The Design of the FALCON Framework for Application Level Communication Optimization

Eddie C. Shek, Richard R. Muntz, Larry Fillion
Data Mining Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024
{shek, muntz, larry}@cs.ucla.edu

November 14, 1996

Abstract

There exist a wide-variety of communication-intensive applications which run in networks and platforms of greatly varying characteristics. This implies the need for application level communication optimization, which is the optimization of network communication by exploiting application semantics as well as network and compute node characteristics. In this paper, we propose a flexible object-oriented framework called FALCON for application level communication optimization by allowing complementary network communication optimization techniques to be combined in the form of matching stack layers at the endpoints of a communication channel. Each stack layer is composed of a pair of matching modules, executed in the sender and receiver endpoints respectively. To exploit application knowledge that is only available to one of the communicating peers, the framework allows an executable stack layer module to be supplied by either of the communication peers and provides for safe transport to and execution at the other end of the channel. Automatic rule-based optimization techniques similar to those used for extensible database query optimization are developed to optimize the communication channel stacks based on the characteristics of available stack layers, required properties of the channel, and an application-dependent cost model. In addition to providing architectural support for optimizing network communication, FALCON can also be used to introduce support for new communication and computing paradigms in high-level distributed computing environments. For example, while OMG's CORBA distributed object management architecture adopts remote method invocation as its primary communication mechanism, data streaming and service migration can be easily accommodated within the FALCON framework.

1 Introduction

Large-scale distributed applications such as geoscientific and multimedia information systems are generally characterized by a common need to transfer large data objects in a timely manner for complex data processing and presentation. Nevertheless, the actual requirements for delivery ranges from a need for hard real-time support to best-effort delivery depending on the semantics of the application. In addition, within any distributed application, clients and servers can have vastly different capabilities as a result of the different functions they have to support, as well as the limitations imposed by the platform on which they run. For example, visualization software running on a PDA has vastly inferior capability in terms of both rendering speed and display quality when compared to one running on a graphics workstation. At the same time, a visualization client's behavior may have to change when connected to different multimedia servers supporting different data delivery rate and varying data selection criteria. Similar to the diversity of client and server capability, the characteristics of the network can vary greatly. High-speed networks like ATM, FDDI, and Myrinet with bandwidths over 100Mbps are becoming commonplace to support high-performance networked applications. At the same time, the attractiveness of mobility and ubiquity in computing are fueling an increase in popularity of relatively slow wireless LANs and high-latency satellite-based networks.

Run-time application level knowledge such as application semantics, network configuration, and communication peer characteristics heavily influence the optimal choice of communication-related algorithms in a distributed application. However, low-level network software is oblivious to such knowledge. To address the need to perform application level optimization of network communication in a large number of different applications, we propose a flexible object-oriented communication software framework called *FALCON (Framework for Application Level Communication OptimizationN)*. The central idea of FALCON is to allow communication peers to collaborate and construct communication channels that are customized with matching sender-side and receiver-side stacks in the communication endpoints that exploit run-time application level knowledge to optimize network communication. Before establishing a communication channel, communication peers in FALCON specify the characteristics of application level protocol stack layer implementations they support and an optimizer is used to automatically generate optimal matching stacks in the endpoints of the communication channel based on an application-dependent cost model and quality of service requirements. Note that the bottom-most layer in the stack is always a networking layer that interfaces to an existing network protocol commonly supported by the both the sender and receiver (such as TCP/IP supported by Internet-ready platforms). If multiple network protocols are supported, the most appropriate one is selected by the optimizer when the channel is being established.

In addition to optimization, application semantics can also be used to provide high-level network services such as marshalling that automatically packages complex structured data for transparent transfer between heterogeneous computing platforms. Additional examples of such services include encryption, authentication, and authorization.

Figure 1 shows an example FALCON communication channel. The example protocol stack at the endpoints of the channel contain six well-defined layers, namely FILTER, CACHE, COMPRESS, MARSHALL, FLOW CONTROL, and NETWORK, each performing a task that is suggested by its name.

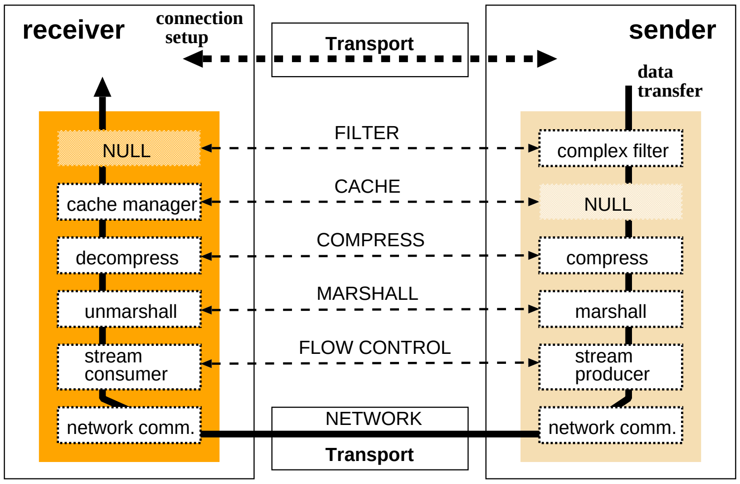

Figure 1: An Example FALCON Communication Channel

In a distributed object computing environment, there are many cases where servers support the same interface but with different implementation and semantics. At the same time, each server may have to serve a wide variety of clients with different quality of service requirements and data usage patterns. Because of the diversity, it is unrealistic to expect network communication between all possible pairs of clients and servers to be optimized staticly, especially as new clients and servers are continuously being introduced. As a result, the implementation of the FALCON communication framework allows each pair of stack layer implementations to be supplied either by the client or by the server and dynamically linked into its communication peer at run-time. A process can supply stack layer implementations that are executed in its communication peer to manipulate the stream of data flowing between them. Implemented in the context of a distributed object management environment, it extends the remote execution semantics of the object model by allowing the migration of object services from the server to be executed in clients to optimize performance.

The rest of this paper is organized as follows. Section 2 further motivates application level optimization of network communication in distributed applications by listing complementary techniques that can be combined to achieve the goal of reducing communication overhead in different computing environments. Section 3 presents the design of the proposed communication optimization framework. We discuss design decisions and tradeoffs in keeping the framework simple yet flexible. We also outline the interface of the main components of the framework in IDL, as well as the major algorithms that govern the operation of the framework. Section 4 describes the prototyping of our proposed framework in the CORBA [11] distributed object computing environment to provide optimized data streaming support. We take advantage of CORBA to simplify the channel configuration process, and use Java as a safe and platform-independent language to allow a sender-side stack layer implementation to be supplied by the receiver (and vice versa). In Section 5, we present related work that has influenced or is relevant to our design. Finally, Section 6 concludes the paper with a brief summary and our future direction.

2 Application Level Communication Optimization

To satisfy the bandwidth and latency requirements presented by different communication-intensive applications, it is often necessary to combine and customize a collection of network communication optimization techniques. In this section, we list a collection of complementary techniques that can be customized to exploit application semantics and network environment to minimize the effect that network communication has on the overall performance of a distributed application. We will also briefly describe how these techniques can be supported in our proposed framework.

2.1 Eliminate Irrelevant Data

One of the main design goals of high-performance network infrastructures and protocols is to maximize the effective bandwidth and throughput of data transfer. Towards this goal, it is often worthwhile to perform sophisticated querying and filtering at the sender to minimize the amount of irrelevant data that are sent to the receiver and hence lower the overall network load.

The main reason why unnecessary data are sent to the client is that the data subset of interest is often not known until run-time (based on user preference). Even if the server supports a "queryable" interface in which the return data satisfies some condition that is specified by the client, the type of filtering that can be done is generally static because of lack of support for dynamic extensibility of the interface or the query language. In this case, our framework allows the client to compose and register executable filters that can be migrated to the server when the communication channel is set up to maximize the scope of filtering that can be supported.

Even if a piece of data served by a server is logically of interest to a client, it may be irrelevant because of the client's inability to present it to the user in time and satisfy the quality of service requirement of the application. In this case, knowing that the client platform has limited capability gives us the possibility of filtering and simplifying data at the sender before they are sent to the client. Similarly, limitations imposed by the network connection (e.g., bandwidth and latency) suggest the use of filtering techniques to reduce the need for bandwidth. The decision of whether filtering should be done, as well as what type of filtering should be done (assuming the availability of multiple filtering strategies) and how much filtering should be done (assuming configurable filtering strategies) can be made by an optimizer which has access to up-to-date information about relevant network configuration and client characteristics.

2.2 Reduce Data Size

In an environment in which network resources are relatively scarce compared to computing resources (such as those in many mobile computing systems or wide-area communications), it is often advantageous for the sender to first compress data to achieve a more compact data representation before transfer. The receiver is then responsible for decompressing the network packets to reconstruct the original data item.

In FALCON, data compression before transfer can be made transparent. A client does not have to know anything about whether data that are delivered across the network are compressed or not if the server supplies both the compression and the corresponding decompression algorithms, and conversely, neither does the server if the client provides the compression modules. If multiple compression algorithms with different characteristics (e.g., compression ratio, cost, lossiness) are available, the optimizer will exploit network and compute node characteristics to determine the best compression algorithm that satisfies application requirements to be registered in the communication channel.

2.3 Amortize Cost of Data Transfer and Retrieval Over Multiple Accesses

Client-side caching is widely used to eliminate the need to retrieve a data item from the server every time it is needed by temporarily saving it in a local cache which will satisfy requests for the data item. At the same time, a cache can also reside on the server side to service multiple client processes. While caching is often performed in the operating system, in either case the behavior and semantics of the application can dictate the optimal configuration of the caching algorithm in terms of the choice of parameters such as cache replacement policy and cache size.

2.4 Minimize Latency

Pipelining and prefetching aim to eliminate the latency of network communication by depositing data on the receiver side in the background of the client application so that data retrieval requests can be readily satisfied locally. They are especially important for interactive distributed applications as well as applications in a wide-area network with high latency.

Data-driven communication allows two communicating processes connected in a pipeline to execute simultaneously. The sender continuously and asynchronously sends data to the receiver and a method at the receiver is awakened when the data arrives to process the data. This allows communication and computation at the consumer and producer to overlap to improve system performance. In this form of data flow, the processes independently manipulate either a sequence of data objects or different portions of a large logical data objects which is manifested in the form of a stream as it flows through the pipe.

However, in many applications such as a parallel query processing engine, guaranteed delivery of stream data is needed. In this case, if the data production rate is much higher than the data consumption rate, the receiver will be flooded with data and may fail. To prevent this from occurring, we need to combine data-driven pipelining with a mechanism that regulates the rate of data flow. For example, the Volcano extensible query processing system [5] proposed a stream buffer control protocol in the context of the exchange operator. A buffer is created in the receiver into which the sender continuously deposits data until it is full. The receiver application on the other hand retrieves data from the buffer rather than from the communication endpoint. In this protocol, the sender can only get ahead of the receiver by an amount limited by the receiver-side stream buffer. Nevertheless, by depositing stream data in the receiver before it is actually needed by a client application eliminates network communication latency.

In real-time multimedia applications such as video conferencing, the sender generates and delivers data streams at a rate that is determined by the characteristics of the encoding scheme, but is independent of the capability of its clients. A client is often only expected to process the data stream as fast as it can; failure of the client to keep pace with data delivery results in data loss and is simply manifested as artifacts in the presentation of the multimedia data to the user instead of a fatal error. To support this kind of streaming operation, the stream producer residing in the sender continuously receives stream data from the server application and asynchronously sends it to the client. The corresponding stream consumer does not maintain a stream buffer but instead only processes the last "frame" in the stream that is available from the network through a lossy protocol such as UDP; the data frames that are not processed are then lost.

2.5 Tailor Network Protocol to Satisfy Communication Requirements

Many networking protocols and communication software packages have been developed. Some present a convenient and high level interface to fit a specific computing paradigm, while others are designed specifically to reduce network communication overhead by trading off different levels of quality of service guarantee, as well as by optimizing for different network infrastructures. For example, MPI is optimized for MPP environments, while UDP is a connectionless protocol particularly suitable for asynchronous and non-guaranteed data delivery.

Distributed applications vary in their needs for communication support; in the case where multiple network communication methods are available, the quality of service guarantee of the chosen network protocol must satisfy application needs. Techniques developed to allow multiple network protocols to be used given a uniform interface such as multimethod communication [4] can be supported transparently in FALCON at the network layer in conjunction with distributed computing environments (especially those with suboptimal networking implementation such as CORBA) for the cases where optimized networking support significantly improves overall system performance.

3 Framework Design

FALCON is a collection of objects that together support the operation of communication channels that optimizes the transfer of large volumes of data through matching communication protocol stacks. FALCON objects have to be customized and instantiated with the appropriate application specific information such as the communication cost model and the performance measures to optimize, as well as the implementations and characteristics of application-specific stack layers. The two main architectural components in the framework are communication channels and the channel manager.

3.1 FALCON Communication Channel

At the core of FALCON are uni-directional communication channels which are logical entities composed of matching channel endpoint objects on the sender and the receiver. A FALCON communication channel optimizes communication by transparently stacking different application level communication optimization layers on top of a network protocol layer in each of the endpoints. Figure 2 shows, in OMT object model notation [10], the object aggregation/composition relationships between objects that constitute communication channels in FALCON.

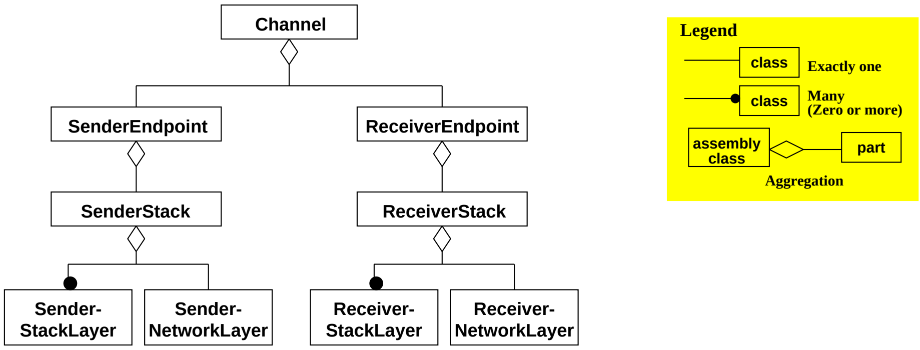

Figure 2: Object Aggregation of a FALCON Communication Channel

A FALCON channel behaves like a connection-oriented network link at initialization time which has an explicit connection phase during which the optimization stacks in the endpoints in a communication channel are configured. Although there is an initial synchronous configuration step to create and set up a communication channel for use, the actual behavior of the channel at run-time (e.g., whether it is synchronous or asynchronous, connection-based or connectionless) depends entirely on the networking protocol chosen.

Similar to UNIX sockets, a process explicitly reads data from the receiver-side endpoint and writes data to the sender-side endpoint after the channel connection has been established (see Figure 3). When data is sent through the sender-side endpoint, the top sender-side stack layer gets called to process the data and in turn recursively passes the data down the stack through a function call (by reference) until the network layer is hit. The network layer actually ships that data across the network. On the other hand, when the receiver calls a receiver method in the receiver-side endpoint to retrieve data, each stack layer recursively calls its producer (the layer immediately below it) until the data is retrieved from the network at the network layer. To reduce latency, a flow control layer module on the receiver side can run in its own thread and prefetch data into its buffer space so that data requests from its consumer can be satisfied immediately. When the data item flows up the receiver-side stack, the stack layers process it and undo the effect the sender-side stack has on the representation of the data. In the case where no non-network stack layers are used in the channel, the client and server interface directly to their network protocols encapsulated by the FALCON network layer interface.

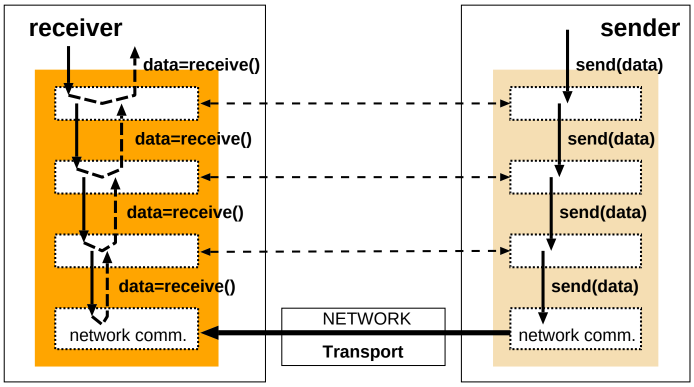

Figure 3: Interaction Between Application, FALCON Communication Channel, and Stack Layers During Data Transfer

3.1.1 Stack and Stack Layers

Each stack layer has 2 matching parts, one for execution in the sender and the other for execution in the receiver. All sender-side stack layers support a simple symmetric interface that looks the same from above and below. Similarly, all receiver-side stack layers support another symmetric interface. This maximizes the flexibility in composing stacks from different stack layer implementations by allowing stack layers to be stacked in arbitrary order as long as they satisfy the constraints imposed on the structure of the channel and the semantics of the stack layers defined by a set of rules registered with the FALCON optimizer as stack layer implementations are introduced. To simplify the implementation of stack layers in FALCON, we choose to define a minimalistic interface for the sender-side and receiver-side stack layers which only support a method to send data to and retrieve data from respectively (in addition to basic methods to initialize and destroy the instance of the stack layer).

One of the most novel features of our framework is the "collaborative" nature of the setup of the communication channel between the communication peers. Specifically, a process can supply both the sender-side and receiver-side implementations of a layer. For example, the data sender can choose to compress data to be sent to a receiver using some content-based compression algorithm; a corresponding decompression routine will have to be registered at the receiver side so that compressed data shipped across the network to the receiver can be decompressed before becoming visible by the application receiver code. On the other hand, the receiver can register a complex, procedurally defined function on the sender-side to filter out irrelevant data not filtered by sender-side queries. As a result, stack layer module objects should be implemented in languages supporting code mobility, such as Java [1] and safe-Tcl [2], so they can be safely migrated from one node to another.

3.2 FALCON Channel Manager

The FALCON channel manager (see Figure 4) controls the channel optimizer which determines the optimal choice of stack layers to be instantiated in the endpoints of a communication channel. In addition, the channel manager maintains system-wide statistics for use by the optimizer, as well as serves as a naming service for all FALCON objects and information required to establish an optimized communication channel.

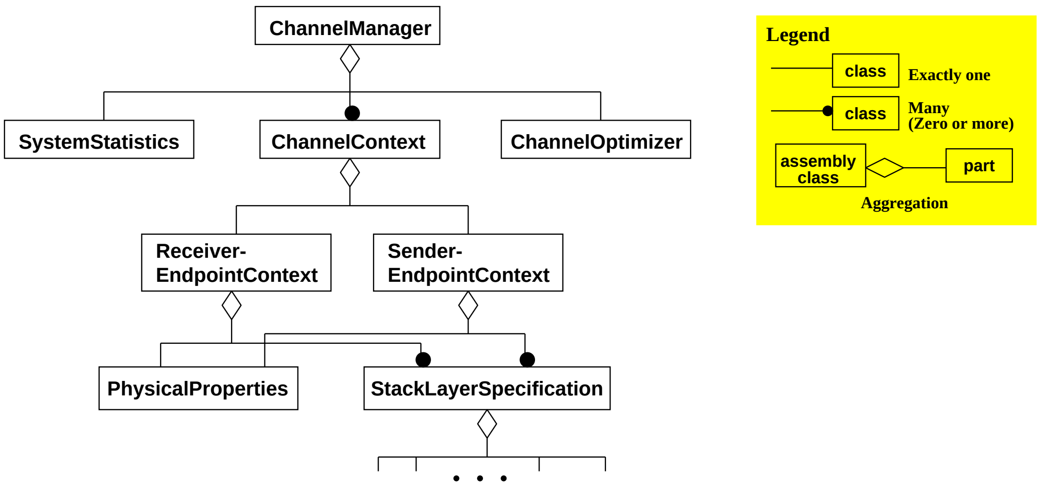

Figure 4: Object Aggregation in the FALCON Channel Manager

Each FALCON communication channel is created to fit a channel context that specifies available stack layers supplied by both the sender and receiver, as well as the expected properties of the channel. A channel context is composed of a sender endpoint context and a receiver endpoint context which contain information supplied by the sender and receiver processes respectively.

The specification of a stack layer includes a reference to the location of the module implementing the layer so that it can be copied from its repository and registered in a FALCON-aware process when the channel is being set up (some well-known FALCON stack layers are compiled in and do not have to be copied at run-time). To allow the optimizer to select the optimal communication channel, the characteristics of each stack layer is specified by functions which estimates its cost and determines the properties that it can deliver respectively. We will describe these functions in detail later in the Section on optimization.

In addition to the stack layer specifications, the sender and receiver specify the expected data delivery and consumption characteristics respectively which together dictates the properties of the channel generated and guide the search for optimal stack layering in the channel. The set of physical properties that can be defined for a communication channel is application-dependent, and the following are some of the parameters that can specify the requirement:

- **Data Schema.** Many stack layers operate only on data of a specific data type. For example, many compression algorithms achieve high compression ratio by exploiting knowledge of data semantics. As a result, the structure of data in a stream is an important property that needs to be captured if structured data were to be supported in FALCON.
- **Reliability of Data Delivery and Data Quality.** Asynchronous network protocols such as UDP often do not guarantee delivery (i.e., are lossy). However, there are often cases where lossy data transfer combined with a synchronous communication protocol is acceptable. For example, given a lossy communication protocol, special multimedia compression and encoding algorithms can be used that allow the receiver to recover from dropped network packets without requiring the sender to retransmit [12].
- **Network Quality of Service.** The quality of service classification scheme of ATM can be adopted to specify bandwidth requirement. In particular, we can allow for the following quality of service specification: CBR (Constant BitRate) vs. VBR (Variable BitRate) vs. ABR (Available BitRate) vs. UBR (Unspecified BitRate). For a communication substrate built on top of networks without support for guaranteed QoS service, a specification of a CBR communication requirement would result in an error since there is no way the requirement can be satisfied.

3.2.1 Flow of Control During Channel Setup

Before a channel can be used for communication, it has to be set up with the appropriate stack layer implementations instantiated at the endpoints. Figure 5 shows the flow of control in establishing a FALCON communication channel for data transfer.

1. **Server Establishes an Endpoint Context.**

   The first step is for the server to establish a endpoint context of communication which contains specification of stack layers it supplies. The endpoint context also maintains either data delivery or data consumption characteristics depending on whether the server is a sender or a receiver in the particular context. After a server endpoint context is properly initialized, it has to be registered at the channel manager to make it available to the channel optimizer and other clients in the distributed environment. This indicates the server's intention to accept connections in the context associated with the endpoint. The semantics of servers awaiting

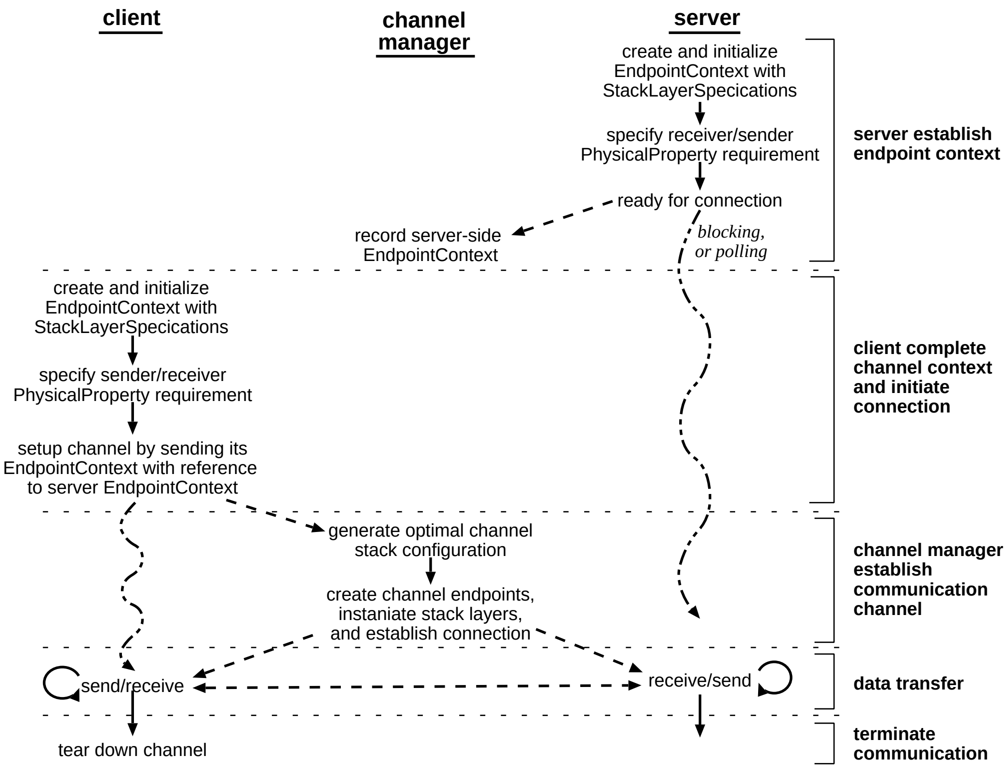

Figure 5: Establishing and Configurating a FALCON Communication Channel

FALCON channel connection is similar to that of servers supporting Unix sockets; a server can be blocked until a connection has been established, or be non-blocking and continuously poll the channel endpoint for connections.

Endpoint contexts are named objects that can persist over the construction of multiple communication channel governed by them. This allows static information about particular communication paths to be reused and helps to reduce the overhead of communication channel set up in FALCON. Nevertheless, they can be modified by their owner as well as explicitly destroyed when they are no longer necessary.

2. **Client Completes Channel Context and Initiates Connection.**

   When a client wants to connect to a server, it initializes an endpoint context that is matched with a well-known server endpoint context to form a channel context. References to the endpoint contexts as submitted to the channel manager as arguments to a request to establish a communication channel that fits the channel context formed by the endpoint contexts.

3. **Channel Manager Establishes Communication Channel.**

   When the channel manager receives the references to the client and server endpoint contexts from the client with a request to setup a channel connection, it feeds them to the channel optimizer which creates client-side and server-side channels, instantiates stack layers in the endpoints (possibly after they are migrated from one platform to another), and finally establishes a direct network connection between the two endpoints.

4. **Data Transfer.**

   At this point, the communication channel is ready for use. The sender creates a sender driver that drives the generation of data that are pushed into the communication channel. On the other hand, the receiver begins to invoke the appropriate method that explicitly consumes data from the communication channel. After communication is completed, the client initiates the tear down of the channel, which disconnects the actual network connection and destroys the stack layer instantiations. The endpoint contexts have to be explicitly destroyed so that they can be reused between channels.

Note that it is not necessary for the client or the server to explicitly register any stack layers. For example, if no stack layer implementations are provided by the client, the protocol stack in the channel will simply be instantiated with stack layer implementation supplied by the server and the well-known ones that are built into the framework by default. Similarly, the data delivery and consumption characteristics of clients and servers do not have to be specified; an application-specific default will be used in the place of a missing specification. This implies that implementors of FALCON-aware clients and servers are not required to have intimate knowledge of the stack layer implementation and channel property specification. If neither the client nor the server supply any endpoint semantics through their endpoint contexts, the communication channel with no quality of service will be constructed with only well-known stack layers built into FALCON. Since channel context essentially captures application semantics for use by the optimizer, the benefits introduced by FALCON become more significant as the amount of information concerning stack layers and channel properties increases.

3.3 FALCON Channel Optimizer

The goal of the FALCON channel optimizer is to generate the optimal communication channel stack layering that satisfies a set of channel properties given a collection of available stack layer modules. Each stack layer has an associated *cost function* that is used to calculate the cost (or overhead) of the stack layer in the context of a stack. The cost model may include both run-time cost and initialization cost, which is primarily dictated by the location of the stack layer implementation since migrating stack layer implementations during channel initialization can be rather costly. Application-specific system statistics and characteristics maintained by the channel manager are implicitly used when the cost function of the stack layers are evaluated. In addition to the requirements of and constraints on communications, the characteristics of the communication peers often also provide hints to optimize communication. In particular, the computing resource (e.g., CPU, RAM, disk space) of a receiver often has a strong bearing on how much processing should be done there. For example, even though compressing a large dataset before sending to a receiver over a slow network can often significantly reduce communication overhead, it may have an adverse effect if the receiver has a very slow CPU and will take a long time to decompress data it receives.

Each stack layer guarantees a well-defined set of physical properties, and an *applicability function* is defined for each to determine if the stack layer can deliver a particular set of properties. In addition, channel properties such as the size of the data stream (both going into the sender-side stack layer module and returning from the receiver-side) and latency are often also carried around for use by the cost function even though they do not specify a requirement. Figure 6 shows the pseudocode of example applicability functions for an ATM-based network layer, a socket-based network layer, and an image compression layer, assuming channel properties consisting only of data schema and network quality of service. Assuming the ATM-based network layer can support any quality of service requirement modeled in the channel properties, its applicability function fails only if it is required to handle structured data. The socket-based network layer is similar to the ATM-based network layer in that they both only support unstructured data. However, socket-based communication does not provide any quality of service guarantee. As a result, the socket-based network layer's applicability function fails if the layer is required to deliver data with stricter quality of service requirement than best-effort delivery. Finally, the image compression layer compresses structured image data into unstructured, flat data before sending. Its applicability function checks if it is required to compress data of non-image types and fails if that is the case. Otherwise, it returns a properties requirement against the stack to sit underneath it in the communication channel which inherits the quality of service requirement against itself and specifies the support for unstructured data created by the compression layer.

3.3.1 Search Algorithm

Stack layers in FALCON support a uniform and symmetric interface so that they can be freely combined. This flexibility introduces an exponential explosion of valid combinations that have to be inspected by the optimizer in order for an optimal stack to be generated. As a result, the FALCON channel optimizer employs an efficient top-down, goal-oriented directed dynamic programming [6] search algorithm that is primarily directed by the need to select a stack layer that can satisfy the required properties, instead of being completely driven by possibilities like dynamic programming based techniques. Figure 7 shows the pseudocode that outlines the FALCON channel optimizer's search algorithm.

Valid communication stacks are generated by applying the applicability function of all stack layers against the expected properties of the communication channel. If a stack layer can satisfy the properties, its applicability function returns the properties the "sub-stack" sitting underneath it has to have in order for it to deliver the expected properties. Assuming that the cost of stack layers satisfy the principle of optimality which states that given a state, an optimal policy for the remaining stages is independent of the policy adopted in the previous stages. In order words, the optimality of a stack generated from a collection of stack layers that satisfy some properties is guaranteed regardless of the stack layers that sit on top. As a result, the process of optimal stack

```
// Applicability function for an ATM-based network layer
ChannelProperties *ATMNetworkLayer::applicability( ChannelProperties *required_props )
{
  // can only send unstructured data with any qos requirement
  if ( required_props.schema == unstructured ) {
    // always bottom-most layer, return a non-failure result
    return required_props;
  } else {
    // cannot deliver required properties
    return NULL;
  }
}

// Applicability function for a socket-based network layer
ChannelProperties *SocketNetworkLayer::applicability( ChannelProperties *required_props )
{
  // can only send unstructured data with best-effort
  if ( required_props.schema == unstructured && required_props.qos == best_effort ) {
    // always bottom-most layer, return a non-failure result
    return required_props;
  } else {
    // cannot deliver required properties
    return NULL;
  }
}

// Applicability function for an image compression layer
ChannelProperties *ImageCompressLayer::applicability( ChannelProperties *required_props )
{
  // compress ``structured''image data into unstructured, flat data
  if ( required_props.schema == image ) {
    // the stack underneath this layer has to handle unstructured data,
    //  and support the same quality of service requirement
    sub_required_props.qos = required_props.qos;
    sub_required_props.schema = unstructured;
    return sub_required_props;
  } else {
    // cannot deliver required properties
    return NULL;
  }
}
```

Figure 6: Example Applicability Functions for an ATM-based Network Layer, a Socket-based Network Layer, and an Image Compression Layer

```
BestStack( req_properties, avail_layers )
{
  // return known best stack that can be generated from a set of
  //  stack layers that satisfy a certain set of properties
  best_stack = KnownBestStack( req_properties, avail_layers );
  if ( best_stack != NULL )
    return best_stack;
  // recursive find the best stack for the required proerties
  best_stack_cost = Infinity;
  for layer in avail_layers {
    // see if stack layer can deliver required properties using
    //  the layer's applicability function
    sub_req_properties = layer.applicability( req_properties );
    if ( sub_req_properties != NULL ) {
      if ( layer->class == "NETWORK" ) {
        // network layer is always at the bottom of the stack
        stack = new Channel();
      } else {
        // find optimal ``sub-stack'' with new required properties
        //  and stack layers not including ones in the same stack
        //  layer class as the ones that has already been used
        sub_avail_layers = avail_layers.RemoveLayerClass( layer.LayerClass() );
        stack = BestStack( sub_req_properties, sub_avail_layers );
      }
      // build stack from layer and optimal sub-stack
      stack.AddLayer( layer );
      stack_cost = stack.Cost();
      if ( stack_cost < best_stack_cost ) {
        best_stack_cost = stack_cost;
        best_stack = stack;
      }
    }
  }
  // save the known best stack that can be generated from a set of
  //  stack layers that satisfy some set of properties for future use
  SaveKnownBestStack( req_properties, avail_layers, best_stack );
  return best_stack;
}
```

Figure 7: FALCON Channel Optimizer's Search Algorithm

building can be repeated recursively for the sub-stack and its properties. Among all stacks that satisfy the channel properties required by the client and server, the one with the lowest cost will be selected.

Each stack layer plays a well-defined role. A representative collection of example layer classes are FILTER, CACHE, COMPRESS, MARSHALL, FLOW CONTROL, and NETWORK shown in Figure 1, each performing a task that is suggested by its name. Multiple stack layers belonging to a layer class may be available for inclusion into a stack when a communication channel is being optimized. For example, multiple network protocols may be supported for actual transfer of data. Similarly, multiple compression algorithms with different compression ratio and overhead tradeoffs may be implemented. However, since stack layers in the same layer class play the same role, we argue that including more than one into the stack will not present any significant cost improvement. As a result, we decided to adopt the heuristic of not including more than one stack layer of the same class in a stack to prune the search tree and reduce optimization time.

4 Example Application: Streaming in CORBA

To verify and demonstrate the benefit of application-level communication optimization, we prototyped FALCON to support data streaming in a CORBA environment. In particular, we chose to prototype in Iona's OrbixWeb which supports the implementation of CORBA clients and servers in Java on both Unix and Windows platforms. Platform-independent distributed computing environments (e.g., DCE, CORBA) provide many valuable services (e.g., naming, location services) and allow applications residing on a wide variety of network and computing platforms to interoperate. The use of these distributed computing environments often introduce significant communication overhead because of their complexity, support for multiple platforms, immature implementation, or a combination of the above. For example, sending unstructured data through CORBA over a high speed network has a high overhead partly because of the unnecessary cost of marshalling (which is required for transferring structured data or between heterogeneous platforms). More importantly, CORBA does not support a stream protocol, and forces a data item, no matter how large it is, to be completely transferred from a distributed object to another before control is returned to the communicating objects. CORBA's lack of a high-performance steaming mechanism [1] hinders its acceptance in domains in which large amount of data needs to be transferred. Examples of such applications include telemedicine, and geoscientific information systems. The goal of supporting streams in CORBA in this prototype is similar to that of blob streaming in CORBA [9]. However, blob streaming aims mainly to push data through a TCP/IP connection as fast as possible by and it does not support application level optimization of communication which we have motivated in earlier sections.

Our prototype allows a data stream to be sent explicitly through a FALCON communication

[1] A RFP for streaming in audio/video applications in the telecommunications domain is out and respond are due in February, 1997.

channel over Java-based socket interface rather than as an argument to CORBA remote method invocation. In the prototype environment, CORBA is mainly used as a convenient location service to allow receivers to transparently locate their senders, as well as a mechanism to facilitate the establishment of a FALCON communication channel. We chose to use Java as the language to implement stack layers to exploit Java's characteristics as a safe, platform-independent, object-oriented, and dynamic general-purpose programming environment. For the same reason, Java is already widely used to develop applets and applications for the Internet and other complex, distributed network. In addition to the network layer, we have implemented simple sender-side filtering and compression stack layers for inclusion into channel endpoints for communication optimization transparent to the application.

To migrate Java-based stack layers, we used SunSoft's Java object serialization technique to serialize stack layer objects implemented in Java into a stream of bytes that are used to rebuild equivalent objects after it is sent to its destination. The serialized representation of an object can be transported to any program supporting the Java class libraries and re-instantiated for execution. In the case where stack layers are sent from the sender to the receiver, the stack layers migration is done in its own thread independent of the data transfer so that they can overlap.

The prototype FALCON channel optimizer implemented the overall framework of the search algorithm. It is augmented with a simple application model consisting of multiple available stack layers (multiple network choices, multiple filters with different selectivity and cost, and marshalling layers), a cost model that captures network bandwidth and computing cost in the stack layers, and channel properties capturing the schema and size of the data stream. The applicability functions of the stack layers are very similar to those outlined earlier in Figure 6.

4.1 Preliminary Performance Results and Lessons Learned

We have performed some preliminary studies on the performance of FALCON with different combinations of a simple built-in random-sampling filter and a low-cost lossy compression algorithm supplied by the server which acts as the sender. Tables 1 and 2 show the setup time and total elapsed time for the transfer of 10MByte octets with the transfer size of 100kBytes with sender and receiver running on two Sun Sparc 20s with 256 MByte RAM over Ethernet and ATM respectively. Note that the main performance parameter measured is not absolute network throughput, but instead the throughput of data flowing into the stream since they would have to be transferred over the network (and be reflected in the "perceived" throughput) if they were not filtered or encoded as they flow through the stack at the sender-side communication channel endpoint.

