Dynamically Configurable Communication Protocols and Distributed Applications: Motivation and Experience

Robin Kravets, Ken Calvert, Karsten Schwan
{robink, calvert, schwan}@cc.gatech.edu

GIT–CC–96-16

May 1996

Abstract

Due to the diverse communication requirements of today's distributed applications, our work has led us in the direction of dynamically configurable protocol systems. This paper motivates the design of a framework for such systems. We discuss the initial study that drove the design of our framework, and describe the framework and the associated interfaces. Finally, we present the results from an experiment involving an adaptable application using a variable reliability protocol.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

Dynamically Configurable Communication Protocols and Distributed Applications: Motivation and Experience

Robin Kravets, Ken Calvert, Karsten Schwan

{robink, calvert, schwan}@cc.gatech.edu

Abstract

Due to the diverse communication requirements of today's distributed applications, our work has led us in the direction of dynamically configurable protocol systems. This paper motivates the design of a framework for such systems. We discuss the initial study that drove the design of our framework, and describe the framework and the associated interfaces. Finally, we present the results from an experiment involving an adaptable application using a variable reliability protocol.

1 Introduction

The recent past has seen a tremendous increase in networking capabilities, in both available bandwidth and decreased latency. Despite these improvements, applications continually stay one step ahead of the networks. The more resources that are available, the more resources that applications need. This is especially true for multimedia applications, which require high bandwidth and low latency communication. These applications not only have stringent requirements, they also have many different types of data to transfer, each of which may have different service requirements. The problems created by these requirements become even more apparent when the service requirements of the applications in question may vary over time.

In order to make a case for the need for dynamically configurable communication protocols used in conjunction with adaptable applications, we will show that the functionality gained by using configurable protocols matches that which is needed by adaptable applications. Specifically, our goal is to show that this functionality provides certain classes of applications with the ability to improve their performance in a simple straightforward way.

Our specific work is focused on showing the importance of being able to dynamically configure protocols during the lifetime of the communication. Our experimental evidence demonstrates that actual applications can benefit from dynamic protocol configurations beyond what is available from the current state of the art. To this end, this paper develops concepts, provides implementation and evaluates functionality that enhances the current state of dynamically configurable protocol systems.

The rest of this paper is organized into the following sections. Section 2 provides some of the background and motivation for using dynamic communication protocols and for having adaptable applications. Sec-

tion 3 describes the initial study we performed to motivate this work and to develop our framework and interfaces. Section 4 describes the dynamic communication framework that we have designed and built. Section 5 outlines the interfaces and functionality of the application, framework and protocol functions. Section 6 describes a variable reliability protocol that we designed and an adaptable application that makes use of it. Section 7 highlights some issues and directions for future work.

2 Background and Motivation

Distributed multimedia applications are straining the resources of today's networks. The ability to better adapt to available resources may provide applications with the ability to continue working in situations where non-adaptable applications would fail or perform very poorly. Current technology provides applications with very limited communication control. Applications are given the choice between specific protocol stacks, and must work around the fact that they cannot easily change communication parameters. This leaves the burden of maintaining flexible communication to the application. Configurable protocol systems solve this problem by supplying applications with a simple model for communication. Applications no longer need to concern themselves with the details of the communication. The configurable protocol systems manage the communication resources in an independent, modular manner which is easily accessible by any application. The remainder of this section contains the motivation and related work for both configurable protocol systems and adaptable applications.

2.1 Configurable Protocol Systems

A commonly used abstraction for the protocol processing associated with a communication channel is the protocol stack. A protocol stack defines what protocol processing is executed during communication and in what order it is executed. In general, a single protocol stack provides a specific service. Configurable protocol systems provide applications with the ability to chose communication functionality from a set of available protocol modules. These protocol modules can be combined in any correct manner into a protocol stack that can provide the application with the appropriate service. Applications can simultaneously use multiple protocol stacks. The problem of providing the correct services for an application is now only limited by providing the correct building blocks. New applications with service requirements that have not previously been considered can still make use of a configurable protocol system by enhancing its set of protocol modules.

