Vol. 21 No. 3 Candle Computer Report April 1999

Understanding the Dynamics of TCP/IP Performance

Transmission Control Protocol/ Internet Protocol (TCP/IP) is the standard network protocol of open systems in commercial, educational and government applications. TCP/IP is more than just a network protocol. It actually provides several major functions and components, including applications for mail, file transfer and network management.

Among the strengths of TCP/IP are universal interconnection and network technology independence. It allows heterogeneous systems such as UNIX, NT and MVS to communicate across a network. TCP/IP networks typically have a low cost of entry and offer a dynamic, connectionless routing methodology. Plus, TCP/IP can support different physical networks, such as Asynchronous Transfer Mode (ATM), Integrated Services Digital Network (ISDN) Token Ring and Ethernet.

With today's networked applications, understanding the performance characteristics and technical limitations of network technology is essential to any analysis of application performance. Without the network, software and hardware on multimanufacturer machines could not communicate meaningfully. And because of the rapidly increasing industry prevalence of TCP/IP for network communication, you should have a thorough understanding of its architecture and of factors that affect performance.

This article will guide you through the components and applications of the TCP/IP layers, as well as the dynamics of network connections and data flow. We will then examine TCP/IP monitoring and performance issues, including those related to IBM's MVS implementation of TCP/IP.

TCP/IP LAYERS

The modular protocol architecture for TCP/IP has its origins in the ARPANET, a network established by the United States government, and often is called the DoD (Department of Defense) model. TCP/IP uses a four-layer, or "stacked" model with many components and protocols interacting (see Fig. 1). The components in each layer communicate only with components in layers directly above and below them. Each component in the stack performs a specific function.

The bottom layer is for the network interface adapters, or data links. It exchanges data between a host and the network, and delivers data between devices on the same network. Examples of protocol standards functioning at this layer include IEEE 802.3 (IEEE Ethernet), IEEE 802.5 (IEEE Token Ring), X.25, Frame Relay and ATM.

(Continued on page 2)

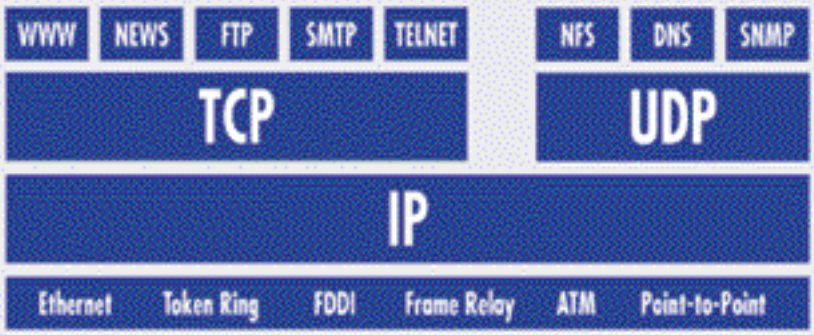

Figure 1 – The TCP/IP layered model

INSIDE THIS ISSUE

Design Patterns, Part 1

A design pattern is to the programmer what the playbook is to a football player: a tome of reusable tactics for use in the appropriate circumstances. A football play is a specific design that describes the role of each player and is likely based on experience. The same is true of a design pattern. Experience tells programmers that there are many glitches that recur in software design and they must be solved again and again. Therefore, why not have a software "playbook" that identifies these predicaments and provides documentation for ready-made plays?

Part one of this article will guide you through the origins and basics of design patterns, then focus on the observer design pattern that is demonstrated in coding examples. This is an appropriate one on which to base instruction because it is suitable for beginners and its wide application makes it a useful tool for the more advanced programmers.

HOMEBUILDING 101: THE ORIGINS OF DESIGN PATTERNS

The term "design patterns" refers to a landmark book published in 1994, of the same name, by four authors: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Their book was inspired by the work of Christopher Alexander. Interestingly, Alexander had nothing to do with computers – at least until 1994. He is a building architect by trade. In 1977, Alexander identified a pattern language and published it in his book *A Pattern Language: Towns/Buildings/Construction*. As he envisioned it, a client could develop his own architectures through pre-defined patterns. This thinking was

(Continued on page 5)

Understanding the Dynamics of TCP/IP Performance *(Continued from page 1)*

The Internet Protocol (IP) layer routes messages through internetworks. It delivers packets of data, or datagrams, on the network. When a datagram must move from one network to another, the IP layer ensures that it reaches the network for which it is destined. Actual delivery to the destination host is a function of the network interface layer. IP is commonly referred to as a "connectionless, best-effort routing protocol." IP by itself adds no reliability, flow control or error recovery to the process.