By replacing OrbixWeb remote method invocation with the Java socket library as a communication mechanism over Ethernet cuts down data transfer time by almost 30% (from 17500ms to 12500ms). The introduction of a simple filter with a selectivity of 80% at the sender reduced the elapsed time by another 20%. Compressing data before sending (to 12.5% of its original size) also significantly reduced the elapsed time because of the large reduction in network communication cost, even though channel setup (which involves copying the decompression module from the server

Network Software	Setup	Total	Perceived Throughput
Original (OrbixWeb)	35ms	17500ms	4.57Mbps
FALCON Stream	50ms	12500ms	6.40Mbps
FALCON Stream + Filter (selectivity=80%)	70ms	9000ms	8.89Mbps
FALCON Stream + Compress (ratio=12.5%)	300ms	2000ms	40.0Mbps
FALCON Stream + Compress + Filter	300ms	1900ms	42.1Mbps

Table 1: Transfer time for 10MByte of Octets (100kByte chunks) over Ethernet

Network Software	Setup	Total	Perceived Throughput
Original (OrbixWeb)	35ms	6500ms	12.31Mbps
FALCON Stream	50ms	11000ms	7.27Mbps
FALCON Stream + Filter (selectivity=80%)	70ms	9000ms	8.89Mbps
FALCON Stream + Compress (ratio=12.5%)	300ms	2000ms	40.0Mbps
FALCON Stream + Compress + Filter	300ms	1900ms	42.1Mbps

Table 2: Transfer time for 10MByte of Octets (100kByte chunks) over ATM

to the client) and data compression presented some overhead. Stacking the filter on top of the compression layer further lowers the elapsed time of the transfer. The communication framework could easily be extended to not only support transparent compression of data, but also allow automatic decompression of compressed executable stack layer modules to reduce stack setup overhead (in a slow network).

In an ATM-based network, the benefit of the FALCON prototype is less significant. As a matter of fact, a simple FALCON stream (even after filtering 20% of the data) takes longer to transfer the 10MByte-dataset than OrbixWeb. This can be attributed to the inefficiency of Java (especially the Java socket library that is used as the FALCON network interface), as well as that of the FALCON prototype which implementation has not been tuned and optimized. Because ATM has a significantly higher bandwidth than Ethernet, the overhead in Java and the FALCON prototype becomes much more prominent.

The ATM-based timing confirmed our concern with Java run-time performance. Preliminary experiments have shown Java applications to be one to two orders of magnitudes slower that compiled C/C++ code. Just-in-time compilation is one of the techniques that promises to significantly improve the performance of Java programs by compiling Java bytecode into native code at run-time. By paying the price of compilation at run-time, we can avoid the overhead of interpreted execution of Java programs. Various Java JIT compiler vendors have claimed that JIT compilation improves Java performance by over an order of magnitude, to within 50-75% of the performance achieved by compiled C code. Unfortunately the availability of JIT Java compilers are currently only limited to Microsoft Windows platforms, even though they should become available on a wide variety of platforms in the very near future.

As a result, we are also implementing stack layers in C++ native code called in Java to achieve better performance in cases where performance is the primary consideration. In the C++-based implementation, transportable stack layer implementations are compiled into a shared library (that is tied to a specific platform) which are transferred from its source and then dynamically linked into the target process. In other words, this prototype implementation trades off safety and platform-independence for higher performance.

5 Related Work

The motivation for application level communication optimization is similar to that for Application Level Framing [3] which emphasizes the fact that a lot of knowledge about an application is oblivious to the network. This implies that the application should organize the data it is sending as a sequence of autonomous "frames", which will be at the same time the unit of transmission, the unit of control and the unit of processing. This allows for efficient design of application specific protocols, while minimizing queuing delays.

Application level framing advocates the use of integrated level processing which is the integration of multiple transmission control layers in a single loop to maximize the efficiency of modern processors. On the other hand, in the discussion of our framework we concentrated on the automatic composition of FALCON communication channels from simple stack layer implementations to encourage the reuse of different optimization techniques that are chosen at run-time to fit a particular application. Nevertheless, we do not preclude the use of ILP techniques to integrate multiple functionalities into a tight loop in the stack to squeeze the most out of the processing cycles.

The idea of stacking simple protocols to simplify the development and encourage reuse is motivated by Horus Protocol Stack composition [13] and Stackable file system [7]. However, stacks in these systems are constructed in an "ad-hoc" manner and are not systematically optimized. On the other hand, the FALCON optimizer adapts well-studied rule-based database query optimization techniques [6] to handle an extensible set of stack layer implementations that can be introduced at run-time by clients and servers in an environment supporting FALCON. The goal of database query optimization is to generate the optimal (usually minimal cost) query execution plan given the input logical query, system-wide statistics, a cost model, and various information about the semantics of available logical operators (e.g., algebraic transformation rules) and physical operator implementations (e.g., implementation rules specifying possible mappings between algebraic operators and implementations). In FALCON, the expected properties of a communication channel are analogous to a logical query that specifies the delivery of data from the sender process to the receiver over the network. Given an input "query" represented as the expected properties of the channel as well as information on available stack layers and up-to-date system statistics, the output of the channel optimizer is an optimal "execution plan" in the form of matching stacks in the communication channel's endpoints.

Many of the design issues involved are similar to those in developing a multimethod communication architecture [4] which involves implementing a high-level communication abstraction in a diverse collection of actual low-level communication methods. In multimethod communication, to initialize communication, a receiver creates a global pointer which contains communication end points for all communication methods ranked in some order. When the global pointer is transferred to the sender, the sender can then select (either automatically through a search or manually) the "best" applicable one based on some communication method specific criteria. While the multimethod communication architecture is only concerned with the selection of the communication method in the sender, we allow the choice of modules to instantiate a wide spectrum of layers (ranging from application level to communication level functions) to be partially determined by the communication requirements. We are considering adopting techniques developed in multimethod communication to realize the support for a wide variety of network protocols in FALCON.

6 Conclusions

We developed a general-purpose framework to construct communication channels supporting matching user-defined stack layers in the endpoints that are transparently called to manipulate data as they flow through. On the sender-side, the main objective of the stack is to reduce irrelevant data (e.g., through filtering and compression), while on the receiver side the stack undoes the effect of its counterpart (e.g., decompression) as well as eliminates data transfer (e.g., caching) and latency (e.g., prefetching). Knowledge of the application is obviously necessary to effectively filter unnecessary data at the sender. In addition, the receiver-side cache manager can also take advantage of application semantics; for example, predictive caching (often used in mobile computing environment) transparently prefetches data that is likely to be accessed by the application. Application behavior and data semantics can also be exploited to allow compression algorithms of different quality to be used if they satisfy the requirements imposed by the application. Similarly, distributed applications vary in their needs for communication support; in the case where multiple network communication methods are available, the quality of service guarantee of the chosen network protocol must satisfy application needs.

FALCON allows application semantics to be exploited to optimize network communication. This is achieved by allowing stack layer implementations on both the sender and receiver sides to be supplied by the same communicating process. For example, application semantics appear in the form of special filtering routine registered at the sender by the client requesting data, and decompression modules that are transported to the receiver to properly decompress data compressed/encoded by some data-specific algorithm.

A cost model is used by the FALCON optimizer to decided whether a particular stack layer should be introduced in the communication channel based on run-time statistics. The optimizer adapts well-studied rule-based database query optimization techniques; in particular, the input is a simple communication channel with application-specific filter and compression routine as input "operators" in the logical channel (analogous to a database query) and the output would be a physical communication channel with appropriate stack layer implementation instantiated (analogous to algorithms such as the different join algorithms and data access paths) in the stack (analogous to a query execution plan).

Our approach aims to eliminate transfer of irrelevant/unnecessary data based on application semantics, network configuration, and communicating process characteristics at run time. This is significantly different from (but complimentary to) the approach taken by most networking researchers which aim purely to maximize the throughput of network protocols by improving the implementation or picking the right one for the situation. Nevertheless, we are investigating the possibility of integrating network protocol optimization techniques implemented in Nexus and Horus at the network protocol layer of our prototype.

In a slow network such as Ethernet, it has been shown that the throughput bottleneck is usually the network instead of the networking software. On the other hand, in a high-speed network, the bottleneck is believed to often be the client application (including display and storage subsystems) as well as the networking software. The implication of this observation is that our scheme of eliminating irrelevant data before it reaches the network should be valuable in either type of network but in slightly different ways. In a slow network in the network throughput is maximized, we increase the perceived network speed through filtering and compression. In a fast network, we possibly reduce the amount of data that needs to be manipulated by the client application if filtering is effectively performed in the sender; the effect of compression/encoding is dependent on the compression ratio, and the cost of compression compared on network cost.

To allow executable modules implementing a stack layer in the communication substrate to be supplied by either communication peers, our framework supports the basic distributed computing pattern of transporting a executable module from one process to another (possibly running on a different platform). A dynamic, platform-independent, and "safe" programming language such as Java is particularly suitable for implementing transportable executable modules in our framework. In addition to being a fundamental building block of our dynamic communication framework, the support for transportable executable module (passed from a process to another as a parameter to remote procedure call or remote method invocation) also has very significant application to the construction of extensible systems such as OODBMSs [8].

As a proof of concept, we have prototyped FALCON in CORBA to support data streaming and application-level communication optimization. CORBA is used mainly as a naming and location service, while remote method invocations are used as the mechanism to set up a communication channel in the context associated with the particular remote method. Preliminary performance numbers are promising and confirmed the usefulness of FALCON in distributed object computing despite inefficiencies in the prototype implementation.

In conclusion, We believe that the design of FALCON has achieve the following high-level requirements for network optimization.

- **Significant Performance Gain and Low Overhead.** The goal of the framework, first and foremost, is to improve the performance of network applications. In particular, the implementation has to demonstrate a significant performance gain for a wide variety of applications. In addition, the overhead incurred by the initialization process of FALCON communication channels should be small to limit the adverse effect on communication latency.

- **Collaborative.** Both the receiver and sender objects are able to contribute stack layer implementations that are specific to their capabilities, characteristics, and run-time environments; the actual instantiations of the stack layers are decided through negotiation or cooperation between the receiver and the sender.

- **Simple and Transparent.** The framework has to be simple and easy to use in developing distributed applications in order to encourage use by application programmers. Client developers are expected to be large in number and have widely varying skill and knowledge levels. Client applications should be able to take advantage of the FALCON communication framework by exploiting server-provided stack layer implementations if they simply view the FALCON channel as a regular socket-like communication channel. If a tradeoff arise between the complexity of the client and that of the server, we sacrifice the simplicity of the server to keep the client simple.

- **Flexible and Highly Extensible.** The proposed framework has to be highly extensible so that a wide variety of communication stack layer implementations can be introduced to fit the communication characteristics of different channels.

6.1 Future Work

The following is a list of potential extensions to and issues related to our proposed communication framework that we are currently engaged in studying.

- **Security.** Our communication framework requires a process to be able to execute stack layer implementations received from its communication peer at run-time. Our prototype implementation relies on the security manager of the Java virtual machine to limit the executable module's access to local system resources (a Java object retrieved from a remote site by default does not have access to the local disk and can only communicate with processes residing on its originating site). While this guarantees system safety by limiting the damage a hostile executable module can do, it also restricts the range of operations that can be performed by a friendly executable module. We are interested in studying techniques that will allow us to safely relax the limitations placed on executable modules originated remotely. For example, sophisticated authentication, authorization, and encryption will allow a process to more reliably determine the friendliness of a remote module (compared to simply using the originating site as the discriminating factor as in Java).

- **Broadcasting (or Group Communication).** Our current design only consider the optimization of point-to-point communication. Support for group communication such as broadcasting will be a worthy extension to our proposed infrastructure.
- **Dynamic Reconfiguration of a Communication Channel.** Resource availability to an application's channels changes. Therefore a stack composition that is optimal when it is constructed may become sub-optimal over time. For a channel that are short-lasting each time it is used, it can be invalidated from the channel cache periodically or when the network environment changes to force the set up of a new channel instance with potentially choices for stack layer implementation. More interestingly, a channel supporting pipelining of a lengthy stream may be so long-lived that its resource availability can change significantly enough to warrant its reconfiguration dynamically.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] N.S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail. In *Proc. IFIP working Group 6.5 ULPAA (Upper Layer Protocols, Architectures, and Applications) '94 Conference*, 1994.

[3] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM '90*, pages 200–208, 1990.

[4] I. Foster, J. Giesler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *To appear in Journal of Parallel and Distributed Computing*, 1997.

[5] G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, Feb 1994.

[6] G. Graefe and W.J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. 9th Int'l Conf. on Data Engineering*, pages 209–218, 1993.

[7] J. S. Heidemann. *Stackable Design of File Systems.* PhD thesis, University of California, Los Angeles, 1995.

[8] V.E. Jones and M. Winslett. JaDE: Access control in a java-based object database. In *Proceedings OOPSLA'95 Workshop on Object Database Behavior, Benchmarks, and Performance*, 1995.

[9] I. Pyarali, T. H. Harrison, and D. C. Schmidt. Design and performance of an object-oriented framework for high-speed electronic medical imaging. In *Proceedings of the Conference on Object-Oriented Technologies and Systems*, 1996.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[11] R. M. Soley, editor. *Object Management Architecture Guide (2nd Edition)*. Object Management Group, 1992.

[12] C.J. Turner and L.L. Peterson. Image transfer: An end-to-end design. In *Proceedings of ACM SIGCOMM '92*, pages 258–268, 1992.

[13] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in horus. In *Proceedings of Principles of Distributed Computing Conference*, 1995.