For this discussion, these systems are broken up into two groups on the basis of when configuration is performed.

Connection Time Configuration : The first level of configurability allows the application to configure its communication at connection time. Connection time configuration allows the application to have different communication channels with different service requirements, by providing multiple static protocol stacks. The limitation here is that once a communication channel has been configured, it can not be changed during its lifetime. Even if the channel only handles one type of data, the service requirements for that data may change over time. With connection time configuration, the channel must be torn down and reestablished for configuration changes to be made. Connection time configuration also limits a channel to handling one type of data correctly,

since one type of service must be chosen for the lifetime of the communication.

Dynamic Configuration : The second level provides applications with the ability to dynamically configure communication channels at run time. Dynamic configuration solves the problem of handling multiple data types over one single channel as well as the problem of changing service requirements over time. Dynamic configuration solves these problems by allowing the communication channel to change over time without tearing down the channel. Our framework implements this functionality.

Some examples of connection time configuration include the following. HOPS (Horizontally Oriented Protocol Structure) [Haa91] provides applications with a single, higher-layer protocol that successfully provides communication over diverse networks. In the *x*-kernel [HP91] [OP92], protocols are divided into modules, and these modules are connected in a protocol graph. Connections can choose a protocol path for their communications, again on a per-session basis. Bhatti and Schlicting [BS95] suggested an enhancement to the *x*-kernel that provides applications with more flexibility, but is still restricted to the original design of the *x*-kernel. Both HOPS and the *x*-kernel provide some of the functionality that we were looking for, specifically in the functionality of the protocol selection in HOPS and the protocol stack determination in the *x*-kernel. Our framework includes this functionality, but expands it to provide applications with a less restrictive model. Our belief is that restricting the application to connection time configuration is an unnecessary requirement that may lead to poor application performance. Some experimental results in the area of parallel protocols with connection time configuration were presented by Lindgren, Krupczak, Ammar and Schwan [LKAS93]. The current implementation of our framework provides sequential protocol processing, but the design allows for both sequential and parallel processing.

A number of proposals have been made for providing dynamic communications through configurable protocol systems. The goal of ADAPTIVE *A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment* [SBS93] is to provide automated support for composing lightweight and adaptive protocols. Their approach employs a collection of reusable "building block" protocol mechanisms that may be composed together automatically at runtime. This work emphasized the need for dynamically configurable protocols, but was limited to experimental results. Da CaPo (**D**yn**a**mic **C**onfigur**a**tion of **Pro**tocols) [PPVW93] is another approach to modular configurable protocols. In Da Capo, configuration is done with respect to application requirements, properties of the offered network services and available resources in the end systems. This work differs from ours in that the connection manager, configuration manager and resource manager are built into the framework. Although these are important components of a dynamically configurable protocol systems, our implementation separates them out from the main functionality of providing configurable communication. In [ZST93], Zitterbart, Stiller and Tantawy also describe a communication subsystem that allows applications to request individually tailored services. We provide results that demonstrate the necessity for this type of refinement.

2.2 Adaptable Applications

Multimedia applications have stringent Quality of Service (QoS) requirements. Data must arrive by a specific time, or it can no longer be used. Due to the fact that there is no widely available way to ensure QoS with today's communication networks, specifically when using the Internet, there are two

suggested ways of dealing with the problem of working with the available bandwidth and information available from the network. The first solution is to use resource reservation throughout the network. The second solution is to provide applications with information regarding the state of the network and allow the applications to adapt to the available resources.

RSVP [ZDE+93] and the work done in the Tenet group [FV90] are approaches to reserving network resources. ATM is intended to provide some level of QoS management, but many current implementations do not support this functionality, and it is not clear when they will. One problem with these solutions is that they require changes throughout the network. The require that each node understand the idea of reserving resources. Another problem is that some resource reservation schemes that do not allow for resource renegotiation. The cost involved in reconfiguration of the communications channel may make it prohibitive for the application to make changes to its reservation when its requirements change.

