Active Transceiver Design Pattern for Data Communication Applications

Omar Bashir and Mubashir Hayat

Communications Enabling Technologies, Software Technology Park, Islamabad, Pakistan
E-mails: obashir@yahoo.com, ms_hayat@hotmail.com

Abstract: *Developers of distributed applications need to write a significant volume of similar code to configure communication components and to achieve reliable and efficient communications in these applications. Moreover, different development environments and operating system utilities provide different interfaces and services to the communications protocol stack. Recently a significant body of work has been conducted in developing extensible object oriented design patterns to document and disseminate solutions to various recurring problems of communicating data between components of distributed applications. Reliable and efficient communications is achieved through the use of event driven and multithreaded components. This paper describes the Active Transceiver design pattern that focuses on the execution of distributed application components in a multi-threaded environment.*

Keywords: *Design Patterns, Object Wrappers, Object Oriented Design of Data Communication Applications*

1. INTRODUCTION

Most development environments and operating system utilities provide mechanisms to facilitate communication between distributed applications. However, in all distributed systems, programmers have to write significant volume of similar code to configure communication components to achieve reliable and efficient communications. Furthermore, any change in the underlying communications technology may also require significant reprogramming of the entire system or at least at the interfaces between the application and communication components. Some of these issues related to configuration of communication components have been resolved by building wrappers over basic communication facilities provided by the development environments. Issues related to efficient and reliable communications are resolved through either event driven programming or by decoupling communications software from application components by making them execute in separate threads. Aspects related to extensibility are largely handled through object oriented programming of wrappers over communication components, event handlers and thread management mechanisms.

Design patterns allow the designers to capture the static and dynamic structures of recurring solutions when producing applications in a particular context. Recurrence adds validity to a pattern as several applications of a pattern reflect directly upon the quality of the solution it presents [1]. Design patterns are descriptions of key aspects of common design structures that form elements of reusable object oriented design. These include the description of the participating classes, collaboration among objects of these classes and their responsibilities [2]. In addition to documenting the successful practices in a specific domain, design patterns allow communication of and enable the reuse of architectural design information.

Because of the similarities in the structure and programming of distributed applications, appropriate design patterns can significantly facilitate the development process of such systems. This paper presents the Active Transceiver design pattern that focuses on the execution of a distributed application in a multi-threaded environment. Key advantages achieved through the practical use of the Active Transceiver design pattern include efficient and reliable communication as well as the extensible design and implementation of the supporting communications components. Active Transceiver also uses components based on some existing patterns for data communications systems. Furthermore, Active Transceiver describes components that permit object marshalling by value.

2. DESIGN PATTERNS FOR COMMUNICATIONS SOFTWARE

System software provides essential services and mechanisms used by the higher-level application software. Several factors such as efficiency, portability and lack of functionality complicate the cross platform reuse of system software. Therefore, it may not be possible to directly reuse algorithms, detailed design, interfaces or implementation of system software developed for one platform on another platform. However, application of appropriate design patterns allows the reuse of architectural information and development expertise at the system level [3], [4].

Mechanisms for achieving data communication in distributed applications can be divided into two broad categories of event driven and multi-threaded mechanisms. In multi-threaded mechanisms, application and communication components are decoupled by making them execute in different threads. The data are exchanged between the application and communication components through data structures shared between them. Active Object is an example of a design pattern

that uses multithreading to decouple method execution from method invocation [5].

In event driven mechanisms, event handlers perform application specific operations. These event handlers are invoked when respective events are detected by the sources of events (i.e., timers, communication ports etc.). Event based mechanisms are further divided into reactive and proactive mechanisms [3]. In reactive mechanisms, the application informs the system software of the event handlers to be invoked when a respective I/O event occurs. Event handlers execute without blocking. Reactor is an example of a design pattern based on the reactive I/O mechanism [3], [4], [5]. On the contrary, proactive mechanisms allow applications to proactively initiate I/O operations or general-purpose event signaling operations. The invoked operation proceeds asynchronously and does not block the caller. When an operation completes, it signals the application. The application then executes a completion routine [3], [4]. Proactor is a design pattern that allows applications to invoke asynchronous operations that are executed by the OS on behalf of the application. The OS notifies the application once these operations are completed. This allows the application to have multiple operations executing simultaneously without requiring the application to spawn a corresponding number of threads [6].

