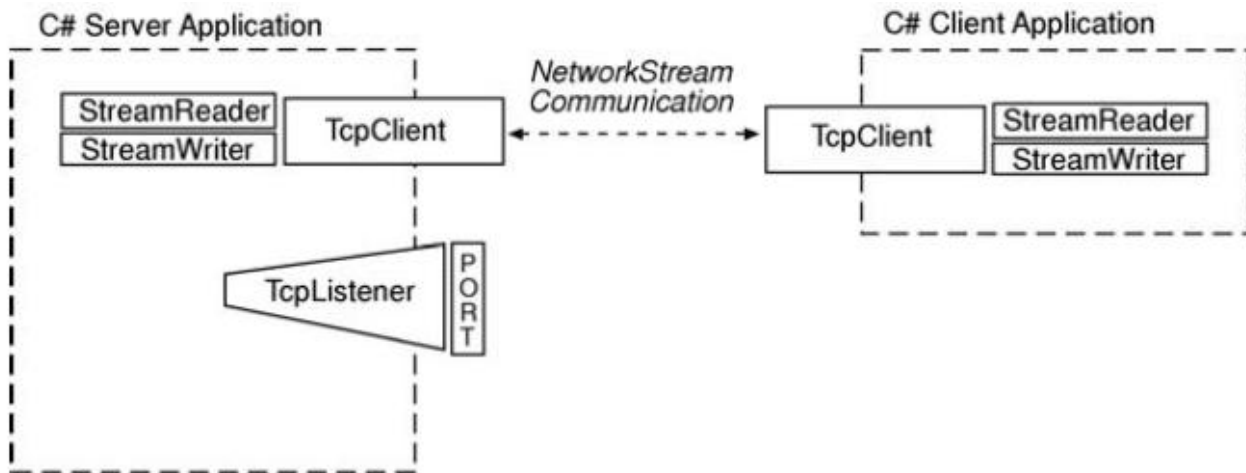


Figure 4: `TcpClients` Communicate via a `NetworkStream` using `StreamReader` and `StreamWriter` Objects

A Simple Client-Server Application

OK, let's put some of what you just learned in the previous section into practical use. The following examples implement a simple client-server application using the `TcpListener` and `TcpClient` classes. The application consists of two parts: an `EchoServer`, which listens for incoming `TcpClient` connections, and an `EchoClient` which connects to an `EchoServer`. When a connection between the `EchoServer` and `EchoClient` is established, messages sent from the client to the server are written to the server console and then sent back to the client for display on the client console. Figure 5 gives the code for the `EchoServer` application.

```

1  using System;
2  using System.IO;
3  using System.Net;
4  using System.Net.Sockets;
5
6  public class EchoServer {
7  public static void Main(){
8  TcpListener listener = null;
9  try {
10     listener = new TcpListener(IPAddress.Parse("127.0.0.1"), 8080);
11     listener.Start();
12     Console.WriteLine("EchoServer started...");
13     while(true){
14         Console.WriteLine("Waiting for incoming client connections...");
15         TcpClient client = listener.AcceptTcpClient();
16         Console.WriteLine("Accepted new client connection...");
17         StreamReader reader = new StreamReader(client.GetStream());
18         StreamWriter writer = new StreamWriter(client.GetStream());
19         String s = String.Empty;
20         while(!(s = reader.ReadLine()).Equals("Exit")){
21             Console.WriteLine("From client -> " + s);
22             writer.WriteLine("From server -> " + s);
23             writer.Flush();
24         }

```

```

25         reader.Close();
26         writer.Close();
27         client.Close();
28     }
29 } catch (Exception e) {
30     Console.WriteLine(e);
31 } finally {
32     if (listener != null) {
33         listener.Stop();
34     }
35 }
36 } // end Main()
37 } // end class definition

```

Figure 5: Source code of the server.

Referring to Figure 5 — notice first the list of namespaces required for this particular application. It includes System.IO, System.Net, and System.Net.Sockets.

The EchoServer application starts by creating an instance of TcpListener, which listens on the local machine IP address of 127.0.0.1 port 8080. (Make sure the port you choose is not in use.) The listener is then started on line 11 by a call to its Start() method. Incoming client connections are processed in the body of the while loop, which begins on line 13.

On line 15, the listener.AcceptTcpClient() method blocks at that point until it detects an incoming TcpClient connection, at which time it unblocks and returns an instance of TcpClient and assigns it to the client reference. The term block refers to a blocking I/O operation. The EchoServer application effectively stops everything until the AcceptTcpClient() method returns, at which time processing continues.

When the listener detects the incoming TcpClient connection, the application prints a short message stating so to the console, and then, on lines 17 and 18, it creates StreamReader and StreamWriter objects using the client.GetStream() method. The server uses these StreamReader and StreamWriter objects to communicate with the client. On line 19, the application creates a string variable named s and uses it to store incoming client strings. The body of the while loop, which begins on line 20, processes client-server communication by reading the incoming client string, printing it to the server's console, and then sending it back to the client via the writer.WriteLine() method. Note on line 23 the writer.Flush() method must be called to actually send the string on its way. The while loop repeats until the incoming string equals "Exit", at which time the EchoServer returns to listening for new incoming TcpClient connections.

Figure 6 gives the code for the EchoClient application.

```

1 using System;
2 using System.IO;
3 using System.Net;
4 using System.Net.Sockets;
5
6 public class EchoClient {
7     public static void Main() {
8         try {
9             TcpClient client = new TcpClient("127.0.0.1", 8080);
10            StreamReader reader = new StreamReader(client.GetStream());
11            StreamWriter writer = new StreamWriter(client.GetStream());
12            String s = String.Empty;
13            while (!s.Equals("Exit")) {
14                Console.Write("Enter a string to send to the server: ");
15                s = Console.ReadLine();
16                Console.WriteLine();
17                writer.WriteLine(s);
18                writer.Flush();
19                String server_string = reader.ReadLine();
20                Console.WriteLine(server_string);

```

```

21     }
22     reader.Close();
23     writer.Close();
24     client.Close();
25 } catch(Exception e){
26     Console.WriteLine(e);
27 }
28 } // end Main()
29 } // end class definition

```

Figure 6: Source code of the client.

Referring to Figure 6 — the EchoClient application creates a TcpClient object that **connects** to the IP address 127.0.0.1 port 8080 (the server). If all goes well, lines 10 and 11 execute and the application creates the StreamReader and StreamWriter objects, which it uses to communicate with the server.

On line 12, a string variable named s is created and used to send data to the server and to control the processing of the while loop, which starts on the following line.

On line 15, the Console.ReadLine() method reads a line of text from the console and assigns it to s. On lines 17 and 18, it sends the string s to the server with calls to writer.WriteLine() and writer.Flush(). It then immediately reads the server's response with a call to the reader.ReadLine() method, which assigns the incoming string to the string variable named server_string and then prints the value of server_string to the console. The EchoServer application repeats this processing loop until the user enters the string "Exit" at the console.

To run this application, compile the EchoServer.cs and EchoClient.cs files, start the EchoServer, then run the EchoClient application. Figure 19-11 shows the results of running these applications.

Server/client architecture and multithreading:

TCP sockets can be used in **blocking (synchronous) or nonblocking (asynchronous)** mode. In the blocking mode, **receiving and sending** data is a blocking operation and sender/receiver thread is blocked until operation is completed. Especially in blocking receiving, the thread waits until a data is sent from remote application.

So, we must make use of **multithreading** to do another operations when waiting data from remote application. This is not a major problem in client side since only one socket is used and one additional thread is enough to listen incoming data. But in server side, there is a dedicated socket for every clients. So, if there are 100 clients connected concurrently, there are 100 sockets to communicate with clients and 100 threads to wait incoming data from clients in blocking mode.

Blocking mode is easy to implement but not scalable for the reasons mentioned above. So, we must use asynchronous sockets with callback mechanism and a thread pool to process data.

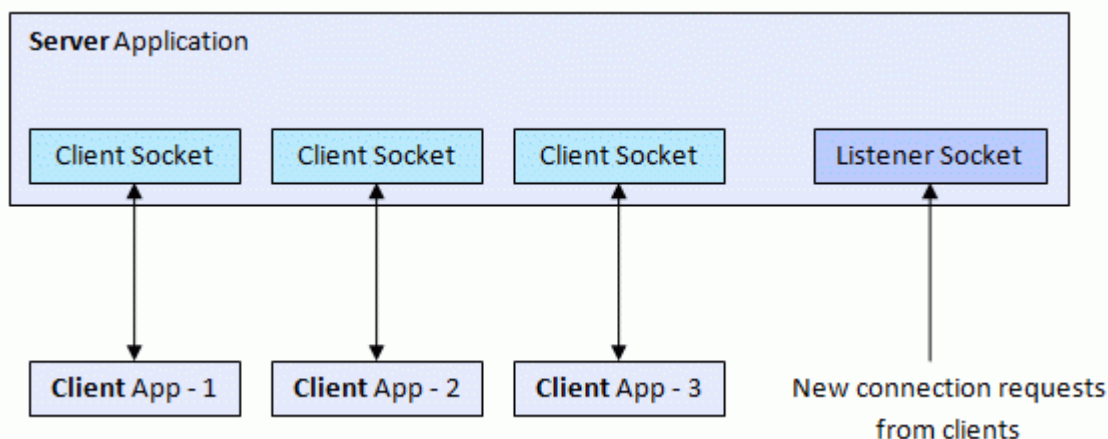


Figure 7: Simple TCP server/client architecture

While the previous example served well to illustrate basic client-server principles, the server application in its current form is only able to communicate **with one client at a time**. In this section, I'll show you how to make a few modifications to EchoServer that will enable it to serve multiple clients simultaneously. A server application that can process multiple simultaneous client connections is referred to as a **multithreaded server**. The following general steps are required to turn EchoServer into a MultiThreadedEchoServer:

Step 1: Create a separate client processing method that handles network stream communication and other applicable processing between server and client applications.

Step 2: For each incoming client connection, spawn a separate thread, passing to it the name of the client processing method.

That's it! Let's see how these modifications look in the code. Figure 8 gives the code for the MultiThreadedEchoServer class.

```
1 using System;
2 using System.IO;
3 using System.Net;
4 using System.Net.Sockets;
5 using System.Threading;
6
7 public class MultiThreadedEchoServer {
8
9     private static void ProcessClientRequests(Object argument){
10         TcpClient client = (TcpClient)argument;
11         try {
12             StreamReader reader = new StreamReader(client.GetStream());
13             StreamWriter writer = new StreamWriter(client.GetStream());
14             String s = String.Empty;
15             while(!(s = reader.ReadLine()).Equals("Exit")){
16                 Console.WriteLine("From client -> " + s);
17                 writer.WriteLine("From server -> " + s);
18                 writer.Flush();
19             }
20             reader.Close();
21             writer.Close();
22             client.Close();
23             Console.WriteLine("Closing client connection!");
24         } catch(IOException){
25             Console.WriteLine("Problem with client communication. Exiting thread.");
26         } finally{
27             if(client != null){
28                 client.Close();
29             }
30         }
31     }
32
33     public static void Main(){
34         TcpListener listener = null;
35         try {
36             listener = new TcpListener(IPAddress.Parse("127.0.0.1"), 8080);
37             listener.Start();
38             Console.WriteLine("MultiThreadedEchoServer started...");
39             while(true){
40                 Console.WriteLine("Waiting for incoming client connections...");
41                 TcpClient client = listener.AcceptTcpClient();
42                 Console.WriteLine("Accepted new client connection...");
43                 Thread t = new Thread(ProcessClientRequests);
44                 t.Start(client);
45             }
46         }
```

```

46     }catch(Exception e){
47         Console.WriteLine(e);
48     }finally{
49         if(listener != null){
50             listener.Stop();
51         }
52     }
53 } // end Main()
54 } // end class definition

```

Figure 8: the code for the `MultiThreadedEchoServer` class.

Referring to Figure 8— first, a new namespace, `System.Threading`, has been added to list of using directives to gain access to the `Thread` class. I created a new method on line 9 named `ProcessClientRequests()`. Note that this method takes one argument of type `Object`, which is cast immediately to a `TcpClient` object. The reason this cast is necessary is because the `ProcessClientRequests()` method has the signature of a `ParameterizedThreadStart` delegate, which specifies one argument of type `Object`. You'll see how this method is actually used in the body of the `Main()` method.

I copied the bulk of the **`ProcessClientRequests()`** method from the previous version of `EchoClient` starting with the creation of the `StreamReader` and `StreamWriter` objects. It includes the whole of the second, or inner, while loop. I enclosed the method's code within its own **`try/catch/finally`** block because once the separate thread begins execution, it must handle any exceptions it generates.

In the body of the `Main()` method, the `TcpListener` object is created as before on line 36 and is started on line 37. The while loop beginning on line 39 repeats forever waiting for incoming client connections. When it detects an incoming client connection, the `AcceptTcpClient()` method returns a reference to a new `TcpClient` object and processing continues with the **creation of a new Thread object on line 43**. The name of the method this thread will execute, **`ProcessClientRequests`**, is passed to the `Thread` constructor.

The new thread is started with a call to `t.Start()` on line 44, passing to it the reference to the `TcpClient` object named `client`. When line 44 completes execution, the while loop continues and the server returns to listening for incoming client connections.

You now have a multithreaded server application! The code for `EchoClient`, given in the previous section, remains unchanged. Figure 19-12 shows the results of running the `MultiThreadedServer` and connecting to it from two `EchoClient` applications.

Sending Objects Between Client And Server

In the previous examples, I've limited the exchange between client and server application to strings. In this section I'll show you how to serialize a complex object on the server side and send it to the client for deserialization. Remember that in the case of .NET remoting applications, the hard work of serializing complex objects is done for you by the remoting framework. Not here, no, no, no. If you want to serialize a complex object and send it across the network you'll need to get your hands dirty.

The following two examples implement a `SurrealistEchoServer`. The application actually consists of three classes: `SurrealistEchoServer.cs`, `SurrealistDB.cs`, and `Person.cs`, which is not repeated here. Figure 9 gives the code for the `SurrealistEchoServer` class.

```

1 using System;
2 using System.Drawing;
3 using System.IO;
4 using System.Net;
5 using System.Net.Sockets;
6 using System.Net.NetworkInformation;
7 using System.Threading;
8 using System.Runtime.Serialization;
9 using System.Runtime.Serialization.Formatters.Binary;
10
11 public class SurrealistEchoServer {
12
13     private static void ProcessClientRequests(Object argument){

```

```

14  TcpClient client = (TcpClient)argument;
15  try {
16  StreamReader reader = new StreamReader(client.GetStream());
17  StreamWriter writer = new StreamWriter(client.GetStream());
18  String s = String.Empty;
19  while(!(s = reader.ReadLine()).Equals("Exit")){
20      switch(s){
21          case "GetSurrealists" : {
22              Console.WriteLine("From client -> " + s);
23              SerializeSurrealists(client.GetStream());
24              client.GetStream().Flush();
25              break;
26          }
27          default: {
28              Console.WriteLine("From client -> " + s);
29              writer.WriteLine("From server -> " + s);
30              writer.Flush();
31              break;
32          }
33      } // end switch
34  } // end while
35  reader.Close();
36  writer.Close();
37  client.Close();
38  Console.WriteLine("Client connection closed!");
39  }catch(IOException){
40      Console.WriteLine("Problem with client communication. Exiting thread.");
41  }catch(NullReferenceException){
42      Console.WriteLine("Incoming string was null! Client may have terminated prematurely.");
43  }catch(Exception e){
44      Console.WriteLine("Unknown exception occurred.");
45      Console.WriteLine(e);
46  }finally{
47      if(client != null){
48          client.Close();
49      }
50  }
51  } // end ProcessClientRequests()
52
53 private static void SerializeSurrealists(NetworkStream stream){
54     SurrealistDB db = new SurrealistDB();
55     BinaryFormatter bf = new BinaryFormatter();
56     bf.Serialize(stream, db.GetSurrealists());
57     } // end SerializeSurrealists()
58
59 private static void ShowServerNetworkConfig() {
60     Console.ForegroundColor = ConsoleColor.Yellow;
61     NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
62     foreach(NetworkInterface adapter in adapters){
63         Console.WriteLine(adapter.Description);
64         Console.WriteLine("\tAdapter Name: " + adapter.Name);
65         Console.WriteLine("\tMAC Address: " + adapter.GetPhysicalAddress());
66         IPInterfaceProperties ip_properties = adapter.GetIPProperties();
67         UnicastIPAddressInformationCollection addresses = ip_properties.UnicastAddresses;
68         foreach(UnicastIPAddressInformation address in addresses){
69             Console.WriteLine("\tIP Address: " + address.Address);
70         }

```

```

71     }
72     Console.ForegroundColor = ConsoleColor.White;
73 } // end ShowServerNetworkConfig()
74
75 public static void Main(){
76     TcpListener listener = null;
77     try {
78         ShowServerNetworkConfig();
79         listener = new TcpListener(IPAddress.Any, 8080);
80         listener.Start();
81         Console.WriteLine("SurrealistEchoServer started...");
82         while(true){
83             Console.WriteLine("Waiting for incoming client connections...");
84             TcpClient client = listener.AcceptTcpClient();
85             Console.WriteLine("Accepted new client connection...");
86             Thread t = new Thread(ProcessClientRequests);
87             t.Start(client);
88         }
89     } catch(Exception e){
90         Console.WriteLine(e);
91     } finally{
92         if(listener != null){
93             listener.Stop();
94         }
95     }
96 } // end Main()
97 } // end class definition

```

Figure 9: the code for the SurrealistEchoServer class.

Referring to Figure 9— first, note the addition of several namespaces required to perform object serialization. These include `System.Runtime.Serialization` and `System.Runtime.Serialization.Formatters.Binary`. The `SurrealistEchoServer` class's `ProcessClientRequests()` method has been slightly modified. It echoes client strings as before, but if the client string equals "GetSurrealists", it returns to the client a serialized collection of `Person` objects. It does this with a call to its `SerializeSurrealists()` method, which begins on line 53.

The `SerializeSurrealists()` method takes a `NetworkStream` object as an argument. On line 54, it creates an instance of `SurrealistsDB` followed by the creation of a `BinaryFormatter` object on the next line. The `BinaryFormatter` serializes the `List<Person>` object returned by the `db.GetSurrealists()` method into the stream. When the `SerializeSurrealists()` method returns, the network stream is flushed to send the collection of `People` objects on their way to the client.

Figure 10 gives the code for the `SurrealistDB` class.

```

1     using System;
2     using System.Collections.Generic;
3
4     public class SurrealistDB {
5
6         private List<Person> surrealists = null;
7
8         public SurrealistDB(){
9             this.InitializeSurrealists();
10        }
11
12        public List<Person> GetSurrealists(){
13            return surrealists;
14        }
15

```

```

16 private void InitializeSurrealists(){
17     surrealists = new List<Person>();
18     Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
19     Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
20     Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
21     Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
22     Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
23     Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
24         new DateTime(1887, 07, 28));
25
26     surrealists.Add(p1);
27     surrealists.Add(p2);
28     surrealists.Add(p3);
29     surrealists.Add(p4);
30     surrealists.Add(p5);
31     surrealists.Add(p6);
32 }
33 } // end class definition

```

Figure 10: The code for the SurrealistDB class.

Referring to Figure 10 — The SurrealistDB class initializes a list of People objects and provides a GetSurrealists() method which returns the populated list. (Note: This class could easily have been written to connect to a data base to fetch the required information. You'll see how that's done in the next Chapter). The EchoClient class must be modified to accept and deserialize the incoming list of People objects. Figure 11 gives the code for the modified EchoClient class.

```

1     using System;
2     using System.IO;
3     using System.Net;
4     using System.Net.Sockets;
5     using System.Runtime.Serialization;
6     using System.Runtime.Serialization.Formatters.Binary;
7     using System.Collections.Generic;
8
9     public class EchoClient {
10         static List<Person> DeserializeSurrealists(NetworkStream stream){
11             BinaryFormatter bf = new BinaryFormatter();
12             return (List<Person>)bf.Deserialize(stream);
13         }
14         static void WriteSurrealistDataToConsole(List<Person> surrealists){
15             foreach(Person p in surrealists){
16                 Console.WriteLine(p);
17             }
18         }
19         public static void Main(String[] args){
20             IPAddress ip_address = IPAddress.Parse("127.0.0.1"); //default
21             int port = 8080;
22             try{
23                 if(args.Length >= 1){
24                     ip_address = IPAddress.Parse(args[0]);
25                 }
26             }catch(FormatException){
27                 Console.WriteLine("Invalid IP address entered. Using default IP of: " +
28                     ip_address.ToString());
29             }
30         }
31     }

```

```

34     Console.WriteLine("Attempting to connect to server at IP address: {0} port: {1}",
35     ip_address.ToString(), port);
36     TcpClient client = new TcpClient(ip_address.ToString(), port);
37     Console.WriteLine("Connection successful!");
38     StreamReader reader = new StreamReader(client.GetStream());
39     StreamWriter writer = new StreamWriter(client.GetStream());
40     String s = String.Empty;
41     while(!s.Equals("Exit")){
42         Console.Write("Enter \"GetSurrealists\" to retrieve list from server: ");
43         s = Console.ReadLine();
44         Console.WriteLine();
45         switch(s){
46             case "GetSurrealists" : {
47                 writer.WriteLine(s);
48                 writer.Flush();
49                 WriteSurrealistDataToConsole(DeserializeSurrealists(client.GetStream()));
50                 Console.WriteLine();
51                 break;
52             }
53             case "Exit" : {
54                 writer.WriteLine(s);
55                 writer.Flush();
56                 break;
57             }
58             default: {
59                 writer.WriteLine(s);
60                 writer.Flush();
61                 String server_string = reader.ReadLine();
62                 Console.WriteLine(server_string);
63                 Console.WriteLine();
64                 break;
65             }
66         }
67     }
68     reader.Close();
69     writer.Close();
70     client.Close();
71 }catch(Exception e){
72     Console.WriteLine(e);
73 }
74 } // end Main()
75 } // end class definition

```

Figure 11: The code for the modified EchoClient class.

Referring to Figure 11 — the modified EchoClient application sends strings to the server as before. When the string it sends equals "GetSurrealists" the server returns a serialized list of People objects. (i.e., List<People>) The client must then deserialize the object and cast it to its expected type, which it does with the DeserializeSurrealists() method. Once the list of People objects is deserialized, the EchoClient application calls the WriteSurrealistData-ToConsole() method. All this action takes place on line 49!

To run these applications, copy the Person.dll into both client and server directories, then change to the server directory and compile the server application using the following compiler commands. First compile the SurrealistDB class into a dll:

```
csc /t:library /r:Person.dll SurrealistDB.cs
```

Then compile the server itself:

```
csc /r:Person.dll;SurrealistDB.dll SurrealistEchoServer.cs
```

Change to the client directory and compile the EchoClient class like so:

```
csc /r:Person.dll EchoClient.cs
```

Finally, start the SurrealistEchoServer application and then run the EchoClient application. Figure 12 gives the results of fetching some surrealists from the server.

Chapter 3: Encrypting Data in Network Connections

Contents of Chapter:

3.1 Introduction.....	31
3.2 what is encryption.....	31
3.3 history of encryption.....	32
3.4 how encryption work.....	33
3.5 A symmetric encryption.....	33
3.6 symmetric encryption.....	34
3.7 data encryption standard.....	36
3.8 RSA algorithm.....	46
3.9 creating a symmetric encryption stream.....	48
3.10 SSL stream class.....	52
3.11 project.....	60

Chapter 3 Encrypting Data in Network Connections

1.Introduction

In today's computing environment, creating applications that transfer data between devices on networks has become a necessity for programmers. Fortunately, Microsoft has included several classes in the .NET Framework that make network programming easy. The TcpClient, TcpListener, and NetworkStream classes are popular classes that provide all the functionality necessary to pass data across any network.

However, sending data across networks can be a risky business. There are lots of prying eyes watching packets as they traverse the network. This can be a serious problem if your application handles sensitive information such as employee addresses and phone numbers. Most commercial applications incorporate some type of encryption technique to hide the data

as it is sent over the network. While encrypting data is not 100% foolproof, it adds a basic level of protection to the data.

The .NET Framework provides several cryptographic classes that can easily be incorporated into your network programs to help disguise your data as it travels across the network. This project demonstrates how to incorporate the cryptographic classes in network applications to create a safer environment for your application data.

2.What is Encryption?

Encryption is the conversion of electronic data into another form, called ciphertext, which cannot be easily understood by anyone except authorized parties.

The primary purpose of encryption is to protect the confidentiality of digital data stored on computer systems or transmitted via the Internet or other computer networks. Modern encryption algorithms play a vital role in the security assurance of IT systems and communications as they can provide not only confidentiality, but also the following key elements of security:

- **Authentication**: the origin of a message can be verified.
- **Integrity**: proof that the contents of a message have not been changed since it was sent.
- **Non-repudiation**: the sender of a message cannot deny sending the message.

3.History of encryption

The word *encryption* comes from the Greek word *kryptos*, meaning hidden or secret. The use of encryption is nearly as old as the art of communication itself. As early as 1900 BC, an Egyptian scribe used non-standard hieroglyphs to hide the meaning of an inscription. In a time when most people couldn't read, simply writing a message was often enough, but encryption schemes soon developed to convert messages into unreadable groups of figures to protect the message's secrecy while it was carried from one place to another. The contents of a message were reordered (transposition) or replaced (substitution) with other characters, symbols, numbers or pictures in order to conceal its meaning.

In 700 BC, the Spartans wrote sensitive messages on strips of leather wrapped around sticks. When the tape was unwound the characters became meaningless, but with a stick of exactly the same diameter, the recipient could recreate (decipher) the message. Later, the Romans used what's known as the Caesar Shift Cipher, a monoalphabetic cipher in which each letter is shifted by an agreed number. So, for example, if the agreed number is three, then the message, "Be at the gates at six" would become "eh dw wkh jdwhv dw vla". At first glance this may look difficult to decipher, but juxtapositioning the start of the alphabet until the letters make sense doesn't take long. Also, the vowels and other commonly used letters like *T* and *S* can be quickly deduced using frequency analysis, and that information in turn can be used to decipher the rest of the message.

The Middle Ages saw the emergence of polyalphabetic substitution, which uses multiple substitution alphabets to limit the use of frequency analysis to crack a cipher. This method of encrypting messages remained popular despite many implementations that failed to adequately conceal when the substitution changed, also known as key progression. Possibly the most famous implementation of a polyalphabetic substitution cipher is the Enigma electro-mechanical rotor cipher machine used by the Germans during World War Two.

It was not until the mid-1970s that encryption took a major leap forward. Until this point, all encryption schemes used the same secret for encrypting and decrypting a message: a symmetric key. In 1976, B.

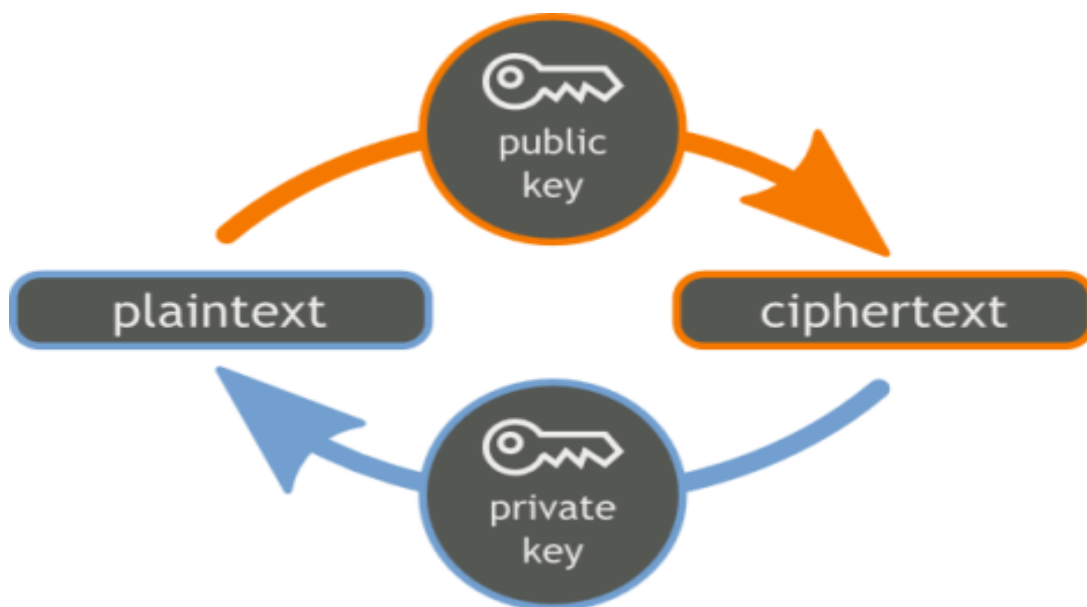
Whitfield Diffie and Martin Hellman's paper *New Directions in Cryptography* solved one of the fundamental problems of cryptography, namely how to securely distribute the encryption key to those who need it. This breakthrough was followed shortly afterwards by RSA, an implementation of public-key cryptography using asymmetric algorithms, which ushered in a new era of encryption.

4.How encryption works

Data, often referred to as plaintext, is encrypted using an encryption algorithm and an encryption key. This process generates ciphertext that can only be viewed in its original form if decrypted with the correct key. Decryption is simply the inverse of encryption, following the same steps but reversing the order in which the keys are applied. Today's encryption algorithms are divided into two categories: symmetric and asymmetric.

Symmetric-key ciphers use the same key, or secret, for encrypting and decrypting a message or file. The most widely used symmetric-key cipher is AES, which was created to protect government classified information. Symmetric-key encryption is much faster than asymmetric encryption, but the sender must exchange the key used to encrypt the data with the recipient before he or she can decrypt it. This requirement to securely distribute and manage large numbers of keys means most cryptographic processes use a symmetric algorithm to efficiently encrypt data, but use an asymmetric algorithm to exchange the secret key.

Asymmetric cryptography, also known as public-key cryptography, uses two different but mathematically linked keys, one public and one private. The public key can be shared with everyone, whereas the private key must be kept secret. RSA is the most widely used asymmetric algorithm, partly because both the public and the private keys can encrypt a message; the opposite key from the one used to encrypt a message is used to decrypt it. This attribute provides a method of assuring not only confidentiality, but also the integrity, authenticity and non-reputability of electronic communications and data at rest through the use of digital signatures.



One of the basic questions in considering encryption is to understand the differences between **symmetric** and **asymmetric encryption** methods, and where to apply each method to best protect your data.

5. Asymmetric encryption

Asymmetric encryption, also known as public-key encryption, utilizes a pair of keys – a public key and a private key. If you encrypt data with the public key, only the holder of the corresponding private key can decrypt the data, hence ensuring confidentiality.

The first practical algorithm for asymmetric encryption was proposed by Diffie and Hellman in 1976. Subsequently, RSA became the most widely deployed asymmetric encryption algorithm.

Many “secure” online transaction systems rely on asymmetric encryption to establish a secure channel. SSL, for example, is a protocol that utilizes asymmetric encryption to provide communication security on the Internet.

Asymmetric encryption algorithms typically involve exponential operations, they are not lightweight in terms of performance. For that reason, asymmetric algorithms are often used to secure key exchanges rather than used for bulk data encryption.

6. Symmetric encryption

Symmetric encryption, as the name suggests, means that the encryption and decryption operations utilize the same key. For two communicating parties using symmetric encryption for secure communication, the key represents a shared secret between the two.

There exist many symmetric encryption algorithms. A few of the well-known ones include AES, DES, Blowfish, and Skipjack.

Symmetric encryption is typically more efficient than asymmetric encryption, and is often used for bulk data encryption.

Encryption Algorithm Types

SYMMETRIC	ASYMMETRIC
AES (AES-128, AES-192, AES-256)	Diffie-Hellman key exchange
Blowfish	RSA asymmetric algorithm
Twofish	SHA-224
DES	SHA-256
3DES	SHA-386
RC4	SHA-512
	SHA-3 (emerging standard)

©2016 TECHTARGET. ALL RIGHTS RESERVED. 

Attack a cryptosystem

Given enough computing resources, both symmetric and asymmetric encryption can be broken.

The most basic way to attack a symmetric cryptosystem is brute-force attacks, where you essentially try every combination of a key. For a 128-bit key, there are 2^{128} combinations to attempt, which requires extensive computing resources. Other cryptanalysis attacks, including chosen-ciphertext and chosen-plaintext attacks, can be more efficient than brute-force, but they require a priori knowledge to work.

To guard against brute-force attacks, the key length of a symmetric cryptosystem needs to be sufficiently long. The **Advanced Encryption Standard (AES)** algorithm with 256-bit key is considered secure enough for most purposes. And the implementation can be made relatively efficient.

The best way to attack a well-designed RSA implementation is through factoring of RSA's public modulus, which is a large number. Factoring large numbers, with today's best known factoring techniques, is a compute-intensive problem.

RSA (the company), ran a [factoring challenge](#) from 1991 to 2007, during which an RSA 768-bit modulus was factored successfully. In 2010, a 1024-bit RSA modulus was factored with relatively low cost.

Today, RSA implementations typically require a 2048-bit key to be secure. For ultra sensitive operations, you would want 4096-bit keys. Of course the longer the key length, the more expensive it is to run the encryption and decryption operations.



Which Method Is Right For You?

How to choose symmetric vs. asymmetric cryptosystems? Here are a few tips:

The case for symmetric-key cryptography

- Symmetric key cryptosystems have been shown to be more efficient and can handle high rates of data throughput
- Keys for symmetric-key cryptosystems are shorter, compared to public key algorithms
- Symmetric key ciphers can be composed together to produce a stronger cryptosystem.

The case for asymmetric-key cryptography

- In a large network, asymmetric key cryptography yields a more efficient system for key management, as you don't have to manage pair-wise keys for every communicating pair.
- Asymmetric key cryptosystems are good for digital signatures and key exchange use cases
- In many cases, the public and private key pairs in an asymmetric-key cryptosystem can remain intact for many years without compromising the security of the system. SSL certificates are one such example.

One of the most interesting facts about asymmetric key cryptosystems is that the security of these systems is based on a small set of number-theory problems that are presumed difficult but were never mathematically proven to be difficult. Factoring, for instance, is one such

problem. Advances in number theory could one day render factoring a much easier problem hence diminishing security of many asymmetric key cryptosystems.

For CipherCloud, as we routinely protect customer data migrating to the cloud. We chose AES, a symmetric cipher, with a strong 256-bit implementation. With this choice, not only our implementation remains efficient, it also lends itself to a model where our customers retain exclusive control of the key.

7.Data encryption standard:

1.How DES Works in Detail:

DES is a *block cipher*--meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. Thus DES results in *apermutation* among the 2^{64} (read this as: "2 to the 64th power") possible arrangements of 64 bits, each of which may be either 0 or 1. Each block of 64 bits is divided into two blocks of 32 bits each, a left half block **L** and a right half **R**. (This division is only used in certain operations.)

Example: Let **M** be the plain text message **M** = 0123456789ABCDEF, where **M** is in hexadecimal (base 16) format. Rewriting **M** in binary format, we get the 64-bit block of text:

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

L = 0000 0001 0010 0011 0100 0101 0110 0111

R = 1000 1001 1010 1011 1100 1101 1110 1111

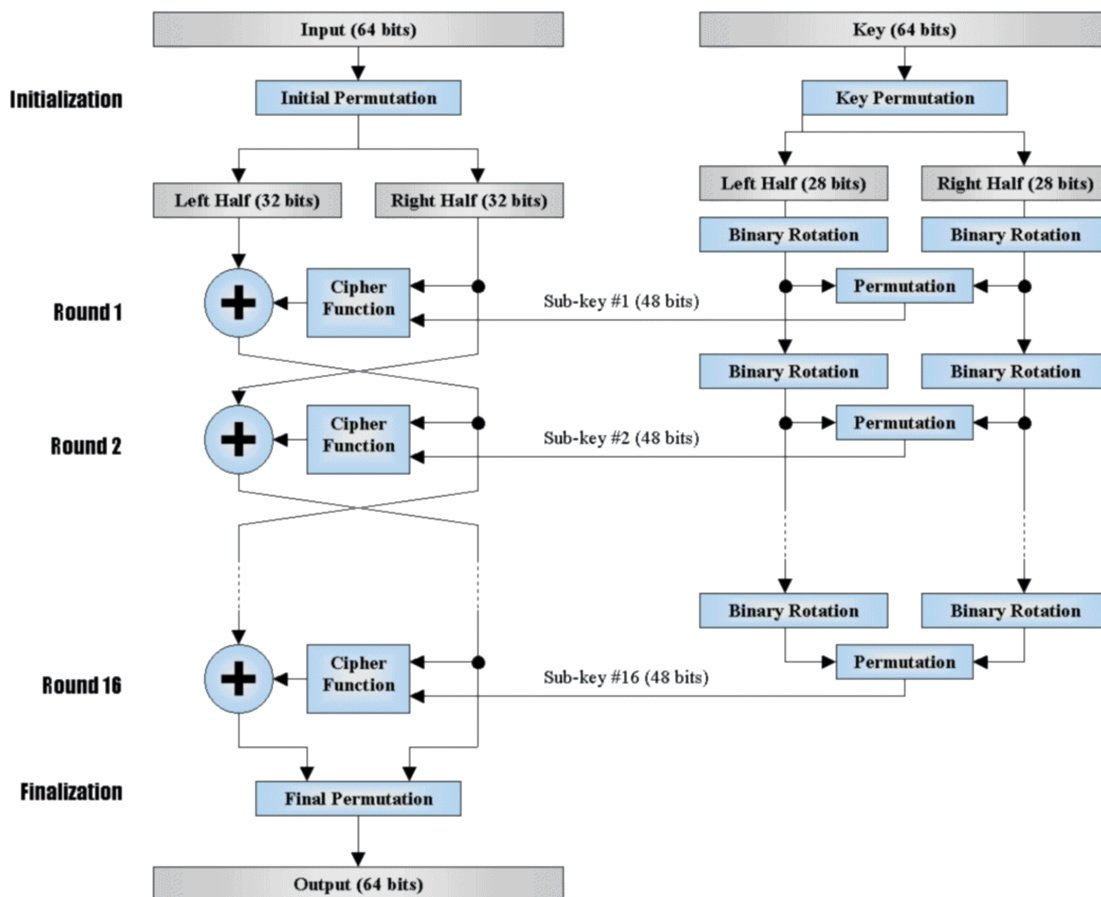
The first bit of **M** is "0". The last bit is "1". We read from left to right.

DES operates on the 64-bit blocks using *key* sizes of 56- bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used (i.e. bits numbered 8, 16, 24, 32, 40, 48, 56, and 64). However, we will nevertheless number the bits from 1 to 64, going left to right, in the following calculations. But, as you will see, the eight bits just mentioned get eliminated when we create subkeys.

Example: Let **K** be the hexadecimal key **K** = 133457799BBCDFF1. This gives us as the binary key (setting 1 = 0001, 3 = 0011, etc., and grouping together every eight bits, of which the last one in each group will be unused):

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

The DES algorithm uses the following steps:



Step 1: Create 16 subkeys, each of which is 48-bits long.

The 64-bit key is permuted according to the following table, **PC-1**. Since the first entry in the table is "57", this means that the 57th bit of the original key **K** becomes the first bit of the permuted key **K+**. The 49th bit of the original key becomes the second bit of the permuted key. The 4th bit of the original key is the last bit of the permuted key. Note only 56 bits of the original key appear in the permuted key.

PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Example: From the original 64-bit key

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

we get the 56-bit permutation

K+ = 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

Next, split this key into left and right halves, **C₀** and **D₀**, where each half has 28 bits.

Example: From the permuted key \mathbf{K}_+ , we get

$$C_0 = 11110000\ 0110011\ 0010101\ 0101111$$

$$D_0 = 0101010\ 1011001\ 1001111\ 0001111$$

With C_0 and D_0 defined, we now create sixteen blocks C_n and D_n , $1 \leq n \leq 16$. Each pair of blocks C_n and D_n is formed from the previous pair C_{n-1} and D_{n-1} , respectively, for $n = 1, 2, \dots, 16$, using the following schedule of "left shifts" of the previous block. To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

This means, for example, C_3 and D_3 are obtained from C_2 and D_2 , respectively, by two left shifts, and C_{16} and D_{16} are obtained from C_{15} and D_{15} , respectively, by one left shift. In all cases, by a single left shift is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3, ..., 28, 1.

Example: From original pair pair C_0 and D_0 we obtain:

$$C_0 = 1111000011001100101010101111$$

$$D_0 = 0101010101100110011110001111$$

$$C_1 = 1110000110011001010101011111$$

$$D_1 = 1010101011001100111100011110$$

$$C_2 = 1100001100110010101010111111$$

$$D_2 = 0101010110011001111000111101$$

$$C_3 = 0000110011001010101011111111$$

$$D_3 = 0101011001100111100011110101$$

$$C_4 = 0011001100101010101111111100$$

$$D_4 = 0101100110011110001111010101$$

$$C_5 = 1100110010101010111111110000$$

$$D_5 = 0110011001111000111101010101$$

$$C_6 = 001100101010101111111000011$$

$$D_6 = 1001100111100011110101010101$$

$$C_7 = 110010101010111111100001100$$

$$D_7 = 0110011110001111010101010110$$

$$C_8 = 001010101011111110000110011$$

$$D_8 = 1001111000111101010101011001$$

$$C_9 = 0101010101111111100001100110$$

$$D_9 = 0011110001111010101010110011$$

$$C_{10} = 0101010111111110000110011001$$

$$D_{10} = 1111000111101010101011001100$$

$$C_{11} = 0101011111111000011001100101$$

$$D_{11} = 1100011110101010101100110011$$

$$C_{12} = 0101111111100001100110010101$$

$$D_{12} = 0001111010101010110011001111$$

$$C_{13} = 0111111110000110011001010101$$

$$D_{13} = 0111101010101011001100111100$$

$$C_{14} = 1111111000011001100101010101$$

$$D_{14} = 1110101010101100110011110001$$

$$C_{15} = 1111100001100110010101010111$$

$$D_{15} = 1010101010110011001111000111$$

$$C_{16} = 1111000011001100101010101111$$

$$D_{16} = 0101010101100110011110001111$$

We now form the keys K_n , for $1 \leq n \leq 16$, by applying the following permutation table to each of the concatenated pairs $C_n D_n$. Each pair has 56 bits, but **PC-2** only uses 48 of these.

PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Therefore, the first bit of K_n is the 14th bit of $C_n D_n$, the second bit the 17th, and so on, ending with the 48th bit of K_n being the 32th bit of $C_n D_n$.

Example: For the first key we have $C_1 D_1 = 1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011110$

which, after we apply the permutation **PC-2**, becomes

$K_1 = 000110\ 110000\ 001011\ 101111\ 111111\ 000111\ 000001\ 110010$

For the other keys we have

$K_2 = 011110\ 011010\ 111011\ 011001\ 110110\ 111100\ 100111\ 100101$
 $K_3 = 010101\ 011111\ 110010\ 001010\ 010000\ 101100\ 111110\ 011001$
 $K_4 = 011100\ 101010\ 110111\ 010110\ 110110\ 110011\ 010100\ 011101$
 $K_5 = 011111\ 001110\ 110000\ 000111\ 111010\ 110101\ 001110\ 101000$
 $K_6 = 011000\ 111010\ 010100\ 111110\ 010100\ 000111\ 101100\ 101111$
 $K_7 = 111011\ 001000\ 010010\ 110111\ 111101\ 100001\ 100010\ 111100$
 $K_8 = 111101\ 111000\ 101000\ 111010\ 110000\ 010011\ 101111\ 111011$
 $K_9 = 111000\ 001101\ 101111\ 101011\ 111011\ 011110\ 011110\ 000001$
 $K_{10} = 101100\ 011111\ 001101\ 000111\ 101110\ 100100\ 011001\ 001111$
 $K_{11} = 001000\ 010101\ 111111\ 010011\ 110111\ 101101\ 001110\ 000110$
 $K_{12} = 011101\ 010111\ 000111\ 110101\ 100101\ 000110\ 011111\ 101001$
 $K_{13} = 100101\ 111100\ 010111\ 010001\ 111110\ 101011\ 101001\ 000001$
 $K_{14} = 010111\ 110100\ 001110\ 110111\ 111100\ 101110\ 011100\ 111010$
 $K_{15} = 101111\ 111001\ 000110\ 001101\ 001111\ 010011\ 111100\ 001010$
 $K_{16} = 110010\ 110011\ 110110\ 001011\ 000011\ 100001\ 011111\ 110101$

So much for the subkeys. Now we look at the message itself.

Step 2: Encode each 64-bit block of data.

There is an *initial permutation* **IP** of the 64 bits of the message data **M**. This rearranges the bits according to the following table, where the entries in the table show the new arrangement of the bits from their initial order. The 58th bit of **M** becomes the first bit of **IP**. The 50th bit of **M** becomes the second bit of **IP**. The 7th bit of **M** is the last bit of **IP**.

IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Example: Applying the initial permutation to the block of text **M**, given previously, we get

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
IP = 1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010

Here the 58th bit of **M** is "1", which becomes the first bit of **IP**. The 50th bit of **M** is "1", which becomes the second bit of **IP**. The 7th bit of **M** is "0", which becomes the last bit of **IP**.

Next divide the permuted block **IP** into a left half **L₀** of 32 bits, and a right half **R₀** of 32 bits.

Example: From **IP**, we get **L₀** and **R₀**

L₀ = 1100 1100 0000 0000 1100 1100 1111 1111
R₀ = 1111 0000 1010 1010 1111 0000 1010 1010

We now proceed through 16 iterations, for $1 \leq n \leq 16$, using a function f which operates on two blocks--a data block of 32 bits and a key K_n of 48 bits--to produce a block of 32 bits. **Let + denote XOR addition, (bit-by-bit addition modulo 2).** Then for n going from 1 to 16 we calculate

$$L_n = R_{n-1}$$

$$R_n = L_{n-1} + f(R_{n-1}, K_n)$$

This results in a final block, for $n = 16$, of $L_{16}R_{16}$. That is, in each iteration, we take the right 32 bits of the previous result and make them the left 32 bits of the current step. For the right 32 bits in the current step, we XOR the left 32 bits of the previous step with the calculation f .

Example: For $n = 1$, we have

$$K_1 = 000110\ 110000\ 001011\ 101111\ 111111\ 000111\ 000001\ 110010$$

$$L_1 = R_0 = 1111\ 0000\ 1010\ 1010\ 1111\ 0000\ 1010\ 1010$$

$$R_1 = L_0 + f(R_0, K_1)$$

It remains to explain how the function f works. To calculate f , we first expand each block R_{n-1} from 32 bits to 48 bits. This is done by using a selection table that repeats some of the bits in R_{n-1} . We'll call the use of this selection table the function E . Thus $E(R_{n-1})$ has a 32 bit input block, and a 48 bit output block.

Let E be such that the 48 bits of its output, written as 8 blocks of 6 bits each, are obtained by selecting the bits in its inputs in order according to the following table:

E BIT-SELECTION TABLE

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Thus the first three bits of $E(R_{n-1})$ are the bits in positions 32, 1 and 2 of R_{n-1} while the last 2 bits of $E(R_{n-1})$ are the bits in positions 32 and 1.

Example: We calculate $E(R_0)$ from R_0 as follows:

$$R_0 = 1111\ 0000\ 1010\ 1010\ 1111\ 0000\ 1010\ 1010$$

$$E(R_0) = 011110\ 100001\ 010101\ 010101\ 011110\ 100001\ 010101\ 010101$$

(Note that each block of 4 original bits has been expanded to a block of 6 output bits.)

Next in the f calculation, we XOR the output $E(R_{n-1})$ with the key K_n :

$$K_n + E(R_{n-1}).$$

Example: For K_1 , $E(R_0)$, we have

$$K_1 = 000110\ 110000\ 001011\ 101111\ 111111\ 000111\ 000001\ 110010$$

$$E(R_0) = 011110\ 100001\ 010101\ 010101\ 011110\ 100001\ 010101\ 010101$$

$$K_1 + E(R_0) = 011000\ 010001\ 011110\ 111010\ 100001\ 100110\ 010100\ 100111.$$

We have not yet finished calculating the function f . To this point we have expanded R_{n-1} from 32 bits to 48 bits, using the selection table, and XORed the result with the key K_n . We now have 48 bits, or eight groups of six bits. We now do something strange with each group of six bits: we use them as addresses in tables called "**S boxes**". Each group of six bits will give us an address in a different **S** box. Located at that address will be a 4 bit number. This 4 bit number will replace the original 6 bits. The net result is that the eight groups of 6 bits are transformed into eight groups of 4 bits (the 4-bit outputs from the **S** boxes) for 32 bits total.

Write the previous result, which is 48 bits, in the form:

$$K_n + E(R_{n-1}) = B_1B_2B_3B_4B_5B_6B_7B_8,$$

where each B_i is a group of six bits. We now calculate

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$$

where $S_i(B_i)$ refers to the output of the i -th **S** box.

To repeat, each of the functions S_1, S_2, \dots, S_8 , takes a 6-bit block as input and yields a 4-bit block as output. The table to determine S_1 is shown and explained below:

S1

		Column Number															
Row																	
No.		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1		0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2		4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3		15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

If S_1 is the function defined in this table and B is a block of 6 bits, then $S_1(B)$ is determined as follows: The first and last bits of B represent in base 2 a number in the decimal range 0 to 3 (or binary 00 to 11). Let that number be i . The middle 4 bits of B represent in base 2 a number in the decimal range 0 to 15 (binary 0000 to 1111). Let that number be j . Look up in the table the number in the i -th row and j -th column. It is a number in the range 0 to 15 and is uniquely represented by a 4 bit block. That block is the output $S_1(B)$ of S_1 for the input B . For example, for input block $B = 011011$ the first bit is "0" and the last bit "1" giving 01 as the row. This is row 1. The middle four bits are "1101". This is the binary equivalent of decimal 13, so the column is column number 13. In row 1, column 13 appears 5. This determines the output; 5 is binary 0101, so that the output is 0101. Hence $S_1(011011) = 0101$.

The tables defining the functions S_1, \dots, S_8 are the following:

S1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5

0 14 7 11 10 4 13 1 5 8 12 6 9 3 2 15
13 8 10 1 3 15 4 2 11 6 7 12 0 5 14 9

S3

10 0 9 14 6 3 15 5 1 13 12 7 11 4 2 8
13 7 0 9 3 4 6 10 2 8 5 14 12 11 15 1
13 6 4 9 8 15 3 0 11 1 2 12 5 10 14 7
1 10 13 0 6 9 8 7 4 15 14 3 11 5 2 12

S4

7 13 14 3 0 6 9 10 1 2 8 5 11 12 4 15
13 8 11 5 6 15 0 3 4 7 2 12 1 10 14 9
10 6 9 0 12 11 7 13 15 1 3 14 5 2 8 4
3 15 0 6 10 1 13 8 9 4 5 11 12 7 2 14

S5

2 12 4 1 7 10 11 6 8 5 3 15 13 0 14 9
14 11 2 12 4 7 13 1 5 0 15 10 3 9 8 6
4 2 1 11 10 13 7 8 15 9 12 5 6 3 0 14
11 8 12 7 1 14 2 13 6 15 0 9 10 4 5 3

S6

12 1 10 15 9 2 6 8 0 13 3 4 14 7 5 11
10 15 4 2 7 12 9 5 6 1 13 14 0 11 3 8
9 14 15 5 2 8 12 3 7 0 4 10 1 13 11 6
4 3 2 12 9 5 15 10 11 14 1 7 6 0 8 13

S7

4 11 2 14 15 0 8 13 3 12 9 7 5 10 6 1
13 0 11 7 4 9 1 10 14 3 5 12 2 15 8 6
1 4 11 13 12 3 7 14 10 15 6 8 0 5 9 2
6 11 13 8 1 4 10 7 9 5 0 15 14 2 3 12

S8

13 2 8 4 6 15 11 1 10 9 3 14 5 0 12 7
1 15 13 8 10 3 7 4 12 5 6 11 0 14 9 2
7 11 4 1 9 12 14 2 0 6 10 13 15 3 5 8
2 1 14 7 4 10 8 13 15 12 9 0 3 5 6 11

Example: For the first round, we obtain as the output of the eight **S** boxes:

$$K_I + \mathbf{E}(\mathbf{R}_0) = 011000\ 010001\ 011110\ 111010\ 100001\ 100110\ 010100\ 100111.$$

$$S_1(\mathbf{B}_1)S_2(\mathbf{B}_2)S_3(\mathbf{B}_3)S_4(\mathbf{B}_4)S_5(\mathbf{B}_5)S_6(\mathbf{B}_6)S_7(\mathbf{B}_7)S_8(\mathbf{B}_8) = 0101\ 1100\ 1000\ 0010\ 1011\ 0101\ 1001\ 0111$$

The final stage in the calculation of f is to do a permutation \mathbf{P} of the **S**-box output to obtain the final value of f :

$$f = \mathbf{P}(S_1(\mathbf{B}_1)S_2(\mathbf{B}_2)...S_8(\mathbf{B}_8))$$

The permutation **P** is defined in the following table. **P** yields a 32-bit output from a 32-bit input by permuting the bits of the input block.

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Example: From the output of the eight **S** boxes:

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8) = 0101\ 1100\ 1000\ 0010\ 1011\ 0101\ 1001\ 0111$$

we get

$$f = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

$$R_1 = L_0 + f(R_0, K_1)$$

$$\begin{aligned} &= 1100\ 1100\ 0000\ 0000\ 1100\ 1100\ 1111\ 1111 \\ &+ 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011 \\ &= 1110\ 1111\ 0100\ 1010\ 0110\ 0101\ 0100\ 0100 \end{aligned}$$

In the next round, we will have $L_2 = R_1$, which is the block we just calculated, and then we must calculate $R_2 = L_1 + f(R_1, K_2)$, and so on for 16 rounds. At the end of the sixteenth round we have the blocks L_{16} and R_{16} . We then *reverse* the order of the two blocks into the 64-bit block

$$R_{16}L_{16}$$

and apply a final permutation \mathbf{IP}^{-1} as defined by the following table:

\mathbf{IP}^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

Example: If we process all 16 blocks using the method defined previously, we get, on the 16th round,

$$\begin{aligned} L_{16} &= 0100\ 0011\ 0100\ 0010\ 0011\ 0010\ 0011\ 0100 \\ R_{16} &= 0000\ 1010\ 0100\ 1100\ 1101\ 1001\ 1001\ 0101 \end{aligned}$$

We reverse the order of these two blocks and apply the final permutation to

$$R_{16}L_{16} = 00001010\ 01001100\ 11011001\ 10010101\ 01000011\ 01000010\ 00110010\ 00110100$$
$$IP^{-1} = 10000101\ 11101000\ 00010011\ 01010100\ 00001111\ 00001010\ 10110100\ 00000101$$

which in hexadecimal format is

85E813540F0AB405.

This is the encrypted form of $\mathbf{M} = 0123456789\text{ABCDEF}$: namely, $\mathbf{C} = 85\text{E}813540\text{F}0\text{A}B405$.

Decryption is simply the inverse of encryption, following the same steps as above, but reversing the order in which the subkeys are applied.

DES Modes of Operation

The DES algorithm turns a 64-bit message block \mathbf{M} into a 64-bit cipher block \mathbf{C} . If each 64-bit block is encrypted individually, then the mode of encryption is called *Electronic Code Book* (ECB) mode. There are two other modes of DES encryption, namely *Chain Block Coding* (CBC) and *Cipher Feedback* (CFB), which make each cipher block dependent on all the previous messages blocks through an initial XOR operation.

Cracking DES

Before DES was adopted as a national standard, during the period NBS was soliciting comments on the proposed algorithm, the creators of public key cryptography, Martin Hellman and Whitfield Diffie, registered some objections to the use of DES as an encryption algorithm. Hellman wrote: "Whit Diffie and I have become concerned that the proposed data encryption standard, while probably secure against commercial assault, may be extremely vulnerable to attack by an intelligence organization" (letter to NBS, October 22, 1975).

Diffie and Hellman then outlined a "brute force" attack on DES. (By "brute force" is meant that you try as many of the 2^{56} possible keys as you have to before decrypting the ciphertext into a sensible plaintext message.) They proposed a special purpose "parallel computer using one million chips to try one million keys each" per second, and estimated the cost of such a machine at \$20 million.

Fast forward to 1998. Under the direction of John Gilmore of the EFF, a team spent \$220,000 and built a machine that can go through the entire 56-bit DES key space in an average of 4.5 days. On July 17, 1998, they announced they had cracked a 56-bit key in 56 hours. The computer, called Deep Crack, uses 27 boards each containing 64 chips, and is capable of testing 90 billion keys a second.

Despite this, as recently as June 8, 1998, Robert Litt, principal associate deputy attorney general at the Department of Justice, denied it was possible for the FBI to crack DES: "Let me put the technical problem in context: It took 14,000 Pentium computers working for four months to decrypt a single message We are not just talking FBI and NSA [needing massive computing power], we are talking about every police department."

Responded cryptography expert Bruce Schneier: ". . . the FBI is either incompetent or lying, or both." Schneier went on to say: "The only solution here is to pick an algorithm with a longer key; there isn't enough silicon in the galaxy or enough time before the sun burns out to brute-force triple-DES" (*Crypto-Gram*, Counterpane Systems, August 15, 1998).

Triple-DES

Triple-DES is just DES with two 56-bit keys applied. Given a plaintext message, the first key is used to DES-encrypt the message. The second key is used to DES-decrypt the encrypted message. (Since the second key is not the right key, this decryption just scrambles the data further.) The twice-scrambled message is then encrypted again with the first key to yield the final ciphertext. This three-step procedure is called triple-DES.

Triple-DES is just DES done three times with two keys used in a particular order. (Triple-DES can also be done with three separate keys instead of only two. In either case the resultant key space is about 2^{112} .)

8.RSA Algorithm:

The RSA algorithm is named after Ron Rivest, Adi Shamir and Len Adleman, who invented it in 1977 [RIVE78]. The basic technique was first discovered in 1973 by Clifford Cocks [COCK73] of CESG (part of the British GCHQ) but this was a secret until 1997. The patent taken out by RSA Labs has expired.

The RSA cryptosystem is the most widely-used public key cryptography algorithm in the world. It can be used to encrypt a message without the need to exchange a secret key separately.

The RSA algorithm can be used for both public key encryption and digital signatures. Its security is based on the difficulty of factoring large integers.

Party A can send an encrypted message to party B without any prior exchange of secret keys. A just uses B's public key to encrypt the message and B decrypts it using the private key, which only he knows. RSA can also be used to sign a message, so A can sign a message using their private key and B can verify it using A's public key.

Key generation algorithm:

This is the original algorithm.

1. Generate two large random primes, p and q , of approximately equal size such that their product $n = pq$ is of the required bit length, e.g. 1024 bits.
 2. Compute $n = pq$ and $(\phi) \phi = (p-1)(q-1)$.
 3. Choose an integer e , $1 < e < \phi$, such that $\gcd(e, \phi) = 1$.
 4. Compute the secret exponent d , $1 < d < \phi$, such that $ed \equiv 1 \pmod{\phi}$
 5. The public key is (n, e) and the private key (d, p, q) . Keep all the values d, p, q and ϕ secret. [We prefer sometimes to write the private key as (n, d) because you need the value of n when using d . Other times we might write the key pair as $((N, e), d)$.]
- n is known as the *modulus*.
 - e is known as the *public exponent* or *encryption exponent* or just the *exponent*.
 - d is known as the *secret exponent* or *decryption exponent*.

A practical key generation algorithm:

Incorporating the advice given in the [notes below](#), a practical algorithm to generate an RSA key pair is given below. Typical bit lengths are $k = 1024, 2048, 3072, 4096, \dots$, with increasing computational expense for larger values. You will not go far wrong if you choose e as 65537 ($=0x10001$) in step (1).

Algorithm: Generate an RSA key pair.

INPUT: Required modulus bit length, k .

OUTPUT: An RSA key pair $((N, e), d)$ where N is the modulus, the product of two primes ($N=pq$) not exceeding k bits in length; e is the public exponent, a number less than and coprime to $(p-1)(q-1)$; and d is the private exponent such that $ed \equiv 1 \pmod{(p-1)(q-1)}$.

1. Select a value of e from $\{3, 5, 17, 257, 65537\}$
2. **repeat**
3. $p \leftarrow \text{genprime}(k/2)$
4. **until** $(p \bmod e) \neq 1$
5. **repeat**
6. $q \leftarrow \text{genprime}(k - k/2)$
7. **until** $(q \bmod e) \neq 1$
8. $N \leftarrow pq$
9. $L \leftarrow (p-1)(q-1)$
10. $d \leftarrow \text{modinv}(e, L)$
11. **return** (N, e, d)

The function $\text{genprime}(b)$ returns a prime of exactly b bits, with the b th bit set to 1. Note that the operation $k/2$ is *integer* division giving the integer quotient with no fraction.

If you've chosen $e = 65537$ then the chances are that the first prime returned in steps (3) and (6) will pass the tests in steps (4) and (7), so each repeat-until loop will most likely just take one iteration. The final value of N may have a bit length slightly short of the target k . This actually does not matter too much (providing the message m is always $< N$), but some schemes require a modulus of exact length. If this is the case, then just repeat the entire algorithm until you get one. It should not take too many goes. Alternatively, use the trick setting the two highest bits in the prime candidates .

Encryption:

Sender A does the following:-

1. Obtains the recipient B's public key (n, e) .
2. Represents the plaintext message as a positive integer m , $1 < m < n$ [see [note 4](#)].
3. Computes the ciphertext $c = m^e \bmod n$.
4. Sends the ciphertext c to B.

Decryption:

Recipient B does the following:-

1. Uses his private key (n, d) to compute $m = c^d \bmod n$.
2. Extracts the plaintext from the message representative m .

9. Creating a Symmetric Encryption Stream

The `CryptoStream` class is used to pass encrypted blocks of data to an underlying stream. The underlying stream can be any `Stream` type - `FileStream`, `MemoryStream`, or `NetworkStream`. The `CryptoStream` constructor requires three parameters:

```
CryptoStream(Stream stream,  
ICryptoTransform transform,  
CryptoStreamMode mode)
```

The first parameter, `stream`, represents the underlying stream the encrypted data will be passed to (or read from). The third parameter, `mode`, defines whether the `CryptoStream` will read data from, or write data to, the underlying stream (unfortunately, you cannot use the same `CryptoStream` object to both read and write data on the same stream).

The second parameter, `transform`, is the tricky one. It controls the encryption algorithm used, and whether the stream will be used for encrypting or decrypting the data. For this parameter you must use either the `CreateEncryptor()` or `CreateDecryptor()` method from one of the symmetric encryption classes in Table 1 to define the specific algorithm used.

When the encryptor and decryptor are defined they must include the private key and IV values used to encrypt or decrypt the data. And of course, both the encryptor and decryptor must use the same key and IV values. Listing 1 shows a sample code snippet that demonstrates how to create an encryptor in an application.

The TCP/IP Problem

Reading the encrypted data from the `FileStream` object was easy - the decryptor program knew when all of the encrypted data was read when the end of the data file was reached. This is not the case when dealing with data on a `NetworkStream`. The problem with sending encrypted data streams with a `NetworkStream` object is that it is difficult to determine when the end of the encrypted data is received from the remote host.

TCP is a stream-oriented protocol. As separate messages are fed into the same stream, they all merge to become a single stream of data, with no delineation between individual messages. As the stream is read, multiple messages can be extracted from the stream in a single `Read()` method. This is a problem for the decryptor program. For the `CryptoStream` to properly decrypt a message it must know exactly where each encrypted message starts and stops.

There are three methods that are commonly used to differentiate messages in TCP streams. The first method is similar to the `FileStream` solution - terminating the stream at the end of a message. This means only one encrypted message can be sent per TCP connection. In applications where this method is impractical (such

as a chat program), you must resort to one of the other two methods - sending a message boundary marker between messages, or sending each message size to the remote device.

For encrypted messages, boundary markers are often impractical, as it is difficult to determine a character to use as the marker. The easiest solution is to determine the size of each encrypted message, and send it to the remote device before the actual message. When the remote device receives the message size, it knows exactly how many bytes to read from the `NetworkStream` and feed into the decryptor. The problem is often how to accomplish this task.

Using a `MemoryStream`

If the data is fed directly to a `NetworkStream`, it is impossible to determine its size. The solution is the `MemoryStream` object. By feeding the output of the `CryptoStream` to a `MemoryStream` instead of directly to the `NetworkStream`, you can determine the encrypted data length of the message, and then easily move the data into a byte array for sending out the `NetworkStream`.

The `MemoryStream` object is linked to the `CryptoStream`, placing the output of the encrypted data directly into the `MemoryStream`. The `MemoryStream` can then be converted to a byte array using the `GetBytes()` method, and the length of the encrypted data is determined using the `Length` property of the `MemoryStream` (which must be typecast to an `int`).

A Complete Encrypted Network Application

Listing 3

```
using System;
using System.IO;
using System.Net.Sockets;
using System.Security;
using System.Security.Cryptography;
using System.Text;

public class CryptoNet {
    public string CryptoRecv(NetworkStream strm) {
        MemoryStream memstrm = new MemoryStream();
        byte[] Key = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                     0x09, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};
        byte[] IV = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                    0x09, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};

        TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();
        CryptoStream csw = new CryptoStream(memstrm, tdes.CreateDecryptor(Key, IV),
            CryptoStreamMode.Write);

        byte[] data = new byte[1024];
        int recv = strm.Read(data, 0, 4);
        int size = BitConverter.ToInt32(data, 0);
        int offset = 0;
        while(size > 0)
        {
            recv = strm.Read(data, 0, size);
            csw.Write(data, offset, recv);
            offset += recv;
            size -= recv;
        }
    }
}
```

```

    }
    csw.FlushFinalBlock();
    memstrm.Position = 0;
    byte[] info = memstrm.GetBuffer();
    int infosize = (int)memstrm.Length;
    csw.Close();
    memstrm.Close();
    return Encoding.ASCII.GetString(info, 0, infosize);
}

public void CryptoSend(NetworkStream strm, string data)
{
    MemoryStream memstrm = new MemoryStream();

    byte[] Key = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
        0x09, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};

    byte[] IV = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
        0x09, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};

    TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();
    CryptoStream csw = new CryptoStream(memstrm, tdes.CreateEncryptor(Key, IV),
        CryptoStreamMode.Write);

    csw.Write(Encoding.ASCII.GetBytes(data), 0, data.Length);
    csw.FlushFinalBlock();

    byte[] cryptdata = memstrm.GetBuffer();
    int size = (int)memstrm.Length;
    byte[] bytesize = BitConverter.GetBytes(size);
    strm.Write(bytesize, 0, 4);
    strm.Write(cryptdata, 0, size);
    strm.Flush();
    csw.Close();
    memstrm.Close();
}
}

```

Listing 3 shows the CryptoNet class, which contains two methods, CryptoSend() and CryptoRecv(). The CryptoSend() method accepts a string object, encrypts it into a MemoryStream, then sends the size of the encrypted data and the data itself out on a specified NetworkStream. The CryptoRecv() method accepts a NetworkStream object, reads a data size received from the stream, and then reads that amount of data from the stream. The data is decrypted, placed in a MemoryStream, and then converted into a string object, which is returned to the calling program.

Using the CryptoNet class in network programs is a snap. Listing 4 shows the CryptoSrvr program, which acts as an echo server for encrypted messages. Listing 5 shows the CryptoClient program, which connects to the server, sends an encrypted message, and waits for the encrypted message to be returned. Both programs use the CryptoNet classes to encrypt and decrypt the data.

Listing 5: echo server

```
using System;
using System.Net.Sockets;
class CryptoClient
{
    public static void Main()
    {
        CryptoNet cn = new CryptoNet();
        TcpClient sock = new TcpClient("127.0.0.1", 9050);
        NetworkStream strm = sock.GetStream();

        while(true)
        {
            Console.Write("Enter text to send:");
            string data = Console.ReadLine();
            if (data.Equals("bye"))
                break;
            cn.CryptoSend(strm, data);
            string newdata = cn.CryptoRecv(strm);
            Console.WriteLine("got back: {0}", newdata);
        }
        strm.Close();
        sock.Close();
    }
}
```

Listing 4: CryptoClient

```
using System;
using System.Net;
using System.Net.Sockets;

class CryptoSrvr
{
    public static void Main()
    {
        CryptoNet cn = new CryptoNet();
        TcpListener sock = new TcpListener(IPAddress.Any, 9050);
        sock.Start();
        Console.WriteLine("Waiting for a client...");
        TcpClient client = sock.AcceptTcpClient();
        Console.WriteLine("accepted client, waiting for data...");
        NetworkStream strm = client.GetStream();

        while(true)
        {
            string data = cn.CryptoRecv(strm);
```

```

        if (data.Equals(""))
            break;
        Console.WriteLine("Received: {0}", data);
        cn.CryptoSend(strm, data);
    }
    strm.Close();
    client.Close();
    sock.Stop();
}
}

```

Because there is no guarantee of receiving the message at once, there is just a stream of bytes, it is necessary to use some message splitting mechanism. There are the following options:

- Prefix the message with its size (size itself has fixed length, for example: 1 byte, 2 bytes, 4 bytes, etc.), i.e. introduce the message header.
- Message terminator, use some byte or byte sequence to mark the end of the message.

That is why application-level protocol (see Application layer) has to be designed using one of those mechanisms to exchange with the messages.

10.SSLStream Class

SslStream is a stream used for client-server communication that uses the Secure Socket Layer (SSL) security protocol to authenticate the server and optionally the client.

SSL protocols help to provide confidentiality and integrity checking for messages transmitted using an SslStream. An SSL connection, such as that provided by SslStream, should be used when communicating sensitive information between a client and a server. Using an SslStream helps to prevent anyone from reading and tampering with information while it is in transit on the network.

An SslStream instance transmits data using a stream that you supply when creating the SslStream. When you supply this underlying stream, you have the option to specify whether closing the SslStream also closes the underlying stream. Typically, the SslStream class is used with the TcpClient and TcpListener classes. The GetStream() method provides a NetworkStream suitable for use with the SslStream class.

After creating an SslStream, the server and optionally, the client must be authenticated. The server must provide an X509 certificate that establishes proof of its identity and can request that the client also do so. Authentication must be performed before transmitting information using an SslStream. Clients initiate authentication using the synchronous AuthenticateAsClient methods, which block until the authentication completes, or the asynchronous BeginAuthenticateAsClient methods, which do not block waiting for the authentication to complete. Servers initiate authentication using the synchronous AuthenticateAsServer() or asynchronous BeginAuthenticateAsServer() methods. Both client and server must initiate the authentication.

The authentication is handled by the Security Support Provider (SSPI) channel provider. The client is given an opportunity to control validation of the server's certificate by specifying a RemoteCertificateValidationCallback delegate when creating an SslStream. The server can also control validation by supplying a CertificateValidationCallback delegate. The method referenced by the delegate includes the remote party's certificate and any errors SSPI encountered while validating the certificate. Note that if the server specifies a delegate, the delegate's method is invoked regardless of whether the server requested client authentication. If the server did not request client authentication, the server's delegate method receives a null certificate and an empty array of certificate errors.

If the server requires client authentication, the client must specify one or more certificates for authentication. If the client has more than one certificate, the client can provide a LocalCertificateSelectionCallback delegate to select the correct certificate for the server. The client's certificates must be located in the current user's "My" certificate store. Client authentication via certificates is not supported for the SSL2 (SSL version 2) protocol.

If the authentication fails, you receive an AuthenticationException, and the SslStream is no longer useable. You should close this object and remove all references to it so that it can be collected by the garbage collector. When the authentication process, also known as the SSL handshake, succeeds, the identity of the server (and optionally, the client) is established and the SslStream can be used by the client and server to exchange messages. Before sending or receiving information, the client and server should check the security services

and levels provided by the SslStream to determine whether the protocol, algorithms, and strengths selected meet their requirements for integrity and confidentiality. If the current settings are not sufficient, the stream should be closed. You can check the security services provided by the SslStream using the IsEncrypted and IsSigned properties. The following table shows the elements that report the cryptographic settings used for authentication, encryption and data signing.

Element	Members
The security protocol used to authenticate the server and, optionally, the client.	The SslProtocol property and the associated SslProtocols enumeration.
The key exchange algorithm.	The KeyExchangeAlgorithm property and the associated ExchangeAlgorithmType enumeration.
The message integrity algorithm.	The HashAlgorithm property and the associated HashAlgorithmType enumeration.
The message confidentiality algorithm.	The CipherAlgorithm property and the associated CipherAlgorithmType enumeration.
The strengths of the selected algorithms.	The KeyExchangeStrength, HashStrength, and CipherStrength properties.

After a successful authentication, you can send data using the synchronous Write() or asynchronous BeginWrite() methods. You can receive data using the synchronous Read() or asynchronous BeginRead() methods.

If you specified to the SslStream that the underlying stream should be left open, you are responsible for closing that stream when you are done using it.

If the application that creates the SslStream object runs with the credentials of a Normal user, the application will not be able to access certificates installed in the local machine store unless permission has been explicitly given to the user to do so.

The following code snippet example creates an SslStream and initiates the client portion of the authentication.

=====

As well as receiving and sending information (just like a client), a server has to keep track of who is connected to it. It is also useful to be able to broadcast messages, that is send them to every client currently connected (for example, a message indicating the server is about to be taken down for maintenance). You can try to check out the following article about client-server interactions using C#

- <http://www.codeproject.com/cs/internet/sockets.asp> for reference.

Here are some steps for you to follow:

1. Creating a client socket with ordinary .NET functions is slightly messy, so I provide a cover. And once you have the socket, you need to create an instance of ClientInfo to get event-driven access to the data.

```
// At the top of the file, you will always need
using System.Net.Sockets;
using RedCorona.Net;
class SimpleClient{
    ClientInfo client;
    void Start(){
        Socket sock = Sockets.CreateTCPSocket("www.myserver.com", 2345);
        client = new ClientInfo(sock, false); // Don't start receiving yet
        client.OnReadBytes += new ConnectionReadBytes(ReadData);
        client.BeginReceive();
    }

    void ReadData(ClientInfo ci, byte[] data, int len){
        Console.WriteLine("Received "+len+" bytes: "+
            System.Text.Encoding.UTF8.GetString(data, 0, len));
    }
}
```

2. Messaged Communication:

Often, being able to pass text with a suitable end marker, often a new line, is all we need in an application. Similar things can then be done with data received like this as with command strings. To do this, assign a handler to the OnRead event:

```
class TextClient{
```

```

ClientInfo client;
void Start(){
    Socket sock = Sockets.CreateTCPSocket("www.myserver.com", 2345);
    client = new ClientInfo(sock, false); // Don't start receiving yet
    client.OnRead += new ConnectionRead(ReadData);
    client.Delimiter = '\n'; // this is the default, shown for illustration
    client.BeginReceive();
}

void ReadData(ClientInfo ci, String text){
    Console.WriteLine("Received text message: "+text);
}
}

```

3. Here is a simple server class which simply 'bounces' messages back where they came from, unless they start with '!' in which case it broadcasts them:

```

class SimpleServer{
    Server server;
    ClientInfo client;
    void Start(){
        server = new Server(2345, new ClientConnect(ClientConnect));
    }

    bool ClientConnect(Server serv, ClientInfo new_client){
        new_client.Delimiter = '\n';
        new_client.OnRead += new ConnectionRead(ReadData);
        return true; // allow this connection
    }

    void ReadData(ClientInfo ci, String text){
        Console.WriteLine("Received from "+ci.ID+": "+text);
        if(text[0] == '!')
            server.Broadcast(Encoding.UTF8.GetBytes(text)); //Broadcast Message here
        else ci.Send(text);
    }
}

```

=====

I have a error when opening a window and closing another.

The calling thread must be STA, because many UI components require this.

I am using RedCorona Sockets... <http://www.redcorona.com/sockets.cs> Here is the code...

```

public partial class MainWindowThingy : Window
{
    public ClientInfo client;
    public MainWindowThingy() //The Error appears here
    {
        InitializeComponent();
        StartTCP();
    }
    public void StartTCP()
    {
        Socket sock = Sockets.CreateTCPSocket("localhost", 2345);
        client = new ClientInfo(sock, false); // Don't start receiving yet
        client.OnReadBytes += new ConnectionReadBytes(ReadData);
        client.BeginReceive();
    }
}

```



```

    }
    void ReadData(ClientInfo ci, byte[] data, int len)
    {
        string msg = (System.Text.Encoding.UTF8.GetString(data, 0, len));
        string[] amsg = msg.Split(' ');
        switch (amsg[0])
        {
            case "login":
                if (bool.Parse(amsg[1]) == true)
                {
                    MainWindowThingy SecondWindow = new MainWindowThingy();
                    Login FirstWindow = new Login();
                    SecondWindow.Show();
                    FirstWindow.Close(); //It starts here, the error...
                }
                else
                {
                }
                break;
            }
        }
    }
}

```

Basically, It gives me the error at the "Public Control()" When closing The First Form...

Uhm... I want to open another form and close the other... basicley

2.0 Streams and Protocols

A stream is a continuous flow of bytes. For a chat server and a chat client to communicate over the network, each would read and write to the network stream. The network stream is duplex, and bytes transmitted and read are always in FIFO order.

When a chat client is connected to a chat server, they would have established a common network stream to use.

However, for any meaningful communication, there must be some rules and order. These rules and order would be known as the communication protocols.

There are at least a few layers of protocols. At the lowest level, the communicating parties would need to know how to break a continuous flow of bytes into packets/frames. These rules would be referred to as the Network Protocol. At the next level, the parties would need to interpret the packets/frames. These rules would be known as the Application Protocol(s).

3.0 Network Protocol

I would like to make a distinction between a text stream and a binary stream. In a text stream, only text characters are allowed. In a binary stream, all byte values (0x00 - 0xFF) are allowed.

One way to set markers in a text stream is to use a non-text byte as a marker. For example, the traditional C string is terminated by 0x00, which serves as an end marker.

For a binary stream, there is no way to set a marker because all byte values are legal. Thus, one way to break a binary stream to packets is for the parties to communicate to one another about the size of the binary bytes to follow, before actually sending the bytes.

In ChatStream.cs (and ChatStream.vb), the ChatStream class is implemented with methods for reading and writing text data (Read() and Write()), and also methods for reading and writing binary data (ReadBinary() and WriteBinary()).

Note that in the Write() method, we set a 0x03 as the marker, and Read() will read until the marker is encountered. For the WriteBinary() method, no marker is set, and ReadBinary() requires an input parameter to indicate the number of bytes to read.

4.0 Sending and Receiving Pictures

If you ever use a binary editor to view a picture file (JPG or BMP), you will know that all byte values are possible in a picture file. To transfer picture binary data from a file or memory stream, we would not be able to set a marker to break the stream as what we can do for text data.

For this program:

The protocol for sending a picture is as follows:

- The client sends a command: send pic:<target>.
- When the server receives the command, it will check if <target> has an active connection. It then replies with "<server> send pic".
- When the client receives this special message, it will send a text message to indicate to the server the number of bytes of binary data that will be sent over. Following that, the binary data are then sent.
- The server uses the ChatStream ReadBinary method to get the binary data, and then saves the data to a file marked with the sender and target names.
- The server will then send a message to the <target> indicating that there is a picture ready for it to retrieve.

The protocol for getting a picture is as follows:

- The client sends a command: get pic:<sender>.
- When the server receives the command, it will first check if there is a file with the <sender> and the client name. If so, it will send the reply "<server> get pic". It will then send a text message to indicate to the client the number of bytes to read. Then the binary data will be sent over.
- The client uses the common ChatStream ReadBinary method to get the binary data and display the image in the RichTextBox.

Separate Security Protocol

The designers of the Secure Sockets Layer decided to create a separate protocol just for security. In effect, they added a layer to the Internet's protocol architecture. The left side of figure 1-4 shows the key protocols for Web communications.

At the bottom is the Internet Protocol (ip). This protocol is responsible for routing messages across networks from their source to their destination. The Transmission Control Protocol (tcp) builds on the services of ip to ensure that the communication is reliable. At the top is the Hypertext Transfer Protocol; http understands the details of the interaction between Web browsers and Web servers.

As the right side of the figure indicates, ssl adds security by acting as a separate security protocol, inserting itself between the http application and tcp. By acting as a new protocol, ssl requires very few changes in the protocols above and below. The http application interfaces with ssl nearly the same way it would with tcp in the absence of security. And, as far as tcp is concerned, ssl is just another application using its services.

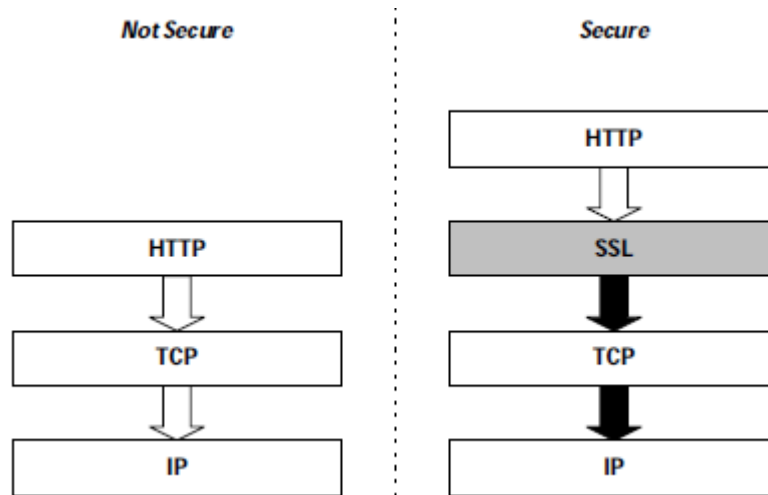


Figure 1-4 SSL is a separate protocol layer just for security.

In addition to requiring minimal changes to existing implementations, this approach has another significant benefit: It allows ssl to support applications other than http. The main motivation behind the development of ssl was Web security, but, as figure 1-5 shows, ssl is also used to add security to other Internet applications, including those of the Net News Transfer Protocol (nntp) and the File Transfer Protocol (ftp).

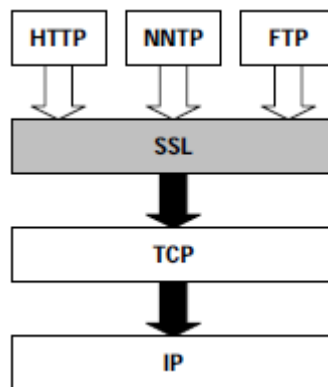


Figure 1-5 SSL can add security to applications other than HTTP.

Application-Specific Security

Although the designers of **ssl** chose a different strategy, it is also possible to add security services directly in an application protocol. Indeed, standard http does include some extremely rudimentary security features; however, those security features don't provide adequate protection for real electronic commerce. At about the same time Netscape was designing ssl, another group of protocol designers was working on an enhancement to http known as Secure http. Figure 1-6 shows the resulting protocol architecture. The Secure http standard has been published by the ietf as an experimen protocol, and a few products support it. It never caught on to the same degree as ssl, however, and oday it is rare to find Secure http anywhere on the Internet.

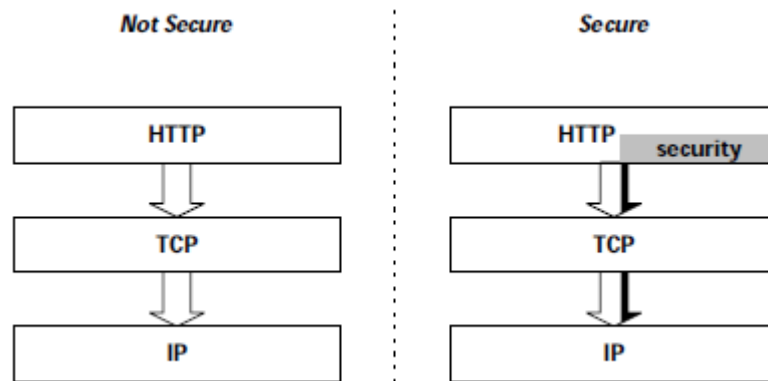


Figure 1-6 Security can be added directly within an application protocol.

protocol, and a few products support it. It never caught on to the same degree as ssl, however, and today it is rare to find Secure http anywhere on the Internet.

One of the disadvantages of adding security to a specific application is that the security services are available only to that particular application. Unlike ssl, for example, it is not possible to secure nntp, ftp, or other application protocols with Secure http. Another disadvantage of this approach is that it ties the security services tightly to the application. Every time the application protocol changes, the security implications must be carefully considered, and, frequently, the security functions of the protocol must be modified as well. A separate protocol like ssl isolates security services from the application protocol, allowing each to concentrate on solving its own problems most effectively.

Security within Core Protocols

The separate protocol approach of **ssl** can be taken one step further if security services are added directly to a core networking protocol. That is exactly the approach of the ip security (**ipsec**) architecture; full security services become an optional part of the Internet Protocol itself. Figure 1-7 illustrates the ipsec architecture. The ipsec architecture has many of the same advantages as ssl. It is independent of the application protocol, so any application may use it. In most cases, the application does not need to change at all to take advantage of ipsec. In fact, it may even be completely unaware that ipsec is involved at all. This feature does create its own challenges, however, as ipsec must be sufficiently flexible to support all applications. This complexity may be a big factor in the delays in development and deployment of ipsec.

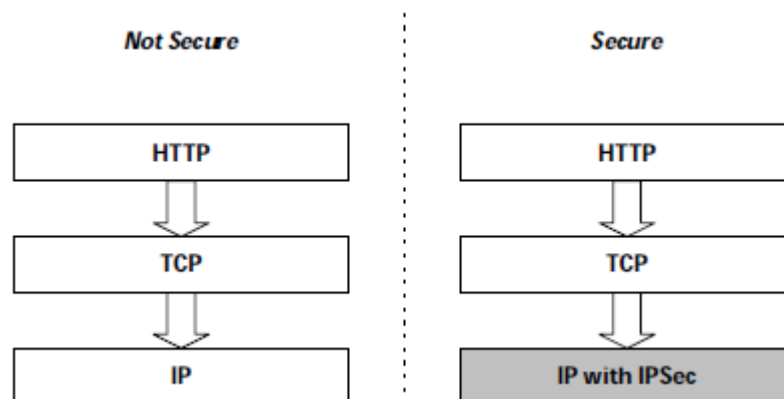


Figure 1-7 IPSEC adds security to a core network protocol.

Another concern with the ipsec approach is that it provides too much isolation between the application and security services. At least in its simplest implementations, ipsec tends to assume that secure requirements are a function of a particular system, and that all applications within that system need the same security services.

The ssl approach provides isolation between applications and security, but it allows some interaction between the two. The internal behavior of an application such as http need not change when security is added, but the application typically has to make the decision to use ssl or not. Such interaction makes it easier for each application to direct the security services most appropriate to its needs.

Despite these drawbacks, ipsec adds powerful new security tools to the Internet, and it will undoubtedly see widespread deployment. The ssl protocol, however, has significant benefits as well, and its deployment is also expected to grow substantially in the future.

Protocol Limitations

The ssl protocol, like any technology, has its limitations. And because ssl provides security services, it is especially important to understand its limits. After all, a false sense of security may be worse than no security. The limitations of ssl fall generally into three categories. First are fundamental constraints of the ssl protocol itself. These are a consequence of the design of ssl and its intended application. The ssl protocol also inherits some weaknesses from the tools it uses, namely encryption and signature algorithms. If these algorithms have weaknesses, ssl generally cannot rehabilitate them. Finally, the environments in which ssl is deployed have their own shortcomings and limitations, some of which ssl is helpless to address.

Fundamental Protocol Limitations

Though its design includes considerations for many different applications, ssl is definitely focused on securing Web transactions. Some of its characteristics reflect that concentration. For example, some of its characteristics reflect that concentration. For example, ssl requires a reliable transport protocol such as tcp. That is a completely reasonable requirement in the world of Web transactions, because the Hypertext Transfer Protocol itself requires tcp. The decision means, however, that ssl cannot operate using a connectionless transport protocol like udp.

Another role that ssl fails to fill is support for a particular security service known as non-repudiation. Non-repudiation associates the digital equivalent of a signature with data, and when used properly, it prevents the party that creates and “signs” data from successfully denying that after the fact. The ssl protocol does not provide nonrepudiation services, so ssl alone would not be appropriate for an application that required it.

Tool Limitations

The Secure Sockets Layer is simply a communication protocol, and any ssl implementation will rely on other components for many functions, including the cryptographic algorithms. These algorithms are the mathematical tools that actually perform tasks such as encryption and decryption. No ssl implementation can be any stronger than the cryptographic tools on which it is based.

Environmental Limitations

A network protocol alone can only provide security for information as it transits a network. No network protocol protects data before it is sent or after it arrives at its destination. This is

the only known weakness in Web security that has been successfully exploited in an actual commercial setting. Unfortunately, it has been exploited more than once.

Security in any computer network, whether the public Internet or private facilities, is a function of all the elements that make up that network. It depends on the network security protocols, the computer systems that use those protocols, and the human beings who use those computers. No network security protocol can protect against the confidential printout carelessly left on a cafeteria table.

The Secure Sockets Layer protocol is a strong and effective security tool, but it is only a single tool. True security requires many such tools, and a comprehensive plan to employ them.

11.Project:

I wanted to prototype encrypted communication channel between a client and a server. Now of course there are HTTPS and other TLS channels that work quite well, but what I have in mind is supposed to be used to transfer rather **sensitive data**. So how can I establish a secure channel through an HTTP/HTTPS channel?

In this project, we will build a Very Simple Secured Protocol (VSSP) that sits above the TCP/IP layer.

People sometimes are mistaken to think that a secure channel means a channel where data is encrypted. However, encryption is only part of the equation. A good example can be a treasure map that Alice sent to her best friend Bob the pirate. This map describes the steps needed in order to reach the treasure.

So, in summary, a secure channel needs to have at least three properties:

Encryption

Message validation

Message authentication

Building a basic secure channel

Alice and Bob share a private key (it does not matter how they did it for now), and they want to create a secure channel in order to send a message.

Alice:

$e = \text{encrypt}(k, t)$

$d = \text{digest}(k, e)$

$m = e + d$

send to Bob (m)

Bob:

$(e, d) = m$

if ($\text{digest}(k, e) == d$)

$t = \text{decrypt}(k, e)$

else

 error

Does this basic secure channel achieve the three properties that we have defined?

Encryption – Yes. Bob and Alice share a private key. They use a symmetric encryption algorithm.

Message validation – Yes. Bob and Alice use a digest algorithm (HMAC).

Message authentication – Yes. Because Bob and Alice share a private key, no one else can use this key to forge a message. The underlying assumption here is that both Alice and Bob's machines are secure and no one else has access to this key.

There is one basic flaw in the above algorithm, the assumption that Alice and Bob manage to share a private key. It is not always possible, so is there a way to remove this assumption? Asymmetric encryption comes to the rescue.

Alice:

```
initiate connection to Bob over TCP/IP
// as_k stands for asymmetric public key
as_k = gets from Bob
k = create a random key
encrypted_key = as_encrypt (as_k, k)
send to Bob (encrypted_key)
```

```
e = encrypt (k, t)
d = digest (k, e)
m = e + d
send to Bob (m)
```

Bob:

```
initiate connection with Alice over TCP/IP
send to Alice (as_k)
encrypted_key = gets from Alice
// as_pk stand for asymmetric private key.
k = as_decrypt(encrypted_key, as_pk)
```

```
(e, d) = m
if (digest (k, e) == d)
    t = decrypt (k, e)
else
    error
```

By using asymmetric encryption, Bob can now send his public key to Alice. She, in turn, will use it to encrypt the shared secret (marked as k in the pseudo code above), which will be used as the key for the symmetric encryption and digest algorithms.

Does this basic secure channel achieve the properties that we defined?

Encryption – Yes. Bob and Alice share a private secret. They use a symmetric encryption algorithm.

Message validation – Yes. Bob and Alice use a digest algorithm (HMAC).

Message authentication – No. What will happen if Eve intercepts the public key and instead sends her own? Thus, Alice can not be sure who actually sent her the public key. Without knowing it, Alice can accidentally share the private secret with Eve. In consequence, when the private secret is compromised, the entire channel is broken.

Certificate and Certificate Authorities (CA) can help us to authenticate the public key. As a substitute for the public key, Bob sends a certificate. The certificate is signed by a trusted party (usually the CA). It binds the public key to a DNS name. Now, Alice can authenticate the public key.

TCP/IP

The above algorithm is using TCP/IP as the underlying transport layer. It has a significant influence on how the secured channel manages its connection. TCP/IP is responsible for ensuring that data packets are sent to the endpoint and assembled in the correct order when they arrive. This assumption removes a lot of overhead from our proposed secured channel.

The following paragraphs will demonstrate a real secured channel called VSSP (Very Simple Secured Protocol), which is a simplified version of SSL/TLS.

Handshake - Where the client and the server agree on cryptographic algorithms that will be used, and authenticate each other.

Data transfer - Where the real data is transferred, i.e., files, text, etc.

Closure - Ending the connection in a secured manner.

Handshake

In the above algorithms, the assumption is that `encrypt`, `as_encrypt`, `decrypt`, and `as_decrypt` are known to the client and server but it is not always true. Both client and server need to agree on a set of algorithms that will be used during the entire connection. Try to imagine that not all clients and servers are running the same version of our protocol. Newer versions will include more powerful instances of cryptographic algorithms, stronger keys, or totally different algorithms. We have to ensure that both the client and the server talk in the same language. The handshake phase assures that.

The authentication occurs during the handshake. There are two types of authentication: server authentication and client authentication. Server authentication happens when the client asks from the server to authenticate itself, and vice versa for client authentication. Authentication in VSSP uses Public Key Infrastructure (PKI), i.e., certificates and only supports server authentication.

Handshake protocol phases

Creating a secure channel

The Solution

Well, I soon looked for the solution without the certificates. I knew that SSL uses an asymmetric key to connect, and then creates a symmetric key to continue its communication. Looking at the `System.Security.Cryptography` I saw the `CryptoStream`. I thought it was the solution (at least for the symmetric part), but it wasn't. The `CryptoStream` is unidirectional (so it is not OK for TCP/IP), its `Flush` doesn't flush the stream and, if I use `FlushFinalBlock`, I must dispose the stream. So, I decided to look at the Symmetric and Asymmetric algorithms directly.

After some study, I created a solution. Maybe not the fastest one, but it works.

1. During initialization, the server creates an asymmetric key and sends the public part to the client.
2. The client then creates an symmetric key (at the moment only he knows the key) and encrypts it using the server public key (so, only the server knows how to decrypt it). It sends the key to the server, and then, only this symmetric key is used.

Simple, but as the asymmetric and the symmetric key are created during connection, there is no chance of someone else also knowing the keys. And, as the symmetric key is sent using the cryptography that only the server knows how to decrypt, even someone sniffing the network with a good cryptography knowledge will not have anything to do.

So, let's see the implementation.

1. `using System;`
2. `using System.IO;`
3. `using System.Net.Sockets;`
4. `using System.Collections;`
5. `using System.Collections.Generic;`
6. `using System.Security.Cryptography;`
7. `using System.Text;`


```

8. namespace Pfz.Remoting
9. {

10. public class SecureStream : Stream
11. {
12. private static readonly byte[] fEmptyReadBuffer = new byte[0];

13. private MemoryStream fWriteBuffer;

14. public SecureStream(Stream baseStream, bool runAsServer)

15. {
16. this.BaseStream = baseStream;
17. runAsServer = true;
18. }

19. public SecureStream(Stream baseStream) :
20. this(baseStream, new RSACryptoServiceProvider(), "Rijndael", false) {
21. }

22. public SecureStream(Stream baseStream, bool runAsServer) :
23. this(baseStream, new RSACryptoServiceProvider(), "Rijndael", runAsServer)
24. {
25. }

26. public SecureStream(Stream baseStream, string symmetricAlgorithmName) :
27. this(baseStream, new RSACryptoServiceProvider(), symmetricAlgorithmName, false)
28. {
29. }

30. public SecureStream(Stream baseStream, string symmetricAlgorithmName, bool runAsServer) :
31. this(baseStream, new RSACryptoServiceProvider(), symmetricAlgorithmName, runAsServer)
32. {
33. }

34. public SecureStream(Stream baseStream, RSACryptoServiceProvider rsa, string
    symmetricAlgorithmName, bool runAsServer)
35. {
36. if (baseStream == null)
37. throw new ArgumentNullException("baseStream");

38. if (rsa == null)
39. throw new ArgumentNullException("rsa");

40. if (string.IsNullOrEmpty(symmetricAlgorithmName))
41. throw new ArgumentNullException("symmetricAlgorithm");

42. BaseStream = baseStream;
43. if (runAsServer)
44. {
45. SymmetricAlgorithm = SymmetricAlgorithm.Create(symmetricAlgorithmName);
46. string symmetricTypeName = SymmetricAlgorithm.GetType().ToString();
47. byte[] symmetricTypeBytes = Encoding.UTF8.GetBytes(symmetricTypeName);

48. byte[] sizeBytes = BitConverter.GetBytes(symmetricTypeBytes.Length);

```

```

49. baseStream.Write(sizeBytes, 0, sizeBytes.Length);
50. baseStream.Write(symmetricTypeBytes, 0, symmetricTypeBytes.Length);

51. byte[] bytes = rsa.ExportCspBlob(false); //public key
52. sizeBytes = BitConverter.GetBytes(bytes.Length);
53. baseStream.Write(sizeBytes, 0, sizeBytes.Length);
54. baseStream.Write(bytes, 0, bytes.Length);

55. SymmetricAlgorithm.Key = p_ReadWithLength(rsa); ;
56. SymmetricAlgorithm.IV = p_ReadWithLength(rsa);
57. }
58. else
59. {

60. var sizeBytes = new byte[4];
61. p_ReadDirect(sizeBytes);
62. var stringLength = BitConverter.ToInt32(sizeBytes, 0);

63. var stringBytes = new byte[stringLength];
64. p_ReadDirect(stringBytes);
65. var symmetricTypeName = Encoding.UTF8.GetString(stringBytes);
66. SymmetricAlgorithm = SymmetricAlgorithm.Create(symmetricTypeName);

67. sizeBytes = new byte[4];
68. p_ReadDirect(sizeBytes);
69. int asymmetricKeyLength = BitConverter.ToInt32(sizeBytes, 0);
70. byte[] bytes = new byte[asymmetricKeyLength];
71. p_ReadDirect(bytes);
72. rsa.ImportCspBlob(bytes);

73. p_WriteWithLength(rsa, SymmetricAlgorithm.Key);
74. p_WriteWithLength(rsa, SymmetricAlgorithm.IV);
75. }

76. rsa.Clear();

77. Decryptor = SymmetricAlgorithm.CreateDecryptor();
78. Encryptor = SymmetricAlgorithm.CreateEncryptor();

79. fReadBuffer = fEmptyReadBuffer;
80. fWriteBuffer = new MemoryStream(32 * 1024);
81. }

82. protected override void Dispose(bool disposing)
83. {
84. if (disposing)
85. {
86. var writeBuffer = fWriteBuffer;
87. if (writeBuffer != null)
88. {
89. fWriteBuffer = null;
90. writeBuffer.Dispose();
91. }

92. var encryptor = this.Encryptor;

```

```

93. if (encryptor != null)
94. {
95. Encryptor = null;
96. encryptor.Dispose();
97. }

98. var decryptor = this.Decryptor;
99. if (decryptor != null)
100. {
101. Decryptor = null;
102. decryptor.Dispose();
103. }

104. var symmetricAlgorithm = SymmetricAlgorithm;
105. if (symmetricAlgorithm != null)
106. {
107. SymmetricAlgorithm = null;
108. symmetricAlgorithm.Clear();
109. }

110. var baseStream = this.BaseStream;
111. if (baseStream != null)
112. {
113. BaseStream = null;
114. baseStream.Dispose();
115. }

116. fReadBuffer = null;
117. }

118. base.Dispose(disposing);
119. }

120. public Stream BaseStream { get; private set; }

121. public SymmetricAlgorithm SymmetricAlgorithm { get; private set; }

122. public ICryptoTransform Decryptor { get; private set; }

123. public ICryptoTransform Encryptor { get; private set; }

124. public override bool CanRead
125. {
126. get
127. {
128. return true;
129. }
130. }

131. public override bool CanSeek
132. {
133. get
134. {
135. return false;
136. }

```

```
137.     }

138.     public override bool CanWrite
139.     {
140.         get
141.         {
142.             return true;
143.         }
144.     }

145.     public override long Length
146.     {
147.         get
148.         {
149.             throw new NotSupportedException();
150.         }
151.     }

152.     public override long Position
153.     {
154.         get
155.         {
156.             throw new NotSupportedException();
157.         }
158.         set
159.         {
160.             throw new NotSupportedException();
161.         }
162.     }

163.     private readonly byte[] fSizeBytes = new byte[4];
164.     private int fReadPosition;
165.     private byte[] fReadBuffer;

166.     public override int Read(byte[] buffer, int offset, int count)
167.     {
168.         if (fReadPosition == fReadBuffer.Length)
169.         {
170.             p_ReadDirect(fSizeBytes);
171.             int readLength = BitConverter.ToInt32(fSizeBytes, 0);

172.             if (fReadBuffer.Length < readLength)
173.                 fReadBuffer = new byte[readLength];

174.             p_FullReadDirect(fReadBuffer, readLength);
175.             fReadBuffer = Decryptor.TransformFinalBlock(fReadBuffer, 0, readLength);

176.             fReadPosition = 0;
177.         }

178.         int diff = fReadBuffer.Length - fReadPosition;
179.         if (count > diff)
180.             count = diff;

181.         Buffer.BlockCopy(fReadBuffer, fReadPosition, buffer, offset, count);
```

```

182.         fReadPosition += count;
183.         return count;
184.     }

185.     public override long Seek(long offset, SeekOrigin origin)
186.     {
187.         throw new NotSupportedException();
188.     }

189.     public override void SetLength(long value)
190.     {
191.         throw new NotSupportedException();
192.     }

193.     public override void Write(byte[] buffer, int offset, int count)
194.     {
195.         fWriteBuffer.Write(buffer, offset, count);
196.     }

197.     public override void Flush()
198.     {
199.         if (fWriteBuffer.Length > 0)
200.         {
201.             var encryptedBuffer = Encryptor.TransformFinalBlock(fWriteBuffer.GetBuffer(),
202. 0, (int)fWriteBuffer.Length);
203.             var size = BitConverter.GetBytes(encryptedBuffer.Length);
204.             BaseStream.Write(size, 0, size.Length);
205.             BaseStream.Write(encryptedBuffer, 0, encryptedBuffer.Length);
206.             BaseStream.Flush();

207.             fWriteBuffer.SetLength(0);
208.             fWriteBuffer.Capacity = 32 * 1024;
209.         }
210.     }

210.     private void p_ReadDirect(byte[] bytes)
211.     {
212.         p_FullReadDirect(bytes, bytes.Length);
213.     }
214.     private void p_FullReadDirect(byte[] bytes, int length)
215.     {
216.         int read = 0;
217.         while (read < length)
218.         {
219.             int readResult = BaseStream.Read(bytes, read, length - read);

220.             if (readResult == 0)
221.                 throw new IOException("The stream was closed by the remote side.");

222.             read += readResult;
223.         }
224.     }
225.     private byte[] p_ReadWithLength(RSACryptoServiceProvider rsa)
226.     {
227.         byte[] size = new byte[4];
228.         p_ReadDirect(size);

229.         int count = BitConverter.ToInt32(size, 0);
230.         var bytes = new byte[count];
231.         p_ReadDirect(bytes);

232.         return rsa.Decrypt(bytes, false);

```

```

233.     }
234.     private void p_WriteWithLength(RSACryptoServiceProvider rsa, byte[] bytes)
235.     {
236.         bytes = rsa.Encrypt(bytes, false);
237.         byte[] sizeBytes = BitConverter.GetBytes(bytes.Length);
238.         BaseStream.Write(sizeBytes, 0, sizeBytes.Length);
239.         BaseStream.Write(bytes, 0, bytes.Length);
240.     }
241. }
242. }
243. //end of the class

```

Protocol:

The Server

1. It sends the **name of the symmetric algorithm** being used, which will be used by the client to check for compatibility.
2. It generates an RSA key pair and send the public key to the client.
3. Finally, reads the symmetric key and initialization vector that will be sent by the client.

The Client

The client does the reverse process of the server.

1. It first receives the algorithm named used by the server. If the length of the algorithm name or the name itself don't match, it throws an exception.
2. It receives the RSA key used by the server and
3. Sends the Key and Initialization Vector of its symmetric algorithm.

In 19, it define default constructor which use rijndeal algorithm in encryption and there are other constructor which we decide if it client or server and the final constructor decide which algorithm to use in encryption.

The encryption with the RSA key is done by the **p_ReadWithLength** and **p_WriteWithLength**, that in 225,234. Only to finish the constructor, it clears the RSA key, creates the symmetric encryptor and decryptor and initializes the buffers.

The Write encrypts the message, and then sends the size of the encrypted message and the message itself.

The Read reads the size, then reads the message and to finish decrypts and returns the decrypted message. But, wait, why I use "BaseStream.Write" and p_FullReadDirect? Why not BaseStream.Read?

I am really thinking about making this an extension method. If you look at how Read and Write works, you will notice the difference. Write simply writes all the requested buffers or throws an exception. Read is more problematic, as you can ask for 1024 bytes, and it returns 3, because it read only 3 bytes. But I don't expect this to happen, I want the full block, even if I need to call read many times.(line 166)

So, let's first understand the Encryptor and Decryptor. At least, the part I understood:

The Encryptor and the Decryptor are ICryptoTransform. In it, we have the TransformBlock and TransformFinalBlock. I really considered using TransformBlock, but in my tests, I encrypt a block and try to decrypt it, and nothing happens. If I join the blocks and at the end call TransformFinalBlock, I get the wrong result, so I decided to use only TransformFinalBlock, that alone works fine. (line 193)

The problem is that at each "final encryption", I can end-up with an extra size in the message. So, instead of encrypting each write, I buffer all of them in a memory stream and, during Flush, I encrypt all of the writes and send them.(line 197)

Why?

Because the remote side sends 1MB of data. To decrypt, I must read the 1MB of data and decrypt. But, the calling code only wants to read 4 bytes. I can't simply discard the rest of the buffer, I must read the part of the requested buffer and update the internal position, so the next read can read another part of the buffer. Also, we can have a buffer of 16 bytes and a read of 1024 bytes but, in this case, we use the Read behavior of returning that one 16 was read.

Why?

Because the remote side sends 1MB of data. To decrypt, I must read the 1MB of data and decrypt. But, the calling code only wants to read 4 bytes. I can't simply discard the rest of the buffer, I must read the part of the requested buffer and update the internal position, so the next read can read another part of the buffer. Also, we can have a buffer of 16 bytes and a read of 1024 bytes but, in this case, we use the Read behavior of returning that one 16 was read.

Chapter 4: Application

Contents of Chapter:

4.1 Overview.....	69
4.2 Getting Start.....	69
4.3 Using SQLite GUI client.....	71
4.4 Interaction with your database.....	72

Chapter 4 The application

1.Overview

Adding a database to your application can be an easy way to store data and settings between sessions for your program, but it is not always feasible to use a server based DBMS to store your database. SQLite is a small, fast, and reliable database which can be used without the end user having to install anything extra (achieved by referencing a single .dll in your project). There are a few things we as developers must do to get started with SQLite:

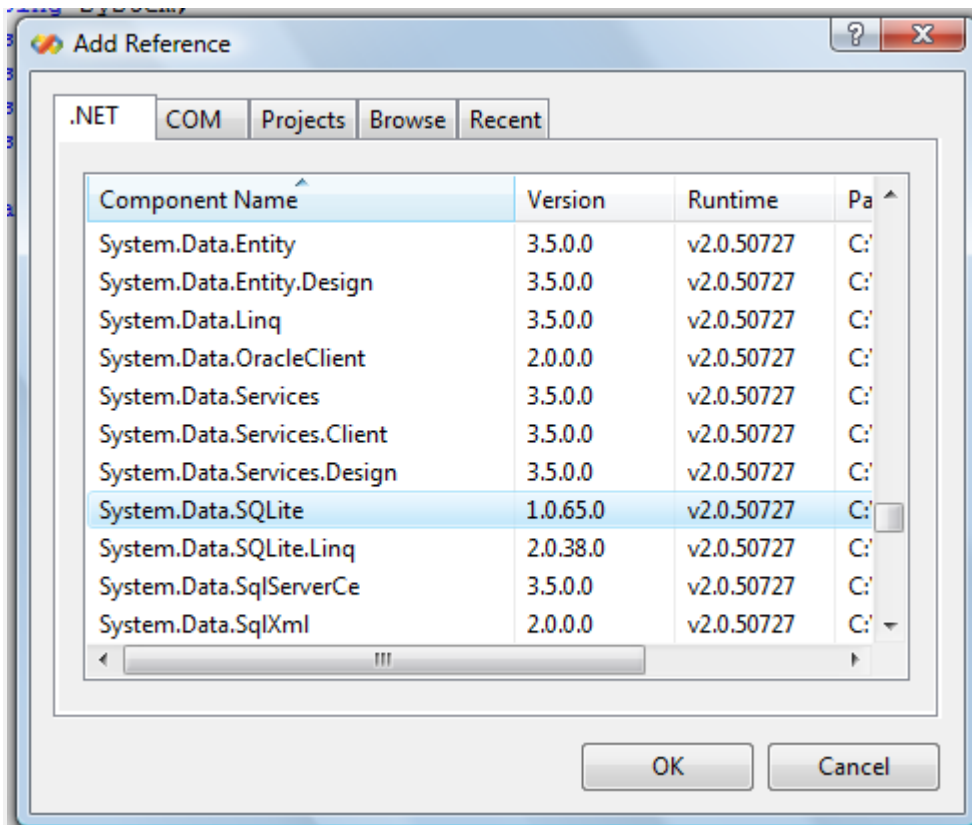
- Install the .NET provider for SQLite ~~from Sourceforge.net~~ from <http://system.data.sqlite.org>
- Add a reference to System.Data.SQLite to your project (and mark the .dll to be copied locally to your project)
- Optionally Download a [SQLite GUI Client](#) and use it to design your DB (Feel free to code it by hand if that is your preference)

If the above section made sense to you, feel free to jump down to the section titled “Interacting with your Database”, otherwise keep reading!

2.Getting Started

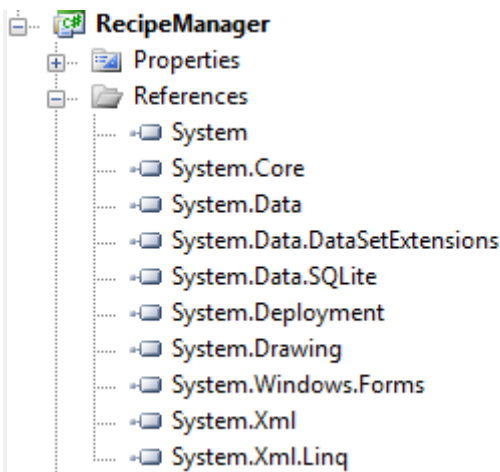
Referencing System.Data.SQLite

After you have installed the .NET provider for SQLite, you need to make sure that your project can access the required .dll. In Visual Studio 2008, this can be done by selecting “Project -> Add Reference...” from the main menu bar. A window will pop up, and under the “.NET” tab, scroll down and find System.Data.SQLite.



Adding a Reference in Visual Studio

Select it and click ok. It is now referenced in your project. The last thing we need to do is make sure Visual Studio copies the .dll for System.Data.SQLite to the project folder, which is necessary for SQLite to work without the provider.



Viewing References in Visual Studio

If the Solution Explorer window is not currently visible, open it by selecting “View -> Solution Explorer” from the main menu bar. Under the current project, click the + sign next to References to see a list of all currently referenced libraries.

Right click the reference to System.Data.SQLite, and select “Properties”. Set the property “Copy Local” to true.

You have now successfully referenced SQLite, and it can be added to any file by “using System.Data.SQLite;”.

3. Using the SQLite GUI Client

SQLite Administrator is a very straightforward Client, and I am not going to go into much detail with its use. I will however note a few things that were not immediately evident to me when I first used it.

- SQLite does not currently support foreign key constraints. Therefore SQLite Administrator does not have any way of linking tables via Foreign Key. That is certainly something to keep in mind.
- The box on the left hand side is for viewing the current Database and all of it's objects. If you see something you don't want to see, or don't see something you want to see, the buttons at the top of the box are toggle switches for tables, views, triggers, indexes, and so on. Since there are no tooltips, you'll just have to play around to figure out which is which function.

4. Interacting with your Database

Once the database is set up, it is time to begin reading from it and writing to it. In order to facilitate the interaction with the DB, I have written a helper class. It should be noted that a portion of this code is adapted from sample code in [this tutorial](#) by Mike Duncan. The Methods GetDataTable(), ExecuteNonQuery(), and ExecuteScalar() are his code and not mine.

Using the SQLiteDatabase Helper Class

```
1. using System;
2. using System.Collections.Generic;
3. using System.Text;
4. using Finisar.SQLite;
5. using System.Data ;
6. using System.Data.Common ;

7. namespace DAL
8. {
9.     public class SQLiteHelper
10.    {
11.        private static string connString;
12.        public static void Connect(string Dbname)
13.        {
14.            connString = String.Format("Data Source={0};New=False;Version=3", Dbname);
15.            try
16.            {
17.                using (SQLiteConnection conn = new SQLiteConnection(connString))
18.                {
19.                    conn.Open ();
20.                }
21.            }
22.            catch {}
23.        }

24.        public static SQLiteConnection GetSQLiteConnection ()
25.        {
26.            return new SQLiteConnection(connString);
27.        }
28.    }
29. }
```

```

27. }

28. private static void PrepareCommand(SQLiteCommand cmd, SQLiteConnection conn, string
    cmdText, params object[] p)
29. {
30. if (conn.State != ConnectionState.Open)
31. conn.Open();
32. cmd.Parameters.Clear();
33. cmd.Connection = conn;
34. cmd.CommandText = cmdText;
35. cmd.CommandType = CommandType.Text;
36. cmd.CommandTimeout = 30;
37. if (p != null)
38. {
39. for (int i = 0; i < p.Length; i++)
40. cmd.Parameters[i].Value = p[i];
41. }
42. }

43. public static int ExecuteNonQuery(string cmdText, params object[] p)
44. {
45. SQLiteCommand command = new SQLiteCommand();
46. using (SQLiteConnection connection = GetSQLiteConnection())
47. {
48. PrepareCommand(command, connection, cmdText, p);
49. return command.ExecuteNonQuery();
50. }
51. }

52. public static DataSet ExecuteDataset(string cmdText, ref SQLiteDataAdapter
    adapter, ref SQLiteConnection connection, params object[] p)
53. {
54. DataSet ds = new DataSet();
55. SQLiteCommand command = new SQLiteCommand();
56. SQLiteDataAdapter da;
57. connection = GetSQLiteConnection();
58. PrepareCommand(command, connection, cmdText, p);
59. da = new SQLiteDataAdapter(command);
60. SQLiteCommandBuilder oBuilder = new SQLiteCommandBuilder(da);
61. da.Fill(ds);
62. adapter = da;
63. return ds;
64. }

65. public static SQLiteDataReader ExecuteReader(string cmdText, params object[] p)
66. {
67. SQLiteCommand command = new SQLiteCommand();
68. SQLiteConnection connection = GetSQLiteConnection();
69. try
70. {
71. PrepareCommand(command, connection, cmdText, p);
72. SQLiteDataReader reader = command.ExecuteReader(CommandBehavior.CloseConnection);
73. return reader;
74. }
75. catch
76. {
77. connection.Close();
78. throw;
79. }
80. }

```

```

81. public static object ExecuteScalar(string cmdText, params object[] p)
82. {
83.     SQLiteCommand cmd = new SQLiteCommand();
84.     using (SQLiteConnection connection = GetSQLiteConnection())
85.     {
86.         PrepareCommand(cmd, connection, cmdText, p);
87.         return cmd.ExecuteScalar();
88.     }
89. }

90. public static DataSet ExecutePager(ref int recordCount, int pageIndex, int pageSize,
    string cmdText, string countText, params object[] p)
91. {
92.     if (recordCount < 0)
93.         recordCount = int.Parse(ExecuteScalar(countText, p).ToString());
94.     DataSet ds = new DataSet();
95.     SQLiteCommand command = new SQLiteCommand();
96.     using (SQLiteConnection connection = GetSQLiteConnection())
97.     {
98.         PrepareCommand(command, connection, cmdText, p);
99.         SQLiteDataAdapter da = new SQLiteDataAdapter(command);
100.         da.Fill(ds, (pageIndex - 1) * pageSize, pageSize, "result");
101.     }
102.     return ds;
103. }

104. }

105. }

106. //end of the class

```

figure 4.3 SQLiteHelper class

as shown in figure 4.3 notice that we include Finisar.SQLite (line 4) which is Data Provider for accessing SQLite-Databases using the .NET-Framework. at (line 5) we include the System.Data namespaces which contain classes for accessing and managing data from diverse sources. we also include The System.Data.Common namespace which contains classes shared by the .NET Framework data providers at (line 6). we may say enough about included namespaces now let's go deeper into the SQLiteHelper class (line 9)

*Connection

first of all we have to connect to the database that is why we have a connect method (line 12) which takes one string parameter (Dbname) so at line 14 Connect method makes a connection to the DB by passing the DB name to the string connString we then open the connection by passing the connstring to the SQLiteConnection class using the new object conn (line17) then we use conn.open() method to keep the connection open. (line 19) GetSQLiteConnection() method(line 24) just make the connection .we make the connection by passing connstring to SQLiteConnection class and return a new object of a connection (line 26).

*PrepareCommand

PrepareCommand method (line 28) responsible for commands executing .

We make sure of opening the connection first using open method from ConnectionState class (line 30). Then We empty the command parameters using Cmd.parameters.clear() which Cmd is an object of SQLiteCommand Class (line32).

The function parameter string cmdText got the user's command and equals it by Cmd.CommandText parameter which Cmd is an object of SQLiteCommand Class (line 34) .

Then we set the command type to Text (line35) notice that we use the CommandTimeout property and set it to

30 seconds which means if the command has not executed at 30 seconds the command would be ignored .
the PrepareCommand function takes array of objects (Parameters) so when we pass parameters to the function we set those parameters to the SQLiteCommand Parameters to be executed (from line 37 to 41)

ExecuteNonQuery used for executing queries that does not return any data.(line49) so if we have a sql statements like update, insert, delete etc. we do it by using ExecuteNonQuery function by passing the command to it then the function uses ExecuteNonQuery Property of SQLiteCommand class .

but what if we have a query like select or search or any resulted query ?
it just can be performed by using the ExecuteDataset function (line 52) which takes the command as a parameter (string cmdText) then making a new DataSet (ds) (line 54) and a new SQLiteCommand (command)(line55).

Notice that we make an object of a DataAdapter SQLiteDataAdapter (da) (line 56)

so what are we going to do with those objects ?

well , first we pass command and cmdText to PrepareCommand function (line 58,28) then DataAdapter will acts as a Bridge between DataSet and database.

This dataadapter object is used to read the data from database and bind that data to dataset.(line 59 and 61)
we also create an object of SQLiteCommandBuilder (line 60) it opens the Connection associated with the DataAdapter and makes a round trip to the server each and every time it's asked to construct the action queries. It closes the Connection when it's done after all the function returns the dataset ofcourse (line 63)

going down in the class we found a DataReader or we can say ExecuteReader function (line 65)
it will work with Action and Non-Action Queries (Select) Returns the collection of rows selected by the Query. Note the enumeration CommandBehavior (line 72) is used so , When the command is executed, the associated Connection object is closed when the associated DataReader object is closed.
that is why we catch and throw connection.close() (line 77) .Return type is DataReader. (line 73)

ExecuteScalar() (line 81):

will work with Non-Action Queries that contain aggregate functions. Return the first row and first column value of the query result. Return type is object. (line 87)

ExecutePager ()(line 90)

retrieving only those records from the database that must be displayed for the particular page of data requested by the user.

first if we don't have any records (line 92) then we call ExecuteScalar method and pass countText and the parameters to it convert the return object to string

using parse method we can set that string to the integer recordCount (line 93) so we can count the coming records .

we make a new objects of DataSet and SQLiteCommand (line 94,95) passing that command to SQLiteDataAdapter by creating the new object da (line 99)

so we can use the method fill of the DataAdapter (line 100) to fill the data set and return a the requested page by using the page index and page size which should be passed to the ExecutePager function as parameters by the user or the client

thus we set the start record parameter using the formula (pageIndex-1)*pagesize) which mean retrieving the particular page of data requested by the user from that point and we set the maxrecord parameter to the page size so it know wehe to stop getting data.

eventually we return the requested page as a data set type (line 102)

5.A Simple Client-Server Application using securestream:

```
1. using System;
2. using System.IO;
3. using System.Net;
4. using System.Net.Sockets;
5. using System.Net.NetworkInformation;
6. using System.Threading;
7. using System.Runtime.Serialization;
8. using System.Runtime.Serialization.Formatters.Binary;
9. using System.Collections.Generic;
10. using System.Data;
11. using Finisar.SQLite;

12. namespace Sqlite_server
13. {
14.     class ObjectServer
15.     {
16.         public static string name = "";
17.         public static DataSet DataSet=new DataSet ();
18.         public static SQLiteConnection conn = new SQLiteConnection();

19.         private static void ProcessClientRequests(Object argument)
20.         {
21.             TcpClient client = (TcpClient)argument;
22.             try
23.             {
24.                 StreamReader reader = new StreamReader(client.GetStream());
25.                 StreamWriter writer = new StreamWriter(client.GetStream());
26.                 String s = String.Empty;

27.                 while (!(s = reader.ReadLine()).Equals("Exit"))
28.                 {
29.                     string[] dataArray;
30.                     dataArray = s.Split(':');
31.                     Console.WriteLine(dataArray[0].ToString());
32.                     switch (dataArray[0])
33.                     {
34.                         case "Open_Ds":
35.                         {
36.                             Console.WriteLine(dataArray[0].ToString());
37.                             Console.WriteLine("From client -> " + s);
38.                             name = "D" + ":\\" + dataArray[1].ToString()+".db";
39.                             Console.WriteLine("D" + ":\\" + dataArray[1].ToString());
40.                             Serialize(getDataset("", name), client.GetStream());
41.                             client.GetStream().Flush();
42.                             break;
43.                         }
44.                         case "COL":
45.                         {
46.                             //Console.WriteLine(dataArray[0].ToString());
47.                             //Console.WriteLine("From client -> " + s);
48.                             break;
49.                         }
50.                         case "Crt":
51.                         {
52.                             Console.WriteLine(dataArray[0].ToString());
53.                             Console.WriteLine("From client -> " + s);

54.                             createTable(dataArray[1]);
55.                             client.GetStream().Flush();
56.                             break;
57.                         }
                    }
                }
            }
            catch { }
        }
    }
}
```

```

58. case "Del":
59. {
60. Console.WriteLine(dataArray[0].ToString());
61. Console.WriteLine("From client -> " + s);

62. deleteTable(dataArray[1]);
63. client.GetStream().Flush();
64. break;
65. }

66. case "Sql":
67. {
68. Console.WriteLine(dataArray[0].ToString());
69. Console.WriteLine("From client -> " + s);
70. //Get Dataset
71. DataSet = getDataset(dataArray[1], name);

72. DataColumn[] keyColumns = new DataColumn[1];
73. keyColumns[0] = DataSet.Tables[0].Columns[0];
74. DataSet.Tables[0].PrimaryKey = keyColumns;
75. //Send Dataset
76. Serialize(getDataset(dataArray[1], name), client.GetStream());
77. client.GetStream().Flush();
78. break;
79. }
80. case "Update":
81. {
82. Console.WriteLine(dataArray[0].ToString());
83. Console.WriteLine("From client -> " + s);
84. DataSet = Deserialize(client.GetStream());
85. if (conn.State != ConnectionState.Open)
86. conn.Open();
87. adapter.Update(DataSet.Tables[0]);
88. break;
89. }

90. case "Delete_table":
91. {
92. Console.WriteLine(dataArray[0].ToString());
93. Console.WriteLine("From client -> " + s);
94. DAL.SQLiteHelper.ExecuteNonQuery(dataArray[1], null);
95. break;
96. }

97. case "DB":
98. {
99. string fileName = "";
100. fileName = Path.ChangeExtension(@"D:\" + dataArray[1].ToString(), ".db");
101. FileInfo fi = new FileInfo(fileName);
102. FileStream fs = fi.Create();
103. fs.Flush(); fs.Close();

104. Console.WriteLine("From client -> " + s);

105. break;

106. }
107. case "c":
108. {
109. Console.WriteLine("From client -> connection close");

```

```

110.     reader.Close();
111.     writer.Close();
112.     client.Close();

113.     break;

114. }

115.     default:
116.     {
117.         Console.WriteLine("From client -> " + s);
118.         writer.WriteLine("From server -> " + s);
119.         writer.Flush();
120.         break;
121.     }
122. } // end switch
123. } // end while
124. reader.Close();
125. writer.Close();
126. client.Close();
127. Console.WriteLine("Client connection closed!");
128. }
129. catch (IOException)
130. {
131.     Console.WriteLine("Problem with client communication. Exiting thread.");
132. }
133. catch (NullReferenceException)
134. {
135.     Console.WriteLine("Incoming string was null! Client may have terminated
prematurly.");
136. }
137. catch (Exception e)
138. {
139.     Console.WriteLine("Unknown exception occured.");
140.     Console.WriteLine(e);
141. }
142. finally
143. {
144.     if (client != null)
145.     {
146.         client.Close();
147.     }
148. }
149. } // end ProcessClientRequests()

150. static SQLiteDataAdapter adapter=null;
151. private static DataSet getDataset(string sql, string name)
152. {
153.     if (sql == "")
154.     {
155.         DAL.SQLiteHelper.Connect(name);
156.         sql = "SELECT name FROM sqlite_master " +
157.             "WHERE type = 'table' " +
158.             "ORDER BY 1";
159.     }
160.     DataSet ds = DAL.SQLiteHelper.ExecuteDataset(sql, ref adapter, ref conn, null);
161.     return ds;
162. }

163. private static void createTable(string sql)
164. {
165.     DAL.SQLiteHelper.Connect(name);

166.     DAL.SQLiteHelper.ExecuteNonQuery(sql, null);

```



```

167.     }

168.     private static void deleteTable(string sql)
169.     {
170.         DAL.SQLiteHelper.Connect(name);
171.         DAL.SQLiteHelper.ExecuteNonQuery(sql, null);
172.     }

173.     public static void Serialize(DataSet ds, Stream stream)
174.     {
175.         BinaryFormatter serializer = new BinaryFormatter();
176.         serializer.Serialize(stream, ds);
177.     }

178.     public static DataSet Deserialize(Stream stream)
179.     {
180.         BinaryFormatter serializer = new BinaryFormatter();
181.         return (DataSet)serializer.Deserialize(stream);
182.     }

183.     private static void ShowServerNetworkConfig()
184.     {
185.         Console.ForegroundColor = ConsoleColor.Yellow;
186.         NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
187.         foreach (NetworkInterface adapter in adapters)
188.         {
189.             Console.WriteLine(adapter.Description);
190.             Console.WriteLine("\tAdapter Name: " + adapter.Name);
191.             Console.WriteLine("\tMAC Address: " + adapter.GetPhysicalAddress());
192.             IPInterfaceProperties ip_properties = adapter.GetIPProperties();
193.             UnicastIPAddressInformationCollection addresses =
ip_properties.UnicastAddresses;
194.             foreach (UnicastIPAddressInformation address in addresses)
195.             {
196.                 Console.WriteLine("\tIP Address: " + address.Address);
197.             }
198.         }
199.         Console.ForegroundColor = ConsoleColor.White;
200.     } // end ShowServerNetworkConfig()

201.     static void Main(string[] args)
202.     {
203.         TcpListener listener = null;
204.         try
205.         {
206.             ShowServerNetworkConfig();
207.             listener = new TcpListener(IPAddress.Any, 8080);
208.             listener.Start();
209.             Console.WriteLine("Sqlite Databae Server started...");
210.             while (true)
211.             {
212.                 Console.WriteLine("Waiting for incoming client connections...");
213.                 TcpClient client = listener.AcceptTcpClient();
214.                 Console.WriteLine("Accepted new client connection...");
215.                 Thread t = new Thread(ProcessClientRequests);
216.                 t.Start(client);
217.             }
218.         }
219.         catch (Exception e)
220.         {
221.             Console.WriteLine(e);
222.         }
223.         finally

```

```

224.     {
225.         if (listener != null)
226.         {
227.             listener.Stop();
228.         }
229.     }
230. }
231. }

232. }
233. //end of the class

```

Figure 4.4 Source code of server

As shown in Figure 4.4 what we got here is a namespace of sqlserver (line 12) but first we have to illustrate some namespaces we include here

well , line 7 The System.Runtime.Serialization namespace contains classes that can be used for serializing and deserializing objects.

to be clear Serialization is the process of converting an object or a graph of objects into a linear sequence of bytes for either storage or transmission to another location.

Deserialization is the process of taking in stored information and recreating objects from it.

and the namespace at line 8 The System.Runtime.Serialization.Formatters.Binary namespace contains the BinaryFormatter class, which can be used to serialize and deserialize objects in binary format.

we are also using System.Data and Finisar.SQLite namespaces as we mention before at SQLiteHelper class .

Good , now we can go deeper into the class

at line 20 ProcessClientRequests method takes one argument it performs client's commands so we create an object of TcpClient Class (line 22) and we can communicate with that client using StreamReader and StreamWriter for reading commands from and write result to the client (line 26,27)

Notice that we create an object of SecureStream class (line 19) and we use it at line 25 to pass the symmetric algorithm name (RC2) and use the method GetStream() of TcpClient class to get the basestream and pass it to the SecureStream Class at line 28 we create string s by which we receive the commands sent by the client and we make it empty using empty method .

now we will use stream reader to read from the client (line 29) and as long as we didn't read the word "Exit" we keep the connection open which created at line 18 .

the message from the client which contains the command should be at the string .

but we just need the command to perform it not all the message so we create an array of string (line 31) and use the split method to cut the message into parts(line 33)

the first part is dataArray[0] and the second part is dataArray[1] and so on .

Message parts are divided by ":" for example the client will send the message like "Sql:Select * from Table" so dataArray[0]="Sql" and dataArray[1]="Select * from Table" dataArray[0] always is the command . so at line 36 we make a switch statement for the commands that would be requested from the client

"Open_ds" command (line 38) which opens specific data set, after showing the command and the message on the console (line 40,41)

at line 42 we use the string name to set the data set name using dataArray[1] .

at line 44 we call the function Serialize which takes two parameters the first one is the data set name and we get that using the function getDataset() line (156) which returns the data set by it's name that is why we pass the string name to it .

the second parameter Serialize function takes is the stream by which send and reciev data so we pass the client stream using the method GetStream() of Client Class .

then we flushes the data from the stream using Flush() method (line 45)

in case of "Crt" command(line 54) by which we create a table so we use createTable function (line 168) and send the data from the message to it which are in the dataArray[1] (line 58)
"Del" command (line 62) use deleteTable function (line 173) to delete a table and send the data from the message to it which are in the dataArray[1] (line 66)

Notice that we flush the stream after execute any command . in case of "Sql" command (line 70) as it is a sql command then we got that command at dataArray[1] .

So, we get the data set from the GetDataset function as usual so we pass the sql command and the database name to it (line 75) .

So , we got the requested data set. NOW we need to add a primary key so we make a new object of DataColumn Class and set the first columns of the first table Tables[0] of the data set to that object then we add the primary key of the first table to that column (line 77 to 79)

finally we send the data set using Serialize function (line 81) the by same way we discussed before at line 44 .

in case of "Update" command (line 85) first we need to take stored information and recreating objects from it and that we call deserialization so we use the deserialize function (line 89) to set what in the stream to the data set . at line 90 we check of the connection state and if it is not open we open it at line 91
we create a SQLiteDataAdapter object at line 155 by which acts as a Bridge between Data Set and database.
So, we use its Update method to making updates to the data set tables (line 92)

in case of "Delete_Table" command (line 95) here we just use a little help from the SQLiteHelper to execute that command which as usual being in dataArray[1] by ExecuteNonQuery method as delete command doesn't return any result . (line 99)

in case of "DB " command (line 102) which create a data base here we treat with the data base as a file so we get the path of the data base and set the extension to ".db" here we got the fileName (line 105)
then we add that "fileName" to the FileInfo class which contains the information of the file using the object (line 106)
at line 107 we use the create method of FileInfo class to create the data base file and set it to the FileStream .
then we flush the File stream and close it using Flush() and Close() methods (line 108) then write the message to the console (line 109)
in case of "c" command (line 112)which closes the connection so use the close method of reader, writer , and client to close the connection (line 115 to 117)

that is all the cases we got here but if there is no commands we keep writing the message to the client and the server (line 122 , 123)

after all we close the connection (line 129 to 131) write to the console (line 132). "Client connection closed!"

at line 156 getDataset Function extended from Data set class it takes two parameters the sql command and the database name both of them are string .
first if there is not a sql command send by the client then it is connect to the data base using connect method of SQLiteHelper class.(line 160)
and after that it set the sql command to get the table names so that is the default if we call the getDataset function with empty sql command (line 161 to 163)

whatever ,at line 165 the data set we got here we got it using a little help from SQLiteHelper class especially ExecuteDataset method by passing the sql command to it and the adapter object which created at line 155 then it returns the requested data set (line 166).

the functions createTable and deleteTable (line 168 and line 173) are using ExecuteNonQuery method of the SQLiteHelper class (line 171 and 176) after connecting to it by connect method using data base name (line 170 and 175)

the functions Serialize (line 178) and Deserialize (line 183) using the Serialize and Deserialize methods of the BinaryFormatter class(line 180 , 185) by pass the stream and the data set to serialize method and just stream to Deserilaize method and it will return the data set .(line 181 , 186)
 at line 188 ShowServerNetworkConfig function as it's name it shows the configuration . at line 190 we set console foreground color to yellow .
 at line 191 we make an Array if type NetworkInterface class and set all network interfaces to it by the method GetAllNetworkInterfaces . then show all the objects in the array description , Name , and all address on the console (line 194 to 200) .at line 204 we set the foure ground color to white .

at the main function (line 206) we set tcplistener to null(line 208) then we show the server network configuration using ShowServerNetworkConfig() functon (line 211)

then the server start listening to the client activity at port 8080 to all kind of network interfaces . (line 212 , 213).at line 218 the server accepts the client's connection using the AcceptTcpClient method of TcpListener class and set it to the TcpClient object .
 line 220 initializes a new instance of the System.Threading.Thread class , specifying a delegate that allows an object to be passed to the thread when the thread is started at line 121 .
 we catch and show the exception (from line 224 to 227),then finally the server stop listening to the client (line 232).

```

1. using System;
1. using System.IO;
2. using System.Net;
3. using System.Net.Sockets;
4. using System.Runtime.Serialization;
5. using System.Runtime.Serialization.Formatters.Binary;
6. using System.Collections.Generic;
7. using System.Data;
8. using System.Windows.Forms;
9. using System.Threading;
10. using asdNet;
11. namespace client_1010
12. {
13.     public partial class frmMain : Form
14.     {
15.         StreamReader reader;
16.         StreamWriter writer;
17.         NetworkStream stream;
18.         SecureStream Sstream;
19.         public string query=null;
20.         public static string ss = "";
21.         string database_name = "";
22.         private int rowIndex = 0;
23.         int iii ;
24.         public frmMain()
25.         {
26.             InitializeComponent();
27.         }
28.         private void openToolStripMenuItem_Click(object sender, EventArgs e)

```

```

29.     {
30.         database_name = "";
31.         openDB();
32.     }
33.     private void openDB(){
34.         //Send
35.         string value = "";
36.         if (database_name=="")
37.         {
38.             if (Tmp.InputBox("Open Database", "Database name:", ref value) ==
DialogResult.OK)
39.             {
40.                 database_name = value;
41.                 writer.WriteLine("Open_Ds:" + database_name);
42. writer.Flush();
43.                 //recive
44.                 DataSet ds = Deserialize(stream);
45.
46.                 WriteToStatusBar(ds.Tables[0].Rows.Count.ToString());
47.                 tablecombobox.Items.Clear();
48.                 for (int i = 0; i < ds.Tables[0].Rows.Count; i++)
49.                     tablecombobox.Items.Add(ds.Tables[0].Rows[i][0].ToString());
50.                 if (ds.Tables[0].Rows.Count != 0)
51.                 {
52.                     btnDelete.Enabled = true;
53.                     tablecombobox.SelectedIndex = 0;
54.                 }
55.             }
56.             else
57.                 MessageBox.Show("No database was selected");
58.         }
59.     }
60.     private void ConnectTo(string ServerName, int Port)
61.     {
62.         TcpClient client;
63.         try
64.         {
65.             WriteToStatusBar(string.Format("Attempting to connect to server at IP
address: {0} port: {1}", ServerName, Port));
66.             client = new TcpClient(ServerName, Port);
67.             WriteToStatusBar("Connection successful!");
68.         }
69.         catch (Exception e)
70.         {
71.             throw new Exception("Failed to create client Socket: " + e.Message);
72.         }
73.         var baseStream = client.GetStream();
74.         stream = baseStream;
75.         Sstream = new SecureStream(baseStream);
76.         reader = new StreamReader(Sstream);
77.         writer = new StreamWriter(Sstream);
78.         String s = String.Empty;
79.     }

```

```

80. private void WriteToStatusBar(string Message)
81.     {
82.         StatusBar.Text = Message;
83.     }
84. public static void Serialize(DataSet ds, NetworkStream Stream)
85.     {
86.         BinaryFormatter serializer = new BinaryFormatter();
87.         serializer.Serialize(Stream, ds);
88.     }
89. public static DataSet Deserialize(NetworkStream stream)
90.     {
91.         BinaryFormatter serializer = new BinaryFormatter();
92. return (DataSet)serializer.Deserialize(stream);
93.     }
94. private void tlstrpbtnconnect_Click(object sender, EventArgs e)
95.     {
96.         ConnectTo("127.0.0.1", 8080);
97.     }
98. private void tlStpbtnnew_Click(object sender, EventArgs e)
99.     {
100.         button2.Visible = true;
101.         new_database.Visible = true;
102.         label6.Visible = true;
103.         button3.Visible = true;
104.     }
105. private void btnAdd_Click(object sender, EventArgs e)
106.     {
107.         if (query != null)
108.             query += ",";
109.         query += txtcolumnname.Text + " " + typecombo.Text;
110.         if (chkprimary.Checked)
111.             query += " primary key";
112.         txtcolumnname.Text = null; typecombo.Text = null;
113.     }
114. private void btnCreate_Click(object sender, EventArgs e)
115.     {
116.         if (query == null)
117.             MessageBox.Show("enter name and type");
118.         else
119.         {
120.             string result = "create table " + txttablename.Text + "(" +
query + ");";
121.             query = null;
122.             writer.WriteLine("Crt:" + result);
123.             writer.Flush();
124.         }
125.     }
126. private void tablecombobox_SelectedIndexChanged(object sender, EventArgs e)
127.     {
128.         try
129.         {
130.             string t = "select * from " + tablecombobox.Text + ";";
131.             writer.WriteLine("Sql:" + t);

```

```

132.         writer.Flush();
133.         DataSet ds = Deserialize(stream);
134.         WriteToStatusBar(ds.Tables[0].Rows.Count.ToString());
135.         userDataGridView.DataSource = ds.Tables[0];
136.     }
137.     catch (Exception ec)
138.     {
139.         MessageBox.Show(ec.ToString());
140.     }
141. }
142. private void btnDelete_Click(object sender, EventArgs e)
143. {
144.     if (MessageBox.Show("Sure you wanna delete this Table?", "Warning",
145.         MessageBoxButtons.YesNo) ==
146.         System.Windows.Forms.DialogResult.Yes)
147.     {
148.         string d = "drop table " + tablecombobox.Text + ";";
149.         writer.WriteLine("Delete_table:" + d);
150.         writer.Flush();
151.         btnRefresh.PerformClick();
152.     }
153. private void btnExecute_Click(object sender, EventArgs e)
154. {
155.     string s = rchtxtQuery.Text;
156.     writer.WriteLine("Sql:" + s + ";");
157.     writer.Flush();
158.     DataSet ds = Deserialize(stream);
159.     tabControl1.SelectTab(0);
160.     WriteToStatusBar(ds.Tables[0].Rows.Count.ToString());
161.     userDataGridView.DataSource = ds.Tables[0];
162. }
163. private void btnUpdate_Click(object sender, EventArgs e)
164. {
165.     iii= userDataGridView.ColumnCount-1;
166.     DataTable dt = userDataGridView.DataSource as DataTable;
167.     writer.WriteLine("Update:");
168.     writer.Flush();
169.     DataSet dss = new DataSet();
170.     DataTable dtCopy = dt.Copy();
171.     dss.Tables.Add(dtCopy);
172.     Serialize(dss, stream);
173.     stream.Flush();
174. }
175. private void tlStrpClose_Click(object sender, EventArgs e)
176. {
177.     writer.WriteLine("c:");
178.     writer.Flush();
179. }
180. private void closeDBToolStripMenuItem_Click(object sender, EventArgs e)
181. {
182.     writer.WriteLine("COL:");
183. }

```

```

184.     private void exitToolStripMenuItem_Click(object sender, EventArgs e)
185.     {
186.         Application.Exit();
187.     }
188.     private void button2_Click(object sender, EventArgs e)
189.     {
190.         writer.WriteLine("DB:" + new_database.Text);
191.         writer.Flush();
192.         button2.Visible = false;
193.         new_database.Visible = false;
194.         label6.Visible = false;
195.         button3.Visible = false;
196.     }
197.     private void button3_Click(object sender, EventArgs e)
198.     {
199.         new_database.Visible = false;
200.         button2.Visible = false;
201.         button3.Visible = false;
202.         label6.Visible = false;
203.     }
204.     private void btnRefresh_Click(object sender, EventArgs e)
205.     {
206.         openDB();
207.     }
208.     private void userDataGridView_CellMouseUp(object sender,
DataGridViewCellMouseEventArgs e)
209.     {
210.         if (e.Button == MouseButtons.Right)
211.         {
212.             this.userDataGridView.Rows[e.RowIndex].Selected = true;
213.             this.rowIndex = e.RowIndex;
214.             this.userDataGridView.CurrentCell =
this.userDataGridView.Rows[e.RowIndex].Cells[1];
215.             this.contextMenuStrip1.Show(this.userDataGridView, e.Location);
216.             contextMenuStrip1.Show(Cursor.Position);
217.         }
218.     }
219.     private void contextMenuStrip1_Click(object sender, EventArgs
e)
220.     {
221.         if (MessageBox.Show("Sure you wanna delete this Record?", "Warning",
222.             MessageBoxButtons.YesNo) ==
System.Windows.Forms.DialogResult.Yes)
223.         {
224.             if (!this.userDataGridView.Rows[this.rowIndex].IsNewRow)
225.             {
226.                 this.userDataGridView.Rows.RemoveAt(this.rowIndex);
227.                 btnUpdate.PerformClick();
228.                 btnRefresh.PerformClick();
229.             }
230.         }
231.     }
232. }

```



```

233.     }
234.
235.     //end of the class

```

figure 4.5 Source code of client

as shown in figure 4.5 client_1010 name space (line 12)
at line 29 openToolStripMenuItem_Click function raises the event when we choose Open_DB tool from the menu at this function we call openDB function (line 32) .

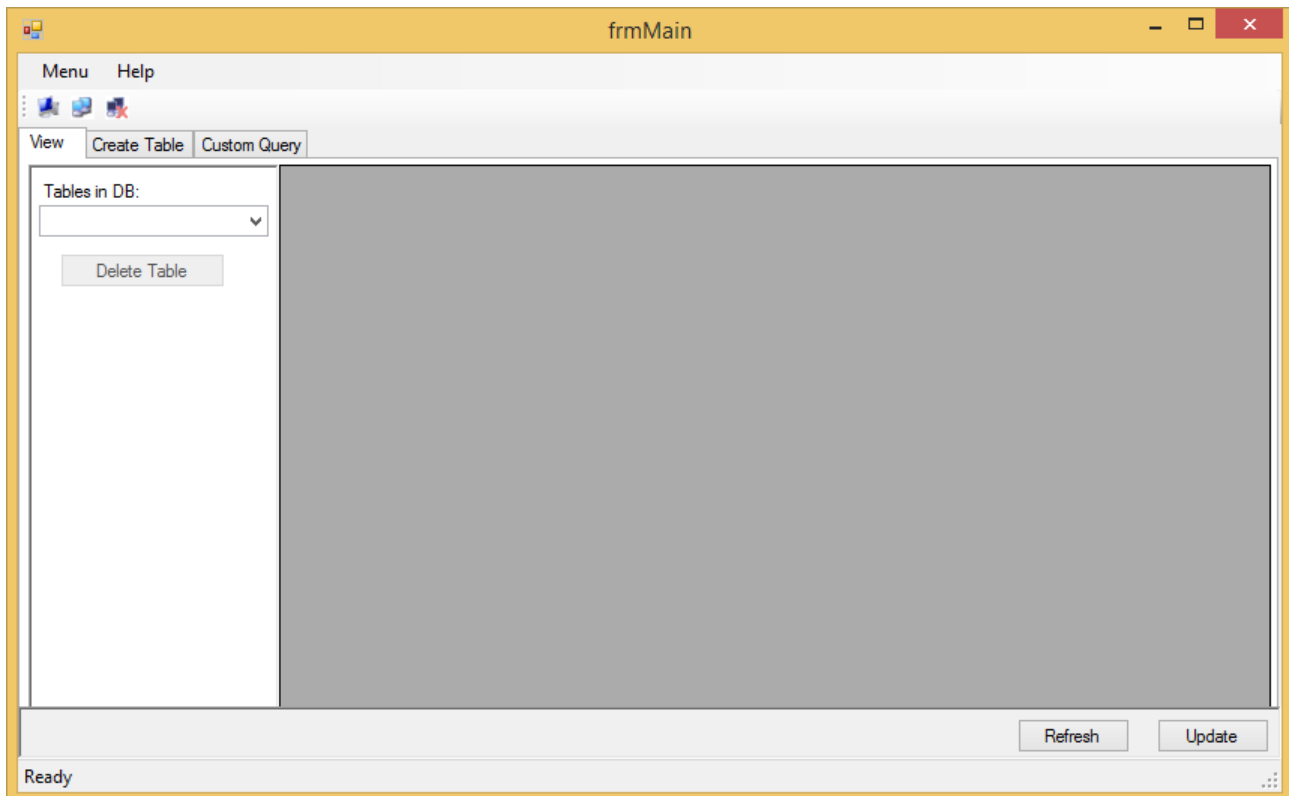


figure 4.6

so , what does this function do ? (line 34)

first we set value to null then see if the database name is null , then open database input box would appear if the user press OK then we set the database_name attribute to the value .(line 39 , 41)

after that the client send the message which start with the command Open_Ds to the server using the writer object (line 42,43)

to receive message from the server we have to Deserialize the stream and set it to a Data set (line 45)
then we write to the status Bar the number of rows contained by the first table of the data set(line 47) and clear the table combo box using clear method (line 48) so we can fill it by adding items (tables names)(line 50).

at line 51 we make sure that the data set contains at least one table then we can show the delete button by set Enabled property to true (line 53) , then we clear the selected index of combo box table (line 54).

the function `ConnectTo` (line 61) makes the connection to the server by taking two parameters `serverName` and the port on the connection would be created, so we create a `TcpClient` object and set the `serverName` and the port to it (line 63,67).
 at line 68 we write to the status bar that the connection is succeeded if it doesn't succeed we catch that exception and show a failed message (line 70 ,73)

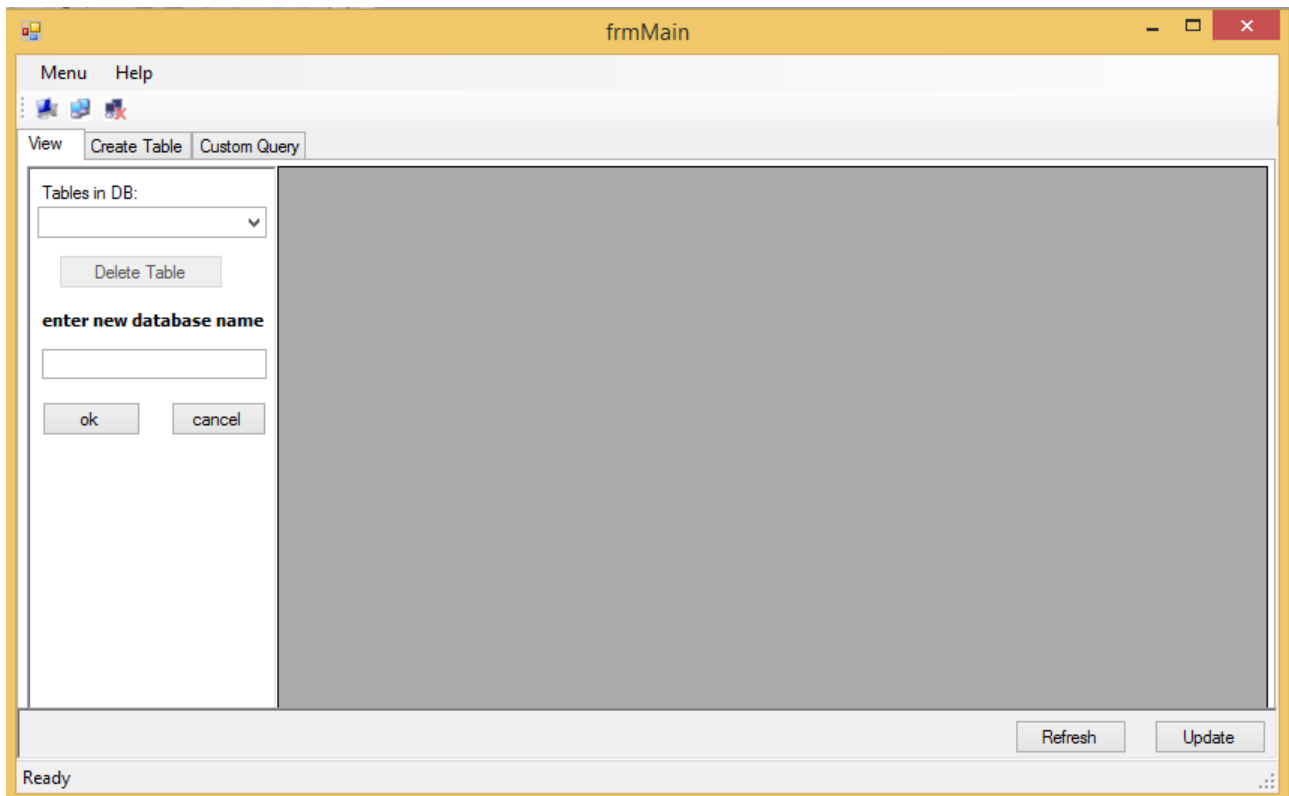


figure 4.7

to be sure that the connection is secure we get the stream we are using as a client and pass it to the `SecureStream` Class then using the `securestream` to make the `streamReader` and `stream writer` .(line 74 to 78)

at line 81 the function `WriteToStatusBar` takes the message which we sent to and write it to the status bar (line 83)

if the user want to make a connection he/she just clicks on on the connection icon/button that raises thae event then we use the connection function to connect to the server by passing the server name and the port to it (line 97)

if the user wants to make a new data base then he/she just clicks on the computer icon/button when that event raises a textbox to enter the name of the data base appears ok button and cancel button are appear as well , so what is happen in the source code is just we make them visible by set their visible property to true (line 99 to 105)

then the user press that ok button if he/she want to create that new database so what happens in the source code as following :

he writer send a message to the server beginning with the "DB" command and contains the data base name the user enters , after flush the writter we set the visible property to false again so all that

tools would disappear .(line 189 to 197)

well , what if the user want to create a new table or add a table to the data base ?

then the user need to open create table page and insert the table name , insert the column name and choose if it is a primary key he/she should check primary key CheckBox then choose the column type from the combo box as shown in figure (4.8)

when the user click the button add column the event btnAdd_click is raised (line 106) so , what is happen inside the source code we check if the query is no empty then we add ","(line 109) after that the query contents would be as the user insert the column name and type (line 110)

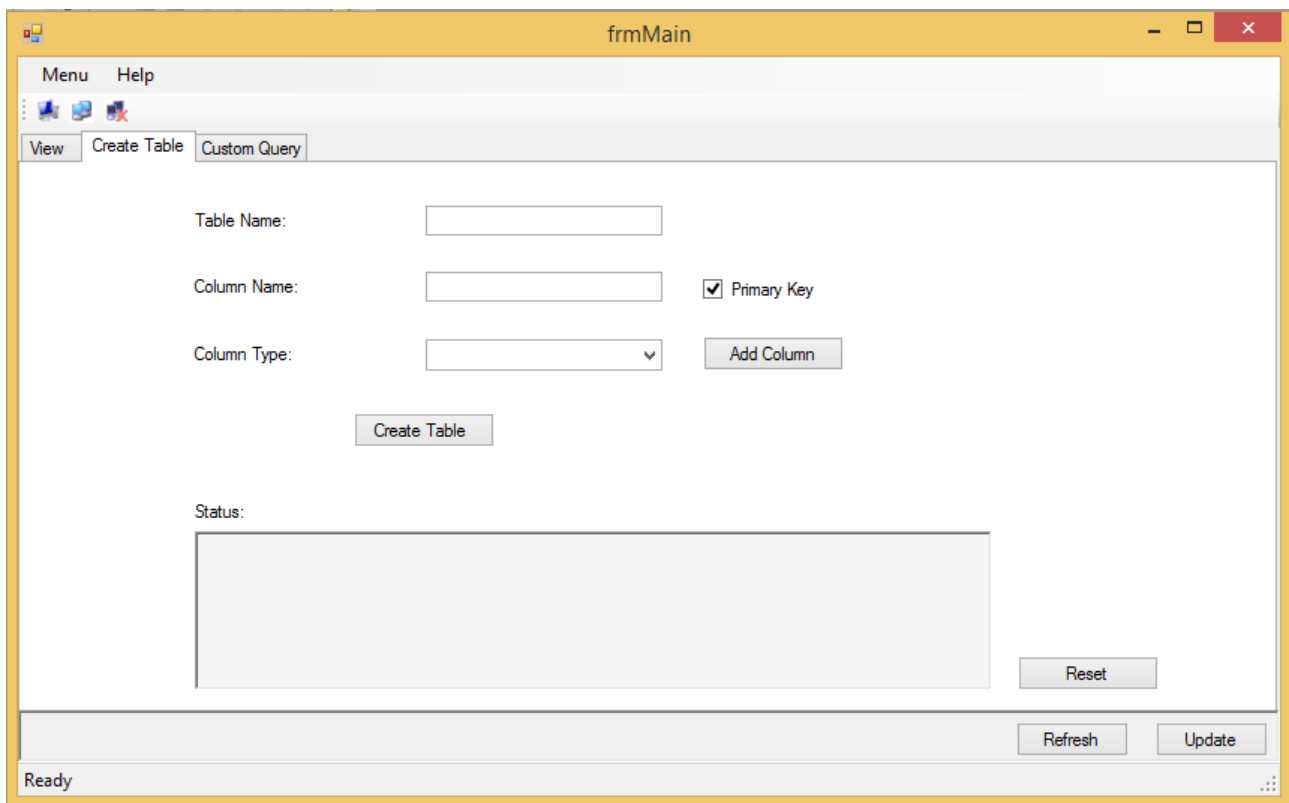


figure 4.8

then if the user checked the primary key we add that to the query as well then we empty all the text boxes in that page . (line 11 to 114)

but what takes place when the user press the create table button ? from line 121 to line 124 we put the table name and the query to the result string then send it to the server by the stream writer with the command word "Crt:" and the server would create that table to the data base as we discussed in the server class .

at the custom query page we can write down sql command into the rich textbox and execute it as soon as we click onto the execute query button .so ,what happens into the source code is shown in figure # starting with line 154 first we set the written command to the string s , then add the word "Sql" to it sending that to the server through the writer stream then then we get out from that page to the view page showing the results of the query in the Grid view (line 154 to 163)

if the user want to delete a specific table in the view tape there is a combo box shows the tables in the data base the user should choose one of them then he click on the delete Table button .

at line 143 there is the function which handle the event click so ,first we show a warning message

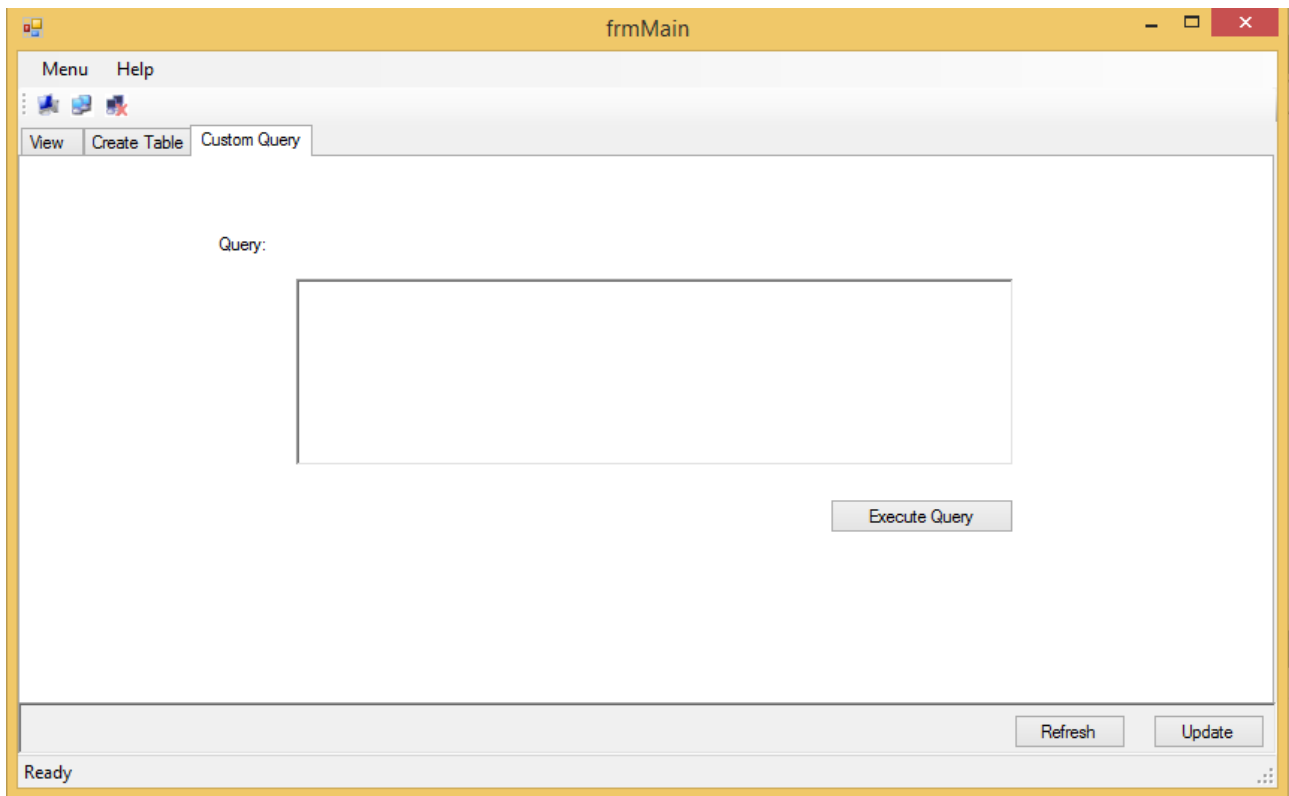


figure 4.9

insure if the user wants to delete the table if he/she press yes, then we set the chosen table name into a sql command "drop table " to the string d , then we send it to the server start with the command "Delete_table" then the server will delete it from the data base .
after that we reopen the data base using refresh handler event function (line 143 to 152).

when the user want to refresh the data base he/she can click on the refresh button then the event would call the openDB function (line 205 , 207)

if the user wants to make an update in the data base he can make it in the grid view shown on the view page , so what happens inside the source code when the user click update button is :
first we create a new datatable object and set the data source on the grid view to it so the data shown on the grid is a data tale now
then then we send the command "Update" to the server . the server will make the update in the data base using the adapter . we make a new data set and data table so we copy the new data table and add it to the new dataset ,then we call Serialize function so we can send the new data set to the server using the stream the new data set updated in the data base so finally we make the update .
(from line 164 to 175)

the data set shown in the grid view consists of rows and columns so if the user want to delete a spesific record he/she can right click on the index row and choose delete row .

that happen in the following event handler functions :

at line 211. userDataGridView_CellMouseUp function which responsible for selecting the row index .

first we make sure that the clicked button is the right button then select it by setting the select property to true ,then we make the current cell of the grid view as the row index which the cursor

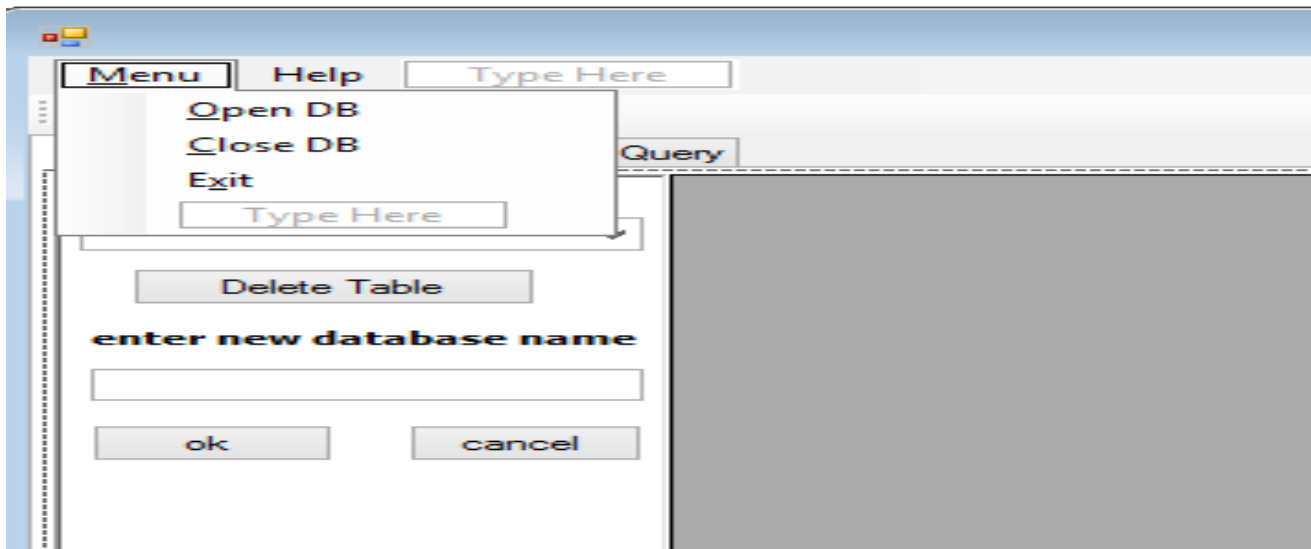


figure 4.10

stands on then we show the mouse location and cursor position . (line 211 to 220).

to delete the selected row we use the RemoveAt at method of Rows at data grid view (line 230)

finally to close the data base the user choose close DB from the menu in the source code the client send the command "c" to the server and the server closes the Data base (line 181 to 183)

eventually to exit from the application the user choose Exit from the menu and the application would be closed using the Exit method (line 187)

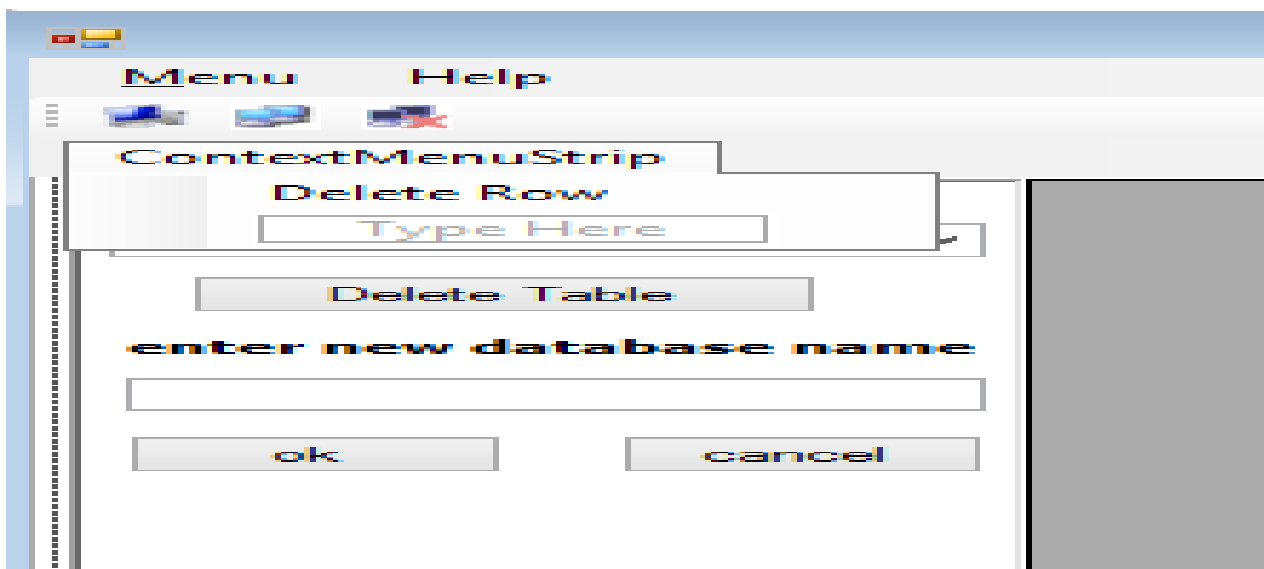


figure 4.11

References:

<http://www.codeproject.com/cs/internet/sockets.asp> for reference.

<http://www.codeproject.com/Articles/13575/Using-a-NetworkStream-with-raw-serialization-GZipS>

<https://www.codeproject.com/Articles/26332/Creating-a-secure-channel>