A Component Framework for Communication in Distributed Applications

Jeffrey M. Fischer *jeff_fischer@acm.org*
Milos D. Ercegovac *milos@cs.ucla.edu*
UCLA Computer Science Department

Abstract

The development of communications services for distributed applications that are both well-structured (layered) and efficient can be difficult. This paper presents a C++ framework which promotes the development of reusable components that can be combined to implement communications services. These flexible services can be built without incurring an unacceptable performance cost. The basic abstractions in the framework are presented, followed by a discussion of more specific services for unstructured binary messages. Then, the performance impact of the framework is analyzed using a simple test application. The framework is shown to have a minimal overhead that is independent of message size.

1 Introduction

The trade-offs between layering and efficiency can be very difficult when designing communication services that must support multiple network interfaces or different client applications. For example, different network programming interfaces may provide different reliability and flow control capabilities. As a result, an application must provide compensation code to ensure that the correct application behavior is achieved for all network interfaces. Managing this compensation code can have a significant cost in efficiency. For instance, extra copies of message packets are frequently made to isolate the memory management requirements of network protocol layers. On the other hand, if an application is optimized to a specific communication interface, the application code is likely to be structured using assumptions about the underlying communication model.

This paper presents a *component framework* to separate an application from the implementation of its communication mechanism and also to isolate the individual layers within the communication code from each other. We are using the term *framework* as defined in [1] – "*A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations.*" We are using the term *component* to mean a unit of software that: implements a well-defined interface, has no external state dependencies, and can be dynamically inserted into a framework.

The primary abstraction provided by the framework is called a "protocol component". A protocol component implements one layer of a communication protocol. It has a "top" to which new outgoing messages are submitted and a "bottom" at which incoming messages arrive. Each component takes messages, performs transformations on them, and then passes the messages onto the next component. Protocol components are connected together in an acyclic graph, a more general version of the standard protocol stack. A "protocol graph" object manages this graph and routes messages through the graph.

Figure 1 shows the relationships between the three basic abstractions in the framework.

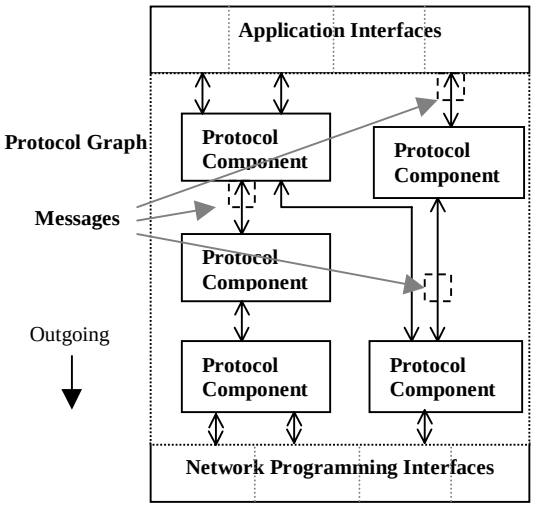

Figure 1. Abstractions in the component framework

2 The Basic Abstractions

This section looks in detail at the basic abstractions in the protocol component framework. The UML class diagram in Figure 2 depicts the C++ classes which implement these abstactions: *dat_message*, *dat_proto_component*, and *dat_protocol_graph*. Each box represents a class, with the name of the class in the top compartment of the box and the operations (methods) of the class in the bottom compartment. Methods with a "#" to the left of their name are protected methods (only accessible to "friend" classes). Methods shown without a "#" are public. If a class has virtual methods and serves primarily to define an interface, this is noted with the string "<<Interface>>" above the class name. The arrows indicate a unidirectional association between classes.

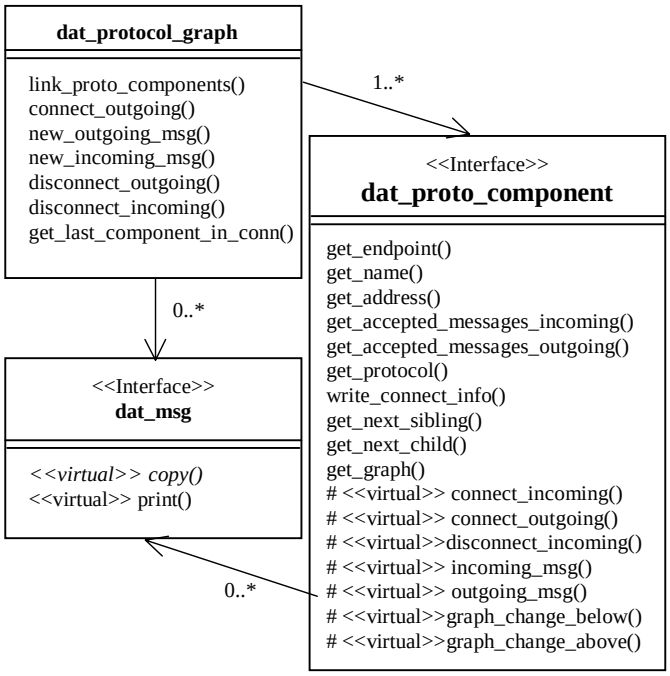

Figure 2. A UML Class Diagram of the Basic Abstractions

The class *dat_msg* provides a single base class for all messages in the protocol component framework. This allows a protocol component to potentially handle any message. Two pure virtual methods are defined by *dat_msg* and must be implemented by all subclasses. The *copy()* method returns a copy of the message. The *print()* method prints the contents of the message to the specified output stream.

The memory management protocol for messages is simple. On the sending side, the protocol component that initiates the send allocates the message. The protocol component that interacts with the Operating System's network is responsible for de-allocating the message, once it is safe to do so. On the receiving side, the message is allocated by the protocol component that reads the message from the network interface. It is de-allocated by the protocol component that actually consumes the body of the message.

A protocol component is a layer of a network protocol stack that may have zero or more components below it and zero or more components above it. Outgoing messages (subclasses of *dat_msg*) enter a component from above, are processed, and then are passed to one or more components below the component. Incoming messages enter the component from below and are passed to one or more components above. The base class for protocol components, *dat_proto_component*, stores the links to a component's neighbors (and provides accessor functions to traverse these links). In addition, it specifies virtual methods for establishing connections, passing messages, and providing notification if the protocol component graph changes. These virtual methods are protected – they can only be called by the protocol graph class (which is defined as a "friend" class to *dat_proto_component*).

The protocol graph class, *dat_protocol_graph*, acts as a container for protocol components. Unlike *dat_msg* and *dat_proto_component*, it provides a complete implementation and does not function as the base class for other classes. The protocol graph class provides methods for linking protocol components within its graph, for creating and destroying connections, and for initializing the send of a message.

The class hierarchy of message types encourages the development of interchangeable protocol components. A component can be designed independently of the components it will be connected to and without full knowledge of message contents. Some examples of services that could be implemented in this fashion include flow control, reliable message delivery, and connection multiplexing.

2.1 Identification and Routing

Application adapter components are globally identified by a structure known as an endpoint (type *dat_endpoint*). An endpoint consists of two numbers: a node id and a component id. Endpoints are generated by the protocol component framework and are guaranteed to be unique over the set of machines that may communicate using the framework.

To specify a message destination, one provides the endpoint of the desired protocol component and a

"*protocol string*". A protocol string is a flat representation of the protocol graph below the destination component. The protocol graph object of the sending process uses this to route a message through its graph to an adapter component at the bottom of the graph. This component then sends the message to its peer in the destination process. The destination process's protocol graph can then route the message up to the destination component using the provided endpoint. The fact that message routing is performed by the protocol graph and not by individual components is an important feature of this framework. Routing algorithms only need to be developed once and protocol components do not need to be aware of them.

2.2 Protocol String Format

The protocol string contains a list of the protocol components in the graph below and including the current component. The components are ordered according to a depth-first search. Each component is identified by its name followed by its connect address (if one exists). To distinguish the components of the list, the following special characters are defined:

```
::  => list component separator
 :  => separates name of an component
       from its address
|   => indicates two parallel paths in
       the graph
()  => parenthesis are used to show a
       grouping of components
\   => indicates that the next
       component is not a special
       character, but instead should be
       treated as a part of the current
       list component
```

Figure 3 shows an example protocol graph.

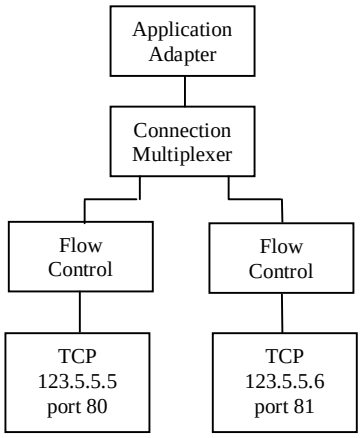

Figure 3. A Protocol Graph

This example graph would be represented as:

```
Application Adapter::Multiplexer::
   (Flow Control::TCP:123.5.5.5,80) |
   (FlowControl::TCP:123.5.5.6,81)
```

2.3 Connection-Oriented vs. Connectionless Messages

Protocol components support both connection-oriented and connectionless communication. When a connection is established, the protocol graph finds a route to the specified destination through the graph. The connect method of each protocol component along the route is called, providing the source endpoint of the connection, the destination endpoint of the connection, and a connection id. Each component can then initialize any state needed for the connection. The protocol graph maintains a table of connections. For each connection, the graph object stores the connection's path through the graph. When a message is sent, the protocol graph can follow this pre-calculated path. When a connection is broken, the protocol graph again traces the route through the graph, calling the disconnect method of each component. The connection-oriented approach amortizes the cost of finding a route through the graph over many message sends. It also enables components to maintain state about the ongoing message dialog between two endpoints.

For connectionless messages, the route through the graph is built for each individual message. The connection methods of components are not called and no information about the route is kept by the protocol graph. As a result, the overhead of finding the correct route through the graph must be paid for each message.

3 Datagram Services

The Datagram Services[1] are a specialization of the protocol component abstractions. These services are designed for the efficient processing of flat, binary messages. Efficiency is achieved by avoiding unnecessary copying and the fragmentation/re-assembly of messages. The datagram services provide one concrete subclass of *dat_msg – dat_datagram*, which is used to send messages through datagram protocol components. The protocol

[1] The name *datagram* here does not necessarily imply the connectionless, unreliable communication of a traditional datagram service [5]. These datagram messages are similar in the sense that that they have well defined message boundaries, specify their destination explicitly, and contain unstructured binary data. However, the quality of service is determined by the choice of protocol components over which these messages traverse.

components inherit from a single base class - *dat_datagram_component*. This base class provides datagram-specific methods not present in the generic component class (*dat_proto_component*).

A datagram protocol component can be written without embedding any assumptions about the components it will be connected to. This permits the development of modular applications and protocol components. Two generic protocol components (the Flow Control Component and the TCP/IP Adapter) will be described that demonstrate this modularity.

Figure 4 is a UML class diagram showing the C++ classes associated with the datagram services, along with their relationships to the framework's basic abstractions. The arrows with the large heads represent a class inheritance relationship. Only methods which are unique to the datagram subclasses are shown in this picture (the methods on the base classes are not shown).

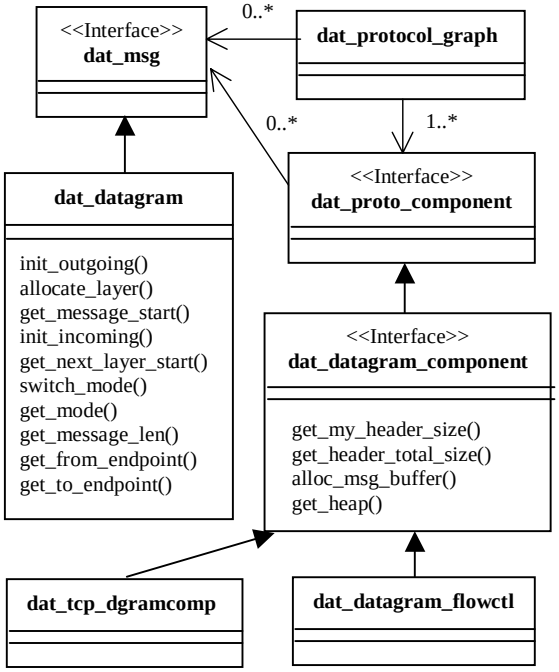

Figure 4. UML Class Diagram for Datagram Classes

3.1 The dat_datagram and dat_datagram_component Classes

A datagram message consists of an application-defined body and one or more headers. The headers are used by protocol components in a sending process to pass information to their peers in the receiving process. An application is not aware of the contents or number of these headers. In addition, each protocol component is only aware of its own header and not any others.

An application creating a *dat_datagram* message specifies a message size and destination. The message object provides an area of memory for the application to place the data it wants transferred. The total amount of memory allocated is actually the amount needed by the application plus any needed by protocol components for header information. After the application has filled its data, the message is then routed through the protocol component graph toward its destination. Protocol components that have reserved space in the message can add their header data as the message passes through them. At the receiving process, protocol components along the path to the destination can read the header data from their peers in the sending process. When the message reaches its destination, the body of the message is then retrieved.

The *dat_datagram_component* class provides additional methods to assist in the allocation of datagram message layers. The protected methods defined in the protocol component base class remain pure virtual and must be defined by the subclasses of *dat_datgram_component*.

3.2 Message Sizing and Buffer Allocation

The size and number of headers for a message sent from a specific source protocol component can vary, depending on the destination. When a message is being allocated, the exact destination and path through the component graph is not necessarily known. For performance reasons, it is often important to allocate the memory for all layers in on contiguous region. To solve this problem, each datagram protocol component provides the largest header size it will need through the method *get_my_header_size()*. The maximum size for a message's headers given its source component, can be calculated in advance by finding the largest cost path through the graph below it. The total allocation size for a message is the sum of the maximum header size and the size of the message's body. This may result in a larger message buffer than necessary for a given message, but it has no performance impact, since only the actual message needs to be sent over the network.

Some network interfaces may have special memory management requirements. For example, a shared-memory or memory-mapped interface may require that memory be allocated from a specific region or be pinned in physical memory by the OS. Other interfaces may require that a memory buffer be held until an acknowledgement from the message's destination is received. To support these

requirements, each datagram protocol component can have a heap associated with it. When a message buffer is allocated (using the *alloc_msg_buffer()* method of *dat_datagram_component*), one of three heaps will be chosen (in order of preference): the network adapter's heap, the originating component's heap, or the default heap. The heap of a buffer is stored in the datagram message so that the buffer can be returned to the correct heap when it is no longer needed.

4 Example Datagram Components

Two example components are presented that make use of the datagram services – a flow control component and a TCP/IP adapter. These components were implemented without any assumptions about their neighboring components in the protocol graph – a key goal of the protocol component framework.

4.1 The Flow Control Component

The Flow Control component limits the number of unacknowledged messages sent to each destination in order to prevent deadlocks and network congestion.

When a Flow Control component is created, the size of the network adapter's receive buffer and maximum number of connections must be specified. The receive buffer space is divided evenly among the connections. When two processes that have Flow Control components connect, they exchange buffer space allocations. Each process agrees to not send more data than the receiver's allocation without receiving an acknowledgement. If all the processes communicating with a given process obey this protocol, that process will never run out of receive buffer space.

To implement this protocol, the Flow Control component maintains for each remote connection a count of available bytes for sending, unacknowledged received bytes, and a message queue. Received messages are acknowledged by passing byte counts in a flow control header in each outgoing message.

The Flow Control component can also fragment/reassemble messages that are larger than the portion of the destination's receive buffer allocated to the sender. Such messages are broken into fragments smaller than the send-bytes allocation and then each fragment is sent or queued individually. On the receiving side, a queue of incoming message fragments is maintained for each connection. Each incoming message fragment is placed on the queue. When the final fragment of a message is received, all the fragments are dequeued and the message is re-assembled. Messages that are smaller than the receive-buffer allocation are passed over the network without undergoing fragmentation and skip the queuing/re-assembly on receiving side.

4.2 The TCP/IP Adapter Component

The TCP/IP adapter creates a TCP connection for each protocol component connection and sends datagram messages over those connections.

When the *connect_outgoing()* method of the component is called, the connection address is parsed to obtain a TCP/IP address and port. Then, the *connect()* function of the BSD socket interface is called with this address. Once the connect call has completed, a short message is sent to the destination specifying the source and destination endpoints. Finally, an entry is created in a table that maps between connection ids and file descriptor numbers.

The component listens for incoming connections on a file descriptor. If another process is attempting to connect, the adapter will get a read event on that descriptor. It accepts the connection, reads the connect message from the new socket, and calls the *connect_incoming()* method of the protocol graph. The protocol graph creates a connection in the local graph up to the destination protocol component and then returns an id for this connection. The TCP/IP adapter stores this id along with the new file descriptor in the connection id to file descriptor mapping table.

The protocol graph passes an outgoing message to the TCP/IP adapter by calling its *outgoing_msg()* method. The adapter looks up the file descriptor associated with the connection id and writes the message to the file descriptor (using *write()* on Unix and *send()* on NT).

When the adapter detects a read event on one of its socket connections, it reads in the global header for the incoming message. From this header, the adapter learns the size of the message. It then allocates a message buffer of the specified size, copies the header to the buffer, and reads the rest of the message into the buffer. The connection id associated with the file descriptor is retrieved and then the message is passed to the protocol graph by calling its *new_incoming_msg()* method. If the adapter detects a remote disconnect event on a socket connection, it calls the disconnect method of the protocol graph, which in term notifies all of the components along the connection.

5 Performance Measurements

In evaluating the utility of the protocol component framework, one must consider the performance penalty of the infrastructure versus a more direct (but less general) communication interface designed for a specific protocol. To measure this overhead, we will use a simple "echo" program built using the TCP/IP Adapter and Flow Control components. Two instances of the echo program are started – one acts as a message initiator and the other as a responder. The initiator creates and sends a fixed length message to the responder. Then initiator then waits for incoming messages. The responder starts out in receive mode. When it get the message from the initiator, it immediately sends the same message back to the initiator. This process is repeated for 1000 message round trips. The total execution time can be divided by 2000 to get the average message latency for a given message size.

A similar program has been written directly on top of TCP/IP, without the protocol component framework. Comparing the performance of the two programs gives us an idea of the framework's latency overhead. Table 1 and Figure 5 show the results of comparing both programs on two PCs running Windows NT. We see that the protocol component framework adds a non-negligible, but not excessive overhead to the message latency. This overhead is roughly constant and thus decreases as a percentage of total latency as the message size is increased. In addition, we learn that the Flow Control component contributes only a small portion of the overhead. Of course, the overhead of using the protocol component framework can vary, depending on CPU speed and the latency of the underlying network.

Table 1. Latency (milliseconds) vs. Message Size.

Msg Size (Bytes)	Average Msg Latency for "Raw TCP/IP"	Average Msg Latency for Protocol Comp	Average Msg Latency for Protocol Comp without Flow Control	Percent Overhd Protocol Comp vs. Raw TCP/IP	Percent Overhd Protocol Comp (no Flow Control) vs. Raw TCP/IP
1	0.130	0. 170	0.168	30.1%	29.2%
64	0.138	0.185	0.175	34.0%	26.8%
256	0.178	0.225	0.213	26.4%	19.7%
1024	0.325	0.370	0.360	13.8%	10.8%
4096	0.703	0.746	0.741	6.1%	5.4%

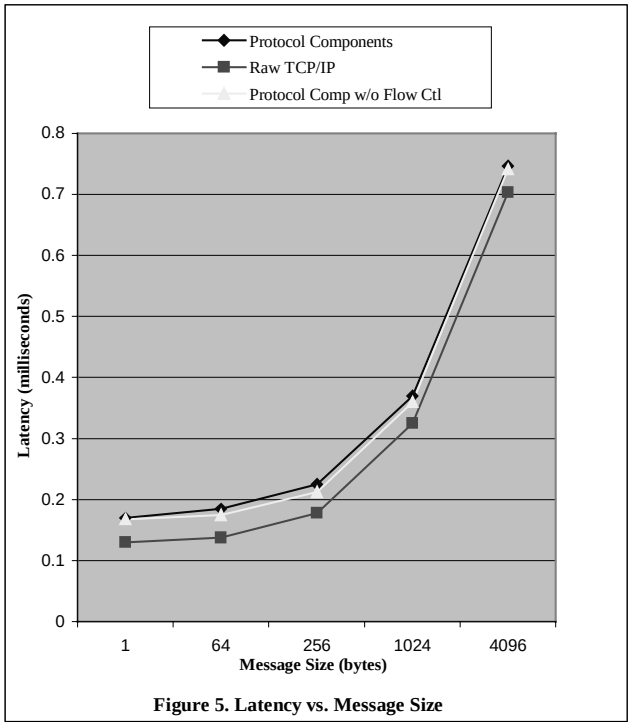

Figure 5. Latency vs. Message Size

6 Conclusions and Related Work

This paper has presented a framework that enables the development of well-structured and efficient communications services. The layers of a communication protocol stack are implemented as interchangeable protocol components. These components can be connected in an acyclic graph – thus allowing the simultaneous support of multiple protocols. A single container class, the protocol graph, handles the routing of connections and messages through this graph. This frees the protocol components from having to deal with routing and connection establishment issues.

The Datagram Services are a specialization of the protocol component framework for processing flat, binary messages. These services permit a layered design but avoid unnecessary copying and fragmentation / de-fragmentation of messages.

To demonstrate the Datagram Services, a flow control component and a TCP/IP adapter were described. Both components were implemented (and can be used) independent of each other and other specific components.

6.1 Related Work

Network software has long been developed in a layered fashion. See [5] for a good discussion of the OSI seven layer network model. The protocol framework does not implement any particular protocol stack but is a mechanism that supports the organization of layered protocol implementations. It can be viewed as an example

of the Pipes and Filters software architecture [4]. The protocol component class corresponds to the *filter* aspect of the architecture: 1) it transforms a stream of data from an input to an output; 2) it does not know the identity of its neighbors; and 3) protocol components do not share state. The protocol graph corresponds to the *pipes* aspect of the architecture; it manages the flow of data between the filters. The protocol component framework supports the general objectives of the pipes and filters architecture: reusability, changeability / maintainability, and testability.

The x-kernel [3] provides a similar infrastructure for layered network protocols. The three distinguishing features of the x-kernel's infrastructure are all present in the protocol component framework as well: 1) a uniform interface to all protocols, 2) late binding between protocol layers, and 3) light-weight layers (no context switches). There are three important differences between the two infrastructures. First, the protocol component infrastructure performs inter-layer routing for its components, while x-kernel protocol layers must perform their own routing between layers. Second, the x-kernel layers run in the OS kernel, while protocol components run in user-space. Lastly, the protocol component framework provides message memory management support, designed to avoid message copying in user-space. The x-kernel does not address this area.

The x-kernel approach is better suited for the implementation of existing network protocols (like TCP/IP). Leaving the responsibility for routing to each layer allows protocol-specific routing algorithms to be used. The protocol component approach is better suited for the development of new user-mode protocols. In this case, implementing routing in the infrastructure reduces the amount of work required to develop a new component and eliminates routing algorithm dependencies between components.

In [2], another framework for network protocols, called *Conduits+*, is presented. The basic abstraction of this framework, the conduit, is similar to the protocol component. Like the x-kernel, Conduits+ implements routing within the components, rather than in the framework. Routing is performed by the *mux*, a specialized class of conduit. A mux links a single conduit on one of its *sides* (ports) to multiple conduits on its other side. The algorithm for routing a message from the mux input to the correct output is embedded in a separate component called an *Accessor*. This approach better encapsulates routing functionality than the x-kernel but still requires the implementation of a new accessor class for each type of routing that must be performed.

The frameworks also differ in the number of basic abstractions required – Conduits+ provides four different conduit types and a number of support classes. The protocol component framework uses only three basic abstractions (the component, the graph and the message). The tradeoff here is between the complexity of an interface and the amount of predefined structure provided by the framework (which may reduce the total amount of code needed).

6.2 Enhancements and Future Work

There are a number of areas where further investigation and enhancement of the protocol component framework is desirable. The results of the performance experiments show that the overhead of the framework is non-negligible. Additional instrumentation of the code is necessary to further isolate the cause(s) of the overhead. A likely candidate is the number of virtual function calls necessary to move a message through the framework. These calls can be removed by using C++ templates instead of polymorphism to implement the protocol component interfaces. This should result in better performance at a cost in flexibility – components would have to be connected together at compile-time instead of dynamically at run-time. Areas for enhancement of the framework include support for multi-threading, the definition of services to support other message types, and the development of adapter components for other network interfaces.

7 References

[**1**] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995, page 360.

[**2**] Huni, Hermann, R. Johnson, and R. Engel, "A Framework for Network Protocol Software", *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, 1995, pp. 358 – 369.

[**3**] Hutchinson, Norman, Larry Peterson, Mark Abbot, and Sean O'Malley, "RPC in the x-Kernel: Evaluating New Design Techniques", *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989, pp. 99-101.

[**4**] Meunier, Regine, "The Pipes and Filters Architecture" in *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 425-440.

[**5**] Tanenbaum, Andrew S. *Computer Networks, 3rd Edition*. Prentice Hall, 1996.