When applications are provided with information regarding the state of the network, they can take this state information and information about the current requirements of the user and adapt to the available resources. This solution puts more of a burden on the application, but since the application knows best about the QoS that it will need, this seems to be the best place to put the control. This solution also makes fewer demands on the individual nodes of the network. QoS is monitored and adjustments are made end-to-end. An example using the INRIA Videoconferencing System (IVS) showed that it is possible for applications to adapt and still receive the QoS that they require [Dio95]. Gopalakrishnan and Parulkar [GP94] define some issues involved in determining what knowledge the application or endsystem may have that can help in providing the QoS requirements.

Since it is likely that some sort of resource reservations systems will be available in the future, as well as other improvements that will be provided by the network itself, we do not want to lose the use of these resources. Allowing the application to adapt to available resources does not rule out the use of networks which provide resource reservation or other enhancements internal to the network. Such networks are worked into a larger picture and used by the application alongside more standard networks. Our definition of an adaptable application includes the idea that if resource reservation were available, the application could decide to make use of it. But if it is not available, the application can adapt to what is available. By providing the control on an end-to-end basis, the application is given the most information available and can decide what resources are best fit for its uses.

3 Initial Study

Dynamic applications often have changing protocol processing needs. As an example, we built an application that demonstrates some of this functionality. The following section describes a multimedia slideshow application that can dynamically change its protocol processing needs on a message by message basis. This functionality in an application enabled us to determine a working interface for our protocol framework. By using an application that has a wide variety of protocol needs, we built up a set of protocol functions for future use.

3.1 Multimedia Slideshow

In general, most multimedia applications are rate and frame based applications. In these applications, individual frames may have differing requirements. This puts a high demand on the communications system, or requires that the application supply the desired functionality itself. By providing the applications with a simple interface, our framework can supply the necessary functionality.

The first application we developed is a multimedia talk session and slideshow. The application connects users and provides the ability to transmit continuous voice, sound files, image files and text. The motivation behind creating this application was to provide a working environment for testing and developing dynamically configurable communication protocols. Having different types of data gives us a broad range of protocol processing configurations.

The goal of this application is to show that a dynamic application that transmitted multiple data types could be simplified by using our framework with its generic interface (see figure 1). The dynamic part of the application is that it has the ability to request different protocol stacks on a message by message basis. For example, the application may start out sending text unencrypted, but at some point during the run decide to increase security. The application simply informs the framework to make use of a new configuration for text with some type of security protocol. The application has the knowledge of what type of service it requires and the framework provides the functionality to provide that service.

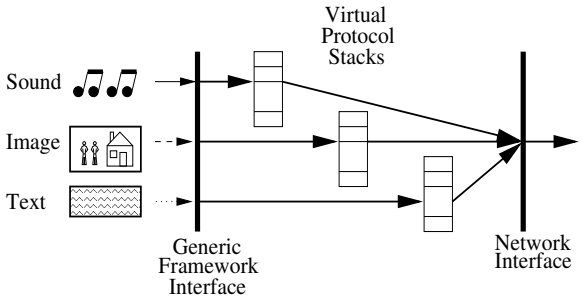

Figure 1: Multimedia Slideshow

3.2 Protocol Suite

Different applications will require varying protocol functionality. By supplying applications with a rich protocol suite, we are able to support a number of different dynamic protocol stacks for each application. Supporting as many protocols as possible, allows the application to find a protocol stack that best fits its communication needs. Our protocol suite provides many standard protocol functions, but can simply be enhanced to include any current or future protocol.

The protocol suite is made up from the following groups of protocol functions.

- Security - DES and IDEA encryption.
- Compression - GSM and ADPCM compression for voice; JPEG compression for images; "Berkeley compress" for text.
- Data Size - determination of the maximum data size for a message.
- Rate Control - leaky bucket flow control.

- Transmission Monitoring - count of the number of dropped messages; count of the amount of data transfered.
- Reliability - TCP for reliable data transfer; UDP for unreliable data transfer.

As a step toward providing applications with network feedback, applications have available to them a simple set of transmission monitoring protocol functions. These protocol functions do not touch the data, they simply collect some statistics that can later be checked by the application in order for it to best determine what protocol processing it needs. Future work will provide the applications with more extensive network feedback.