The TCP layer handles host-to-host data delivery and end-to-end data integrity. It does not route any data through the network. TCP provides reliable communications between processes that run on interconnected hosts. This host-to-host communication functions independently of the network structure. The TCP layer on one host communicates directly with the TCP layer on another host, no matter if the hosts are on the same network or on different networks. TCP establishes a connection between hosts that is maintained for the duration of the data exchange.

On the same layer as TCP is User Datagram Protocol (UDP), which provides a lower-overhead alternative to TCP for applications that do not require reliable delivery. UDP, or the "connectionless" protocol, does not guarantee delivery. Instead, applications in the next layer maintain the integrity of the data. As a result, UDP requires less overhead than TCP.

The TCP/IP application layer is included as a standard in most TCP/IP implementations. Components of the TCP/IP application layer include File Transfer Protocol (FTP), Telnet, Simple Mail Transmission Protocol (SMTP), SNMP and NFS.

Simple Network Management Protocol (SNMP) is an example of an application that runs on the application layer above UDP. It provides network management data via alerts. Network File System (NFS) is another application that uses UDP to enhance performance. It performs its own reliability functions as well.

MAKING CONNECTIONS

Now that we have examined the layers of TCP/IP, let's look at how these layers interact to transfer data across the network. It is important to understand the process of data flow through TCP/IP to better manage network performance.

A socket is the term for the application programming interface (API) between the TCP/IP transport layer and application processes running at both the client and the server. A connection between two applications on a network is defined by the sockets at either end. A socket is an abstract term that describes the end-point of network data exchange.

These connections provide a two-way path between application processes. A TCP/IP network may be viewed as a group of client and server socket pairs with network traffic flowing between them. In a typical business network, the majority of socket traffic is light and short in duration. The minority is longer in duration, but it may account for the bulk of the data traffic.

Figure 2 shows the flow of data through the TCP/IP layers. First, the TCP layer receives the stream of data from the application process. TCP may fragment the data into segments that conform to the maximum segment size (MSS) of IP. Then, IP may – if necessary to observe physical network limitations – further fragment the segments to prepare datagrams sized to conform to the physical network restrictions. These datagrams are divided into maximum transmission units (MTU).

The network transmits the data in bits. At the receiving host, the network reconstructs the datagrams from the bits received. IP receives datagrams from the network. When the datagrams have been fragmented for transmission, they must be reassembled into the original segments. TCP presents the data segments to the application layer in a data stream.

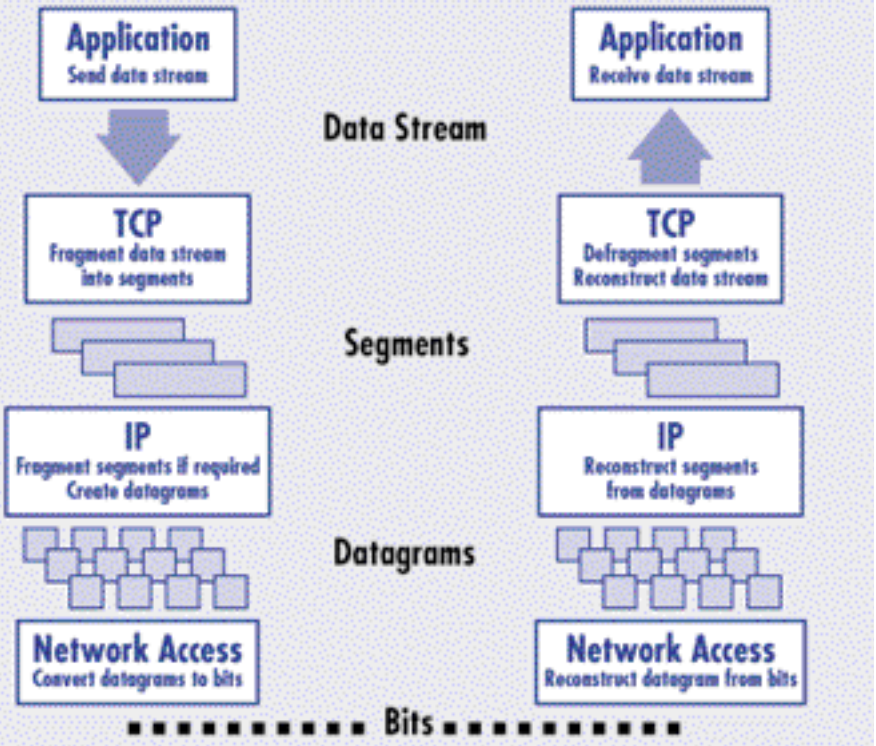

Figure 2 – TCP/IP data flow

Impact of Fragmentation

Some network types may limit the size of the MTU. For example, the largest MTU on Ethernet 802.3 is 1,492 bytes, but for Token Ring it is 2,000. Fiber Distributed Data Interface (FDDI) networks use 4,352 bytes. As shown in Figure 2, data packets that exceed the size of the largest allowable MTU must be fragmented by the sender and reassembled by the receiver.

To reassemble the fragments, the receiver must allocate a buffer in

storage when the first fragment arrives. A buffer is a network device for holding data packets. As subsequent fragments arrive, the receiver also copies them into the buffer. Once all the fragments arrive, the complete datagram is restored.

As a rule of thumb, TCP/IP consumes a consistent amount of CPU resources for each packet – regardless of the packet size – but draws additional resources based on packet size. In general, the draw will be lower as the number of transmitted packets decreases. For the sender, the resource cost of fragmentation is for creating additional packets and retransmitting datagrams when packets are lost in the network. For the receiver, the cost is for reassembling additional packets and using buffers to hold them.

FLOW CONTROL

With our understanding of how data is transferred, we can now begin to explore how it is controlled. TCP/IP's data flow control impacts performance and ensures orderly data transmission. It also optimizes the use of network bandwidth, and protects the network's physical devices and receiver from too much traffic.

The mechanism that provides protection is called the TCP sliding window. Remember that clients and servers are both senders and receivers. Because traffic flow is two-way, the sliding window exists on both ends of the connection. Remember that this flow control mechanism is managed at the clients and the servers; the IP network itself is connectionless and has no flow control.

The sliding window allows the senders and receivers to announce to each other the Maximum Segment Size (MSS) they are able to manage. It allows for the dynamic adjustment of MSS as indicated by the sender or receiver. The result is both a more effective use of network bandwidth and protection of network resources.

Three-Way Handshake

A simple protocol doesn't make the most optimal use of network bandwidth because each packet isn't sent until the previously sent packet's receipt is acknowledged. If the acknowledgement is not received in a predefined time-

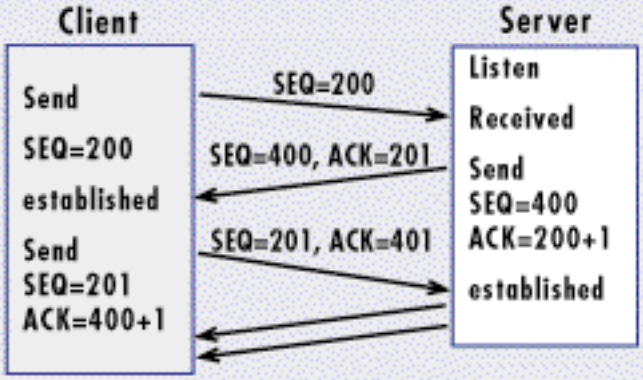

Figure 3 – TCP/IP three-way handshake

frame, the packet is retransmitted. That's why TCP/IP uses what is called a three-way handshake to implement the sliding window (see Figure 3). At the sender, TCP assigns a sequence (SEQ) number and begins a timer for each segment sent. TCP does not necessarily wait for an acknowledgement (ACK) of every sent segment before sending more data, but each segment is eventually ACK'ed by the receiver. In addition, each ACK message will include the window size that the receiver can handle at the time. The ACK carries the sequence number of the last segment received. If the sender does not receive an ACK within a specified time, then retransmission occurs. If the network is slow, there may also be retransmissions of ACKs. A redundant transmission is simply discarded.

Retransmission can adversely affect performance because it is a redundant workload and an inefficient use of network bandwidth. A high number of retransmissions indicates network problems. A bit later in this article, we will discuss how to monitor for retransmissions.

Congestion Avoidance

If congestion from excessive data traffic exists within the network, the sliding window size may decrease from the MSS to the smallest size – a single TCP segment. MSS may increase once network congestion has been relieved. Congestion is often indicated by one or more segment retransmissions due to a time-out. If the retransmission count (typically 16) is exceeded, the TCP connection is broken.

TCP also provides protection of the buffers at the receiver. TCP tells the network how much space is available in the buffer. The receiver can protect itself by notifying the sender of a window size of zero. This informs the sender that there is no space available in the receiver buffers.

To relieve network congestion, devices such as routers are allowed to drop packets when information buffers overflow. This is a standard means of congestion avoidance but it cannot help when demand outweighs network capacity. Typically there are only two ways to reduce congestion in that event: increase capacity via more bandwidth, or reduce network demand through application optimization and tuning.

Data Loss

Even with flow control, a datagram, or segment, may be lost on its travels through the network. This occurs for a variety of reasons:

- Bit transmission errors may occur during transmission;
- Congestion avoidance mechanisms, as described above;
- Absence of path to the destination due to network congestion or a physical connectivity error;
- Fragmentation process.

Data loss will often result in retransmission of data. For example, if a fragment is lost, all the data packets in the original datagram must be retransmitted, even if most of the data in the previous transmission arrived intact. As mentioned earlier, retransmission is undesirable from a performance standpoint.

MONITORING AND IMPROVING PERFORMANCE

To achieve smooth data flow and optimal TCP/IP network performance, you need to monitor activity within the network. Fortunately, TCP/IP includes some commands that are useful for analyzing performance. These network analysis commands are common to most TCP/IP stack implementations.

- The NETSTAT command is highly useful because it reports TCP/IP connection data and protocol statistics. NETSTAT displays will typically show local and foreign addresses, connection status, packets sent and received, and statistics on fragmentation and retransmission activity, among other things.
- The PING command is another vital TCP/IP analysis tool. PING uses a protocol called Internet Control Message Protocol (ICMP) that runs on the same layer as IP. It is used to verify connections between hosts by sending ICMP echo packets to a specified IP address or host name. After waiting for specified intervals of time for each packet it sends, PING reports the number of packets it sends and receives. The PING command commonly offers options to specify the size of packets sent, whether or not to fragment the message, and to record the routes of the outgoing and returning packets.

PING also reports the round trip times of the data packets – a good performance indicator. By varying the amount of data sent with the PING command, and by issuing PING to various hosts, you may also get a better idea of the performance of the [underlying] network.

IBM and TCP/IP

The IBM implementations of TCP/IP stacks on MVS present performance challenges. In addition, there are multiple versions of TCP/IP available for MVS and it can be confusing to decide the best MVS TCP/IP stack to run. By the time IBM began developing V3R2 of the TCP/IP stack, it had pushed the previous TCP/IP stack architecture to its limits. Out of necessity to improve performance, IBM developed a rewrite of the TCP/IP stack. Because of the project's size, the rewrite was done in two phases. The first phase was available in OS/390 V2R4. Phase two of the rewrite was available in OS/390 V2R5.

Communications Server for OS/390 (CS/390) is the packaging of the IBM communications services for the MVS operating system. With CS/390, the systems network architecture (SNA) stack and the TCP/IP stack will share common code components. The most important one is the Communications Storage Manager (CSM). The CSM reduces the number of required data moves for the OS/390 TCP/IP stack and makes the stack much more efficient. According to IBM benchmarks, the instruction path length reductions for send/receive operations has been 70 percent or more compared to the older versions of TCP/IP [SIMM98]. The latest IBM benchmarks are available on the company's Web site found at www.raleigh.ibm.com/tcm/tcmprod.

MVS TCP/IP Performance

There are some general rules of thumb that may be applied to improve TCP/IP performance on MVS. If possible, set the REGION size for the TCP/IP address space to zero K. This provides the maximum available storage to TCP/IP. Set the MVS dispatching priority or importance for TCP/IP to a level comparable to VTAM. In goal mode, consider putting both TCP/IP and VTAM in the SYSSTC service class.

It is important to make the client and server TCP window sizes equal. Recommended TCP window sizes are 16,384, 32,768 or 65,536. Choose sizes to optimize network resources while avoiding fragmentation. Generally, performance will improve as MTU and MSS increase in size.

There are important configuration files for the set-up and tuning of the IBM TCP/IP stack on MVS.

- TCPIP.xxxxxxxx.TCPIP contains information such as buffer definitions, network controller definitions, server port definitions, home IP addresses and gateway definitions.
- TCPIP.TCPIP.DATA contains TCP/IP host names and domain name server information.
- TCPIP.FTP.DATA contains parameters that drive the configuration and set-up of FTP, such as NCP and BUFNO mentioned below.

An excellent source of information for MVS TCP/IP performance and tuning is IBM manual SC31-7188, the *IBM TCP/IP Performance Tuning Guide.* This manual contains more detail on the parameter files and configuration options as well as tuning examples and performance benchmarks.

FTP performance on MVS TCP/IP can be enhanced in a number of ways. As MTU increases, MVS CPU resource usage will typically decrease. MVS throughput will increase and MVS CPU will decrease as the workstation window size increases. The recommended workstation window size is 32 KB or 64 KB. Additionally, MVS throughput will increase and MVS CPU will decrease as the MVS data buffer size increases. The recommended MVS window size is 32 KB or 64 KB.

For an MVS FTP server, MVS throughput will increase for inbound data (PUT command) as the number of specified disk I/O buffers increases. The number of buffers is specified by the BUFNO parameter in the TCPIP.FTP.DATA file. It is recommended to specify BUFNO at 35 or higher. For an MVS FTP client, throughput increases for the GET command and inbound data as the number of channel programs (NCP) parameter is adjusted. It is recommended to set NCP to a value between 10 and 20.

File system characteristics also can greatly impact FTP performance. Relevant parameters include cache or non-cached Direct Access Storage Device (DASD), file blocksize, and type of DASD. In general, MVS FTP throughput increases as MVS dataset blocksize increases. A reasonable dataset blocksize is half the size of a DASD track.

SUMMARY

Throughout the last few years, many business enterprises have adopted TCP/IP as the network protocol of choice. The protocol has been around for more than 25 years, but only recently has it assumed widespread acceptance beyond applications in government and education.

The growing prevalence of TCP/IP compels us to understand its architecture, its data flow processes, its congestion avoidance mechanisms and its data transmission dynamics. With that foundation, we can begin to do a better job tuning TCP/IP networks for optimal performance.

Ed Woods
Candle Corporation

Design Patterns *(Continued from page 1)*

a major evolutionary step in the field. Alexander originally expected it to enable ordinary citizens to design and construct their own homes. Unfortunately, that ambitious goal was never fully realized. The concept did, however, liberate the architect from the "empty architectural dogma," which offered only ridged structures and lacked design with reusable elements.

Since *Design Patterns* was published, a similar evolution has occurred in software engineering. We have gone from cut-and-fit design, which encouraged excessive reinvention, to the art of composing reusable building blocks. The patterns themselves are not necessarily new, but the book introduced methods of classification and sophisticated insight. Most significantly, it suggested a new way of thinking about software design.

In this context, what are design patterns, exactly? Succinctly stated, they are reusable elements of design which should not be confused with reusable code. Patterns are small canned definitions that can be coded any number of ways. Not every design element can be considered a pattern. For an element to earn this distinction, it must meet certain criteria, the most important one being a common, readily recognized design recurrence. The book *Design Patterns* captures those recurrences and presents them like material swatches for furniture.

Today, design patterns are widely used and exist in most modern application tools. From Microsoft's COM to Sun's Java, you will see pattern terms like "factory method" or "observer." There can be no denying that patterns have become a permanent fixture in software. It can be argued that they are the most reusable products ever to come out of object-oriented programming.

ELEMENTS OF DESIGN PATTERNS

To understand what makes design patterns reusable and how they are further defined, let's examine the four elements that compose them.

Name

Patterns must be individual, self-contained components. As such, they are named with apt visual metaphors so that they may be described in a higher level of abstraction. For example, the Factory Method conjures up images of an assembly line with workers busily putting together widgets as they pass by on a conveyor belt. That is exactly what the Factory Method design pattern is: A software conveyor belt that spits out new objects. Attaching proper nouns to patterns lets us describe them in higher terms without losing sight of what they do.

Problem

The problem is the situation to which patterns apply. A problem is sometimes specific and other times general. It describes classes or object structures symptomatic of the design.

Solution

This describes the design, component relationships, responsibilities and collaborations. It is the core of patterns.

Consequences

No solution is free of its own problems. There are always trade-offs. To give the user a complete understanding of application, the authors of *Design Patterns* include a critical analysis of each pattern.

DESIGN PATTERNS VS. ALGORITHMS

At first glance, design patterns seem to be nothing more than glorified algorithms. It is true that some patterns include algorithms, but upon closer inspection it is clear that patterns operate in a different domain.

What's the difference? Algorithms offer narrowly focused sequences of instructions, independent of language. Design patterns define object structures, relationships and roles, and maybe some code. Most importantly, they solve *design* problems arising from object-oriented programming, but they do not necessarily address the coding problems themselves. In other words, an algorithm is a "code in a can" and a design pattern is a "design in a box" with some assembly required.

For example, if you need to implement a hash table, then you employ a hashing algorithm. The algorithm describes the individual operations the code must carry out to hash strings into keys and to ensure those keys are unique. If you then want that hash-table object to broadcast changes, such as the insertion of a new entry, then you would employ the Observer design pattern. This pattern, which we will examine later in this article, tells how to classify the objects, what methods must be supported and how the objects should communicate with one another.

DESIGN PATTERN CLASSIFICATIONS

Patterns are classified by purpose, function and scope, whether they apply to classes or objects. Purpose includes three classifications: creational, structural and behavioral.

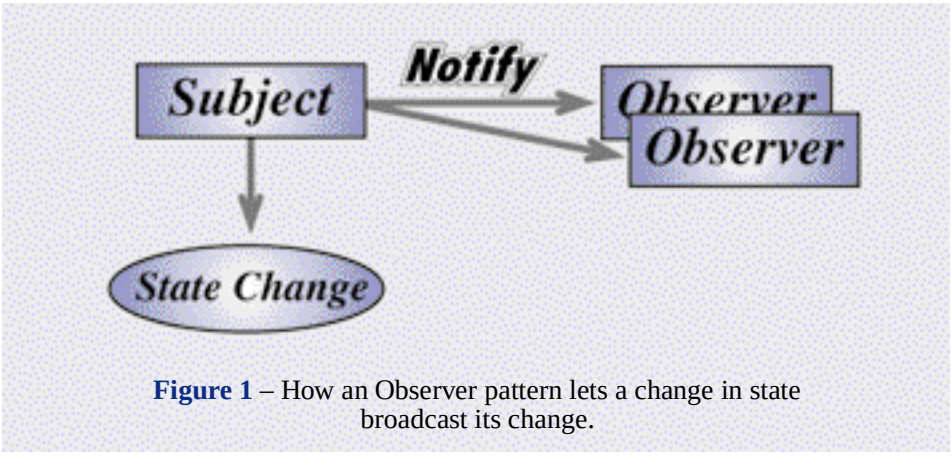

Figure 1 – How an Observer pattern lets a change in state broadcast its change.

Creational Patterns

Creational patterns concern the production of objects. As the book puts it, "[they] abstract the instantiation process." The Singleton is a popular example of a creational pattern.

Structural Patterns

Structural patterns make interfaces into larger structures. These patterns tend to rely on class inheritance. Multiple inheritance could be considered an example of a structural pattern. Here, you take two or more existing interfaces and combine them into a new structure.

Behavioral Patterns

This is where the waters are muddied between patterns and algorithms. The reason is that these patterns do concern themselves with algorithms as well as higher-level design aspects. But instead of just describing the flow of control, which is typical in traditional procedural programming, behavioral patterns describe how objects interconnect. They also tend to use object composition instead of class inheritance, but inheritance is sometimes leveraged to distribute behavior between classes.

THE OBSERVER PATTERN

The Observer pattern is of the behavioral type. It describes how a group of objects, rather than a single one, can achieve an easily understood publish-and-subscribe design. By rigidly assigning object roles, the application programmer is free to build higher level functionality. Without the insight that the pattern brings, it is easy to wind up with a lump of indecipherable code. The Observer pattern provides consistent rules, which are easily understood by anyone maintaining the pattern. Because of this pattern's practical use, it is worthwhile to devote the remainder of this article to examining it.

Application

The Observer pattern is a method of notification that lets a change in state broadcast its change to one or more clients. It defines a one-to-many relationship between objects and specifies the communication mechanics between these objects. In a nutshell, it notifies "observer" objects when a "subject" changes. Figure 1 illustrates the relationships.

The Observer pattern is sometimes called Dependents, or Publish-and-Subscribe. It is very useful in client-server applications in which multiple clients are allowed to register their interest in certain events. In addition, it can easily become the structural basis of an entire application. This is because the Observer brings a level of clarity to the design that probably would not exist otherwise.

Besides its clarity, it's also very flexible. Even though the design targets a one-to-many relationship, a many-to-many relationship can be achieved by a relatively small modification. Look for more detailed instructions on this modification later in this article.

COMPONENTS AND STRUCTURE OF OBSERVER

There are basically two classes and two objects of the Observer pattern. The classes are abstract and can therefore be extended to suit the needs of the application. These classes are: (1) the subject, the component that does the publishing; and (2) the observer, the component that does the subscribing. The two objects are the concrete implementations of these classes, which are the components that the application ultimately instantiates.

```
/* ********************************************** */
/* Subject base class.                            */
/* Each derived subject is responsible for        */
/* monitoring state changes.                      */
/* ********************************************** */
abstract class Subject
{
        private    Vector      ObserverList     = new Vector();

        /* ********************************************** */
        /* Attach the observer to this subject.           */
        /* This means the observer will start             */
        /* receiving events.                              */
        /********************************************** */
        public void attach(Observer observer)
        {
                ObserverList.addElement(observer);
                return;
        }
        /* ********************************************** */
        /* Detach the observer from this subject      */
        /* ********************************************** */
        public void detach(Observer observer)
        {
                ObserverList.removeElement(observer);
                return;
        }
        /* ***********************************************  */
        /* Call the update function of each observer */
        /* attached to this subject.                      */
         /* ********************************************** */
        protected void notify(Object event)
        {
                Enumeration Observers
                                = ObserverList.elements();
                Observer        observer = null;
                while(Observers.hasMoreElements())
                {
                        observer = (Observer)observers.nextElement();
                        observer.update(event);
                }
                return;
        }
        abstract public void getState();
        abstract public void setState();
}
```

Figure 2 – A Java example of a subject base-class definition.

```
/* ******************************************************** */
/* The Users class wraps a hash table that contains */
 /* all active user profiles.                              */
/* ******************************************************   */
class Users extends Subject
{
        private Hashtable UseridTable = new Hashtable();
        public void login(String userid, Profile profile)
        {
                profile.LoginTime = new Date();
                UseridTable.put(userid, profile);
                notify(userid);
                return;
        }
        public void logout(String userid)
        {
                UseridTable.remove(userid);
                return;
        }
        public void getState()
        {}
        public void setState()
        {}
}
```

Figure 3 – A derivative of the subject class that watches for changes in the user hash table.

```
abstract class Observer
{
        // ****************************************
        // This is called by the subject when it
        // has an alert to announce.
        // ****************************************
        abstract public void update(Object event);
}
```

Figure 4 – The abstract base definition for an observer.

```
class UserListener extends Observer
{
        private Subject subject    =    null;
        public UserListener(Subject subject)
        {
                this.subject    =    subject;
                subject.attach(this);
        }
        public void update(Object event)
        {
                System.out.println("\nUser " + event +
                        " has logged on.");
                return;
        }
        // ************************************************
        // A finalizer is provided so the subject will
        // not notify an observer that's no longer there.
        // ************************************************
        protected void finalize()
        {
                subject.detach(this);
        }
}
```

Figure 5 –The instantiator of the observer object supplies a reference to the subject.

Let's take a look at these classes in more detail.

The Subject

The subject is what has the observer's attention. The base subject class is implemented as an abstract class and is extended by the application to do whatever the application needs it to do. In the programming example shown in Figure 2, the subject will be a user hash table which tracks users who log into a system. The base class will provide the methods that communicate to any observer so that the application can focus on its own communication needs. When the base class wants to send a change in state to the observers, it simply calls a local function in the base class.

Figure 2 is a Java example of a subject base class definition. The code is built to track and notify a list of observers, but since this is an abstract class it has no idea what is being observed. That is answered by the implementation that extends the Subject class.

The Concrete Subject

This class completes the implementation of the subject. It inherits Subject and supplies the code that monitors functions such as a log or the change in the state of CPU utilization. In our example, it is the user hash table that the subject manages. When someone logs in, the observers are notified.

The benefit to the programmer is that all of the notification tasks occur in the abstract base class. All the programmer has to do is supply the application code and call "notify" when it is needed.

Since this particular application does not require a "polling" of state (you always know when a state change occurs), getState and setState are not implemented. Therefore, this class does not provide the method bodies for those two functions. Part of the pattern definition, however, does include a polling of the subject's state by the observer, but this is not always necessary. It is interesting to note that the Observer pattern is so consistent in its design that even the polling mechanism, as described in the book, delays the actual update so that it too is ultimately an event-driven process from the observer's point of view.

The Observer

The observer is the object that seeks notification when a state change occurs. It has a passive relationship to the subject except when it needs to call the getState function. The observer must attach itself to the subject. When an event happens, the observer's update function is called.

Figure 4 is the abstract base definition for an observer. The application then inherits this class and provides the application-specific code. Basically, the observer is a listener that needs to supply only one function: Update.

The Concrete Observer

This code completes the implementation of the observer class. Here you would provide the application code. Remember, the object is responsible for attaching itself to the subject in which it is interested. One remaining problem, though: how does the observer reference the subject object? In other words, which component supplies the access to the subject? The answer depends on the application. In the Figure 5 example, it is up to the instantiator of the observer object to supply a reference to the subject.

In the example code, a message is printed when a user logs in stating who the user is.

CONCLUSION

This concludes part one of our examination of design patterns and the most practical one of the behavioral type: Observer class. In part two, which will appear in the next issue of CCR, we will discuss implementation issues related to the Observer design pattern, then turn our attention to key creational patterns, including Singleton, Abstract Factory and Factory Method. These creational classes are fundamental design patterns and are useful in everyday programming. Like the Observer, they show how essential customization is in today's programming environment.

Ray Walker
Candle Corporation

Further Reading

Design Patterns: Elements of Reusable Object-Oriented Software Authors: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides Addison-Wesley, 1994

Advanced C++: Programming Styles and Idioms Author: James O. Coplien Addison-Wesley, 1992

Web page on Dr. Christopher Alexander *www.math.utsa.edu/sphere/salingar/Chris*

Give Your Feedback to AF User Group Advisory Committee

You have a voice in developments to Candle Automated Facilities (AF) products, including AF/OPERATOR® and AF/REMOTE®. That voice is the AF User Group Advisory Committee, composed of representatives from several companies in the United States, Canada, the United Kingdom (UK), Germany and Switzerland.

"The Committee supplies AF-related instruction for publication in the CCR and maintains a diskette of user-supplied code and documentation," says Bell Atlantic's Michael Gelcius, the Committee's president and user liaison. "AF product users can use the instruction and the disk to enhance Candle's automation products."

The diskette is regularly updated with information submitted by users around the world. The current version contains new information submitted by users in the United States, Canada and the UK.

The Committee also helps to orchestrate an annual AF user conference, where it recommends enhancements to AF products based on user input. The three-day event, free to Candle customers, offers presentations from users, product demonstrations and workshops designed for basic and advanced users alike. For example, topics from the 1998 conference included:

Introduction to Trap Coding

Automation in A Sysplex Environment

The Dollars and Cents of Automation Efforts

OMEGAVIEW® to OMEGAVIEW II® Conversion Case Study

Getting Started with POVI

Using Voice and Internet E-mail to Augment Automation Applications

The North American conference is slated for August, 1999. The UK conference is scheduled for September.

Contact the Candle Training & Education Dept., U.S. (888) 332-9226, e-mail: training_and_education@candle.com, to get a copy of the diskette, to submit items for it or for publication in CCR, to suggest product enhancements and to get more information about the next User Conference.

Look for technical tips or articles from the Committee, as well as conference details, in upcoming issues of CCR.

Don't miss the 1999 Candle Performance Seminar

Candle will host the 12th annual Performance Seminar, May 17-18 at the Arabella-Sheraton Hotel Bogenhausen in Munich, Germany. Topics include messaging middleware, OS/390, databases, subsystems and distributed systems.

The price is 1,950 Deutsche Marks (excluding value-added tax). It includes admission to the sessions, full documentation, luncheons and dinner party.

To reserve your place, or for more information, contact Brigitte_Herbst@ candle.com or Simone_Merk@ candle.com or fax to Candle GmbH, attention – Simone Merk: 49 89 54 55 4 119.

ISSN 1071-2976