I/O based on synchronous multithreading, reactive and proactive mechanisms have their respective advantages and disadvantages [6]. Synchronous multithreading allows development of simpler application level code. Moreover, when a device is blocked for I/O, most operating systems put the thread managing that device to sleep. However, there are overheads involved in context switching, in increased synchronisation in the use of critical sections and in copying data between different threads. The latter is usually alleviated by copying pointers in shared memory.

Reactive model presents a potentially lower overhead due to coarse grained concurrency control, i.e., use of single threading requiring no synchronisation or context switching. Furthermore, this model promotes modularity through the decoupling of application logic from the dispatching mechanism. However, implementation of the reactive model can be complicated as the application should not block while servicing simply one task. Also, applications constructed on the reactive model do not efficiently utilise support for concurrency provided by the platforms [6].

Benefits derived from the proactive model include an increased separation of concerns, enhanced application portability, lesser requirements for multithreading at the application level and potentially simplified application synchronisation. However, applications based on the proactive model are hard to debug. Moreover, they may also have a potential lack of control over the scheduling and policies to control outstanding operations [6].

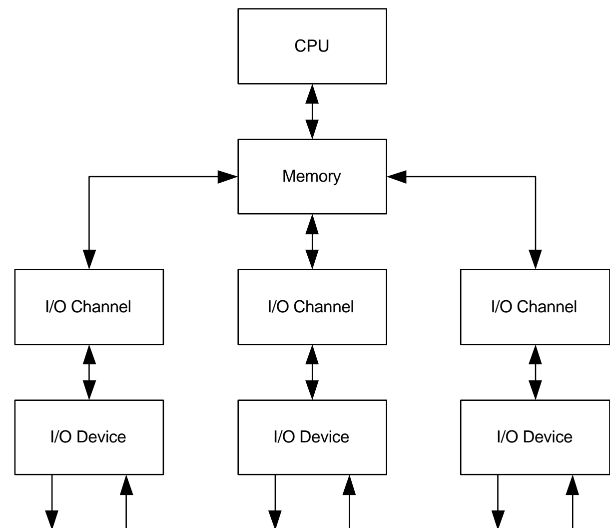

Figure 1. Data flow representation of I/O Channels

3. ACTIVE TRANSCEIVER DESIGN PATTERN FOR COMMUNICATIONS SOFTWARE

3.1 Intent

Active Transceiver design pattern allows the development of distributed application components in a multithreaded environment where the processing and communication subsystems are decoupled and execute as separate active objects. These subsystems within a distributed application component communicate with each other through shared queues.

3.2 Motivation

Active Transceiver design pattern is based on a structure similar to most hardware network devices. For example, large computer systems often employ special computers (referred to as channels) that relieve the main CPU of byte level I/O responsibilities. These channels operate asynchronously and concurrently with other channels and the CPU. CPU performs the data processing tasks whereas communications is delegated to the channels. Data to be communicated to another CPU is written to memory shared between the local CPU and the channel. Data received by the channel is written to the shared memory and is subsequently read by the CPU. This shared memory acts as buffers between the CPU and its associated channels. A CPU may also have several channels, each connecting to a different I/O device [7]. A simplified data flow representation is shown in figure - 1.

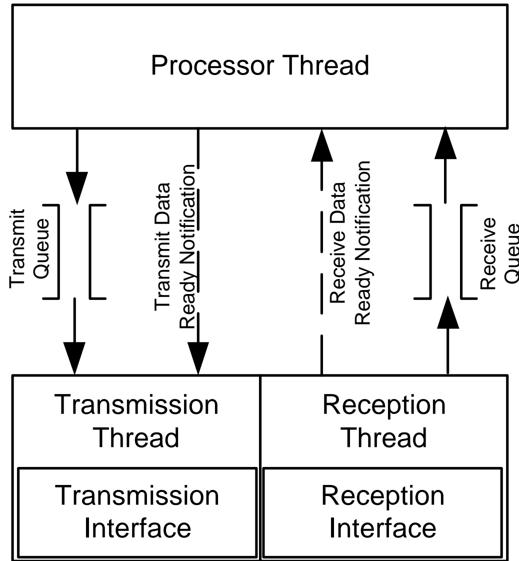

Figure 2. Software Architecture of a Data Communications Application

A data flow software architectural structure of a data communication application based on such an abstraction is shown in figure - 2. The communication subsystem is further subdivided into a Transmission Thread and a Reception Thread. These communicate with the Processor subsystem (executing in a separate thread) via Transmit Queue and Receive Queue respectively. Notification of inserting a message (for transmission or after reception) in the respective queue by the Processor subsystem or the Reception Thread is through condition variables. The Processor subsystem does not block upon writing a message to the Transmission Queue. Depending upon the application logic, the Processor may or may not block for more received messages if the Reception Queue is empty. Transmission and Reception Interfaces to the platform's communication mechanism provide the communication facilities. Additionally, a layer of abstraction can be inserted over the Transmission and Reception Interfaces that allow marshalling of objects by value. This allows the peer Processor subsystems to communicate with each other via objects that are serialised before transmission and deserialised upon reception.

3.3 Applicability

Active Transceiver design pattern is applicable when the application logic needs to be completely decoupled from the underlying communication mechanisms. Active Transceiver design pattern is suited for implementation on platforms that support concurrency.

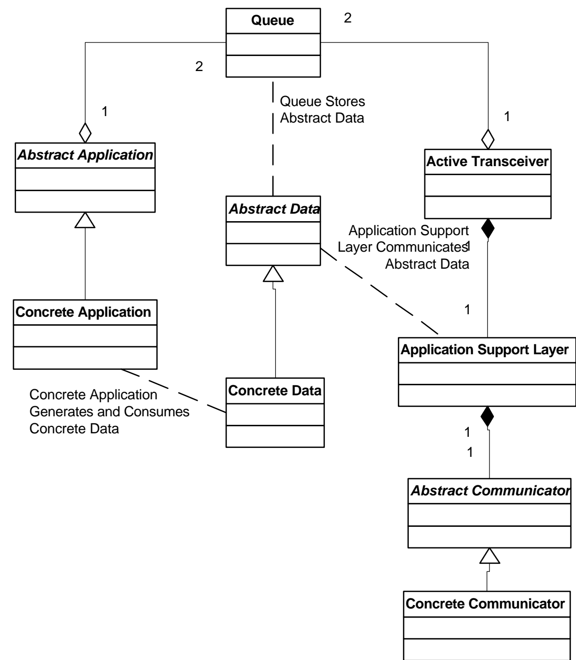

Figure 3. Structure of the Active Transceiver Design Pattern.

.

3.4 Structure and Participants

Figure – 3 shows the structure of the Active Transceiver design pattern. At the highest level, this design pattern shows interaction between the Application (also referred to as the Processor) and the Active Transceiver via two Queues. These Queues communicate data objects between the Application and Active Transceiver. Classes of data objects that need to be communicated are derived from an Abstract Data class. Abstract Data class includes methods to serialize and deserialise simple data types. The derived classes include functions to serialize and deserialise their complete states

Active Transceiver uses an Application Support Layer. Same instance of the Application Support Layer is shared between both the (transmit and receive) threads of the Active Transceiver. Application Support Layer manages the serialization and deserialisation of objects being communicated to the peer Application.

Byte array obtained by serializing a data object is transported over the network using an instance of a subclass of the Abstract Communicator class. Subclasses of the Abstract Communicator base class achieve communication over different communication technologies and protocols, e.g., socket interface or a proprietary protocol over a serial data link. Application Support Layer is only provided an abstract interface for transmitting and receiving byte arrays and the actual technology being used to communicate these is masked from it.

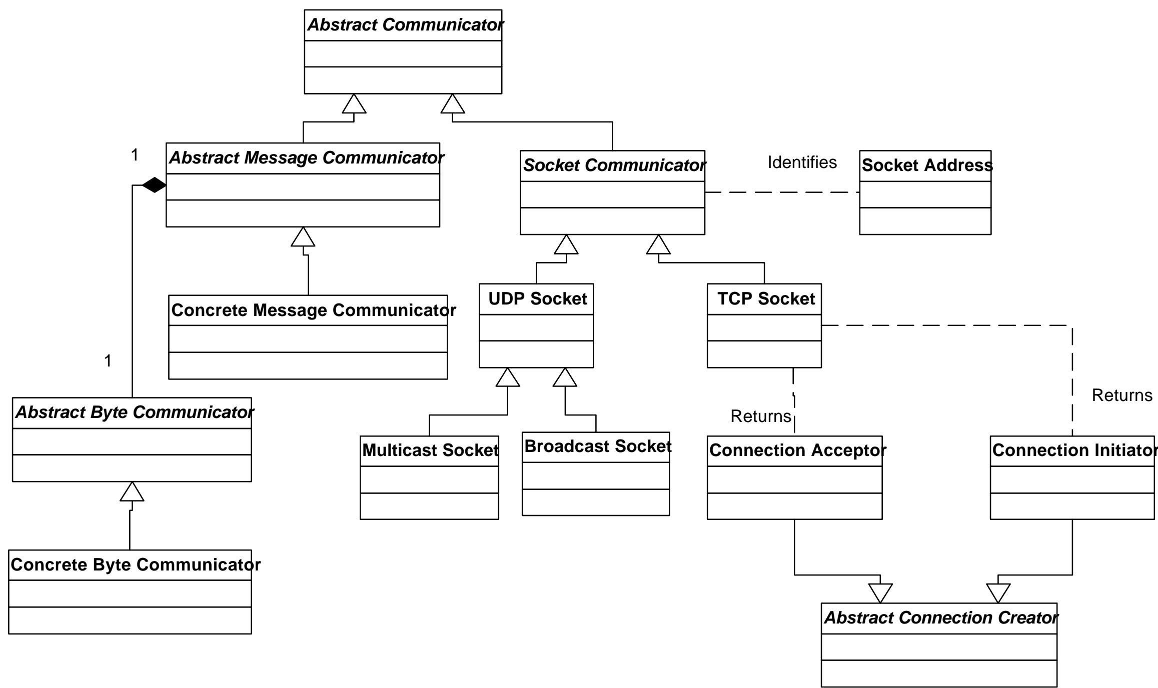

Figure 4: Abstract Communicator and its subclasses

Upon receiving a byte array from an instance of a subclass of Abstract Communicator, the Application Support Layer deserialises the data object whose state is represented in the received byte array. This object is type casted as an object of Abstract Data class and inserted in the Receive Queue. Therefore, the Application Support Layer should have a mechanism to determine the Concrete Data object to be deserialised. In the simplest case when the types of Concrete Data objects are fixed, this information can be embedded within the Application Support Layer. Alternatively, the Application Support Layer may use an object factory to deserialise objects from the received byte array.

Figure 4 shows a further elaboration of class hierarchy of the Abstract Communicator and its subclasses. Such class hierarchies exist in literature, e.g., the ACE (Adaptive Communications Environment) that provides object oriented network programming components encapsulating UNIX and Windows NT network programming interfaces [8]. Components of ACE attempt to alleviate problems of conventional low level, non-portable and non-type safe network programming interfaces.

Abstract Message Communicator inherits from the Abstract Communicator. Subclasses of the Abstract Message Communicator use the subclasses of the Abstract Byte Communicator to implement a high level interface over a variety of data link layer protocols. This class hierarchy is, therefore, useful in building gateways where one or more interfaces use proprietary communication protocols.

For communication using the Internet Protocol suite, an abstract class Socket Communicator inherits from the Abstract Communicator. An instance of a child class of the Socket Communicator represents a socket, which is identified on the network by an instance of the Socket Address class. TCP Socket derives from the Socket Communicator and encapsulates mechanisms for stream communication. Similarly, UDP Socket derives from the Socket Communicator and provides facilities for datagram communication. UDP Socket is further extended to provide facilities for multicast and broadcast communication via Multicast Socket and Broadcast Socket classes.

Instances of the class TCP Socket are created either by an instance of the Connection Acceptor class or by an instance of the Connection Initiator class. An object of the Connection Acceptor class waits for a connection request from a client application. Once it accepts a connection request it returns to the application the reference of an object of the TCP Socket class to communicate data with the connected client. Objects of the Connection Initiator class are client side components that initiate connection requests. Connection Initiator returns the reference of an object of TCP Socket class to communicate data with the connected server. In this regard, these are similar to the Connector and Acceptor patterns that Schmidt and Schmidt et al describe in [5], [3] and [4], however, here these are represented as a Factory Method pattern describe by Gamma et al [2].

Variations of the Bridge pattern [2] are also visible in figures 3 and 4. Considering figure 3, Application Support Layer provides an abstract notion of communicating data objects between peer application components whereas actual communication is performed by instances of subclasses of the Abstract Communicator. Application Support Layer can be assigned an instance of any of the Concrete Communicator classes at runtime. However, extensibility

in the Application Support Layer has not been envisaged yet.

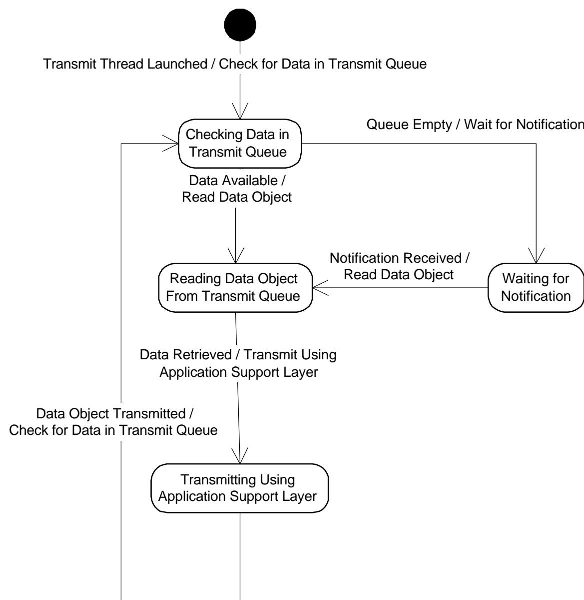

Figure – 5: Dynamics of transmit thread of Active Transceiver

Considering figure 4, Abstract Message Communicator, Concrete Message Communicator, Abstract Byte Communicator and Concrete Byte Communicator form a Bridge pattern. Concrete Byte Communicator could be communicating bytes over a serial synchronous link, serial asynchronous link or even a parallel data link. Communication of bytes within a message is decoupled from message communication tasks such as frame formatting, frame delineation, error detection, addressing etc. and the two can vary independently.

3.5 System Dynamics and Collaborations

The application, i.e., an instance of Concrete Application class collaborates with two instances of the Queue class, i.e., the Transmit Queue and the Receive Queue. While receiving data, the application checks for received data in the Receive Queue. If data are available, the application can retrieve these from the Receive Queue. If no received data are available in the Receive Queue, then the application can wait for a notification from the Active Transceiver that data have been received and inserted in the Receive Queue. Alternatively, the application can proceed to perform other application related tasks.

Operations of the Active Transceiver are better defined. The transmit thread of the Active Transceiver checks if the application has inserted data in the Transmit Queue. If data exist in the Transmit Queue, transmit thread reads these data objects and passes these to Application Support Layer for transmission to the peer Application Support Layer. If no data exist in the Transmit Queue, the transmit thread waits for a notification from the application that it has written a data object in the Transmit Queue. Dynamics of the transmit thread are graphically specified in the state transition diagram shown in figure 5.

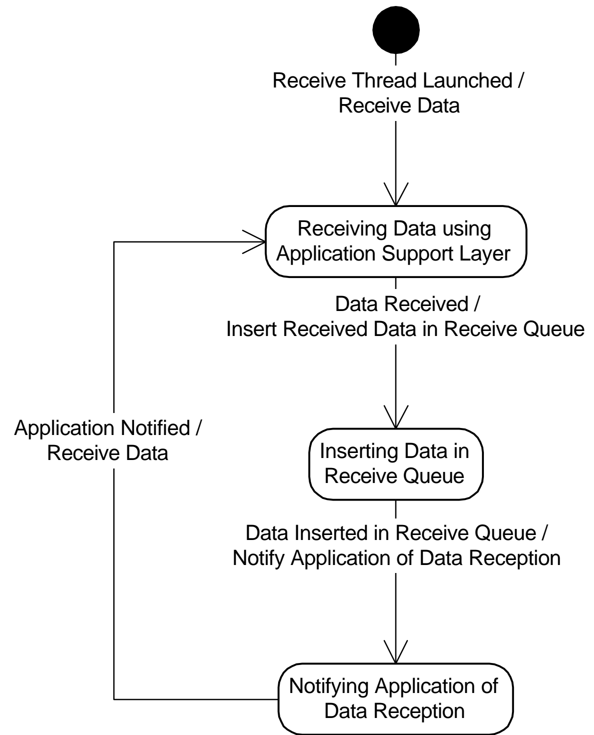

Figure – 6: Dynamics of receive thread of Active Transceiver

Receive thread of the Active Transceiver calls the receive method of the Application Support Layer. This is a blocking call that returns only when data have been received on the underlying communication mechanism and deserialised by the Application Support Layer.

Application Support Layer returns deserialised objects type casted as objects of the Abstract Data class. Receive thread of the Active Transceiver then writes these deserialised objects into the Receive Queue and raises a notification for the application that data have been received and inserted in the Receive Queue. Dynamics of the receive thread are graphically specified in the state transition diagram shown in figure 6.

3.6 Consequences

The main advantage obtained in using the Active Transceiver design pattern is the decoupling of the application from the communication mechanism. This design pattern exhibits a loose coupling of cohesive components. Extensibility of the class hierarchy encapsulating the underlying diverse communication mechanisms also allows the use of proprietary link layer protocols. Blocking calls in transmit and receive threads

ensure optimal CPU resource allocation through pre-emption of blocked thread. Implementation of an Active Transceiver based applications on multiprocessor platforms can further enhance the performance of the system as application and communication components can execute on separate processors on the platform. Furthermore, peer application components communicate application layer data objects rather than byte arrays. Communication at this level of abstraction allows the developers of application components to focus on peer interaction at the application layer.

Implementations of Active Transceiver based applications on uniprocessor platforms may experience overheads related to context switching. These are generally more obvious when the application has several threads executing processing intensive tasks.

Application Support Layer in the Active Transceiver requires the knowledge of application data types for deserialisation. As mentioned earlier, this knowledge could be implemented within the Application Support Layer. Alternatively, it can be encoded in a factory object that is assigned to the Application Support Layer. It is also possible, for reception, to communicate the received data as a general object encapsulating the received byte array to the application layer where a suitable factory object deserialises the received object.

Finally, the Active Transceiver in its current form focuses on efficient and reliable data communication. Tasks such as creation of appropriate Concrete Communicator objects, connection establishment etc. are performed by the application initiation code and the instantiated objects are passed to relevant instances of the Active Transceiver.

4. CONCLUSIONS

Active Transceiver design pattern focuses on decoupling application and communication components of distributed systems. This decoupling allows the two to vary independently without significant implications of variation of one on the other. This design pattern also includes an extensible class hierarchy that allows applications to utilize a variety of standardized and proprietary communication protocols.

However, in its current level of specification, Active Transceiver focuses on efficient and reliable data communications with the tasks related to management of communication channels and links performed by the application initialization procedures. This makes Active Transceiver more suitable for applications where the nodes in a distributed system remain largely static. Thus, upon initialization, these nodes establish the required connections or initialize appropriate communication channels, which are then subsequently used throughout the execution of the application.

Work is currently in progress on the specification of Managed Active Transceiver where instances of Active Transceiver could be created and managed as required by the application during its execution. The challenge, however, is to achieve this objective with lowest possible coupling between the application and communication components.

REFERENCES

[1] Russell Corfman, "An Overview of Patterns". In *The Patterns Handbook: Techniques, Strategies and Applications*, edited by Linda Rising, pages 19 - 29, Cambridge University Press, 1998.

[2] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Pearson Education Inc. 1995.

[3] Douglas Schmidt and Paul Stephenson, "Using Design Patterns to Evolve System Software from UNIX to Windows NT". In *The Patterns Handbook: Techniques, Strategies and Applications*, edited by Linda Rising, pages 471 - 504, Cambridge University Press, 1998.

[4] Douglas Schmidt and Paul Stephenson, "Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms", In *The Proceedings of the 9th European Conference on Object Oriented Programming*, Aarhus, Denmark, August 7-11, 1995.

[5] Douglas Schmidt, A Family of Design Patterns for Application Level Gateways. *Theory and Practice of Object Systems*, 2(1), December 1996.

[6] Irfan Pyarali, Tim Harrison, Douglas Schmidt and Thomas Jordan, "Proactor: An Object Behavioural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events". In *The Proceedings of the 4th Annual Pattern Languages of Programming Conference,* Allerton Park, Illinois, September 1997.

[7] Thom Luce, *Computer Hardware, System Software and Architecture*, McGraw Hill Inc. 1989.

[8] Douglas Schmidt, Tim Harrison and Ehab Al-Shaer, “Object Oriented Components for High Speed Network Programming”, In the *Proceedings of USENIX Conference on Object Oriented Technologies*, Monterey, CA, June 1995.