3.3 Application Feedback for Dynamic Protocol Inclusion

One of the benefits of dynamically configurable protocol systems is the ability to handle feedback about the state of the communication. In order to demonstrate this functionality, we modified our original application to include dynamic inclusion and exclusion of a protocol transparent to the application. In the original application, the application itself determines when changes to the protocol stack need to be made. With this modification, there is an intermediate control module that monitors some specific information (see figure 2). The control module makes decisions about protocol processing based on this information. Our example focuses on bursty audio communication.

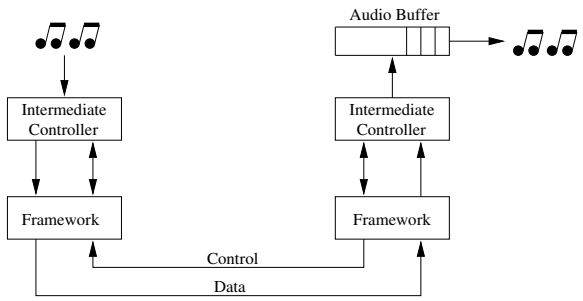

Figure 2: Intermediate Control Module

The dynamic configuration in this example is performed in response to feedback from the receiving control module. The experiment takes advantage of the SGI Indy's ability to query the audio buffer. If the receiving control module notices that its audio buffer is getting overloaded, it sends a control message to the sending control module informing it to turn on its leaky bucket flow control algorithm. As the audio buffer empties out, it sends a control message informing the sending side to turn the flow control off.

The ability to switch on and off the leaky buck flow control is most useful in the situation where the application is sending many short bursts of audio. In these cases, the sending application need not keep track of previous audio bursts, in order to determine if this burst would overflow the receiver's audio buffers. The success of this experiment was determined by sending pieces of a song, since music is more sensitive to loss than voice. As a qualitative measure, the quality of the songs were not lessened by using this dynamic inclusion/exclusion of the leaky bucket algorithm. Quantitative measurements are still under way.

4 Design of a Dynamic Protocol Configuration Framework

The basic idea of a dynamic protocol configuration framework is to be able to allow applications the flexibility they may need. The result of this flexibility includes allowing the ability to change communication configurations on the fly. We may want to be able to turn protocols on and off during communication as well as change protocol parameters during the lifetime of the communication.

Tau [CKK96, Cal93] is a framework for composing end-to-end protocol functions. The framework we have built implements a subset of the functionality designed into Tau.

The dynamic configurability in our framework is realized through the use of a few very simple abstractions. Our framework is designed to provide some very specific functionality. The main goal is to provide the ability to configure communications on a message by message basis. To this end, each piece of our framework lends its specific capabilities. In order to be able to communicate with the framework, the application is provided with an abstraction called a configuration. A configuration represents a protocol stack that the application wants to use at some point in the life time of its communication.

The framework itself is broken up into three parts:

- The **protocol infrastructure** provides the ability to maintain state information for protocol configurations being built on the fly.
- The **metaheader protocol** provides the information necessary to correctly process each message.
- The **protocol functions** and the **standard protocol interface** provide the ability to "plug in" any protocol at the appropriate place.

The following section will describe the implementation of our dynamic protocol configuration framework that we have built. Evaluation of this framework was performed through the use of complex protocol functions on top of TCP/IP and UDP/IP. This decision was made for simplicity of use and ease of expansion. Although most of these techniques expand to the transport layer, we decided to experiment with our framework in user space. This assumption was made to ease development and allow for simple changes needed for any type of enhancement to our framework.

4.1 Configurations

In order for an application to be able to make use of our framework, we provide an abstraction called a configuration. A configuration is a group of protocol modules that have been specifically put together for some specific purpose. In essence, a configuration represents a virtual protocol stack that the application has decided to use for some instance of communication. It is up to the application to decide what combination of protocols is useful for it. To allow for the desired flexibility of communication, an application may define multiple configurations for each communication channel. Through the use of each specific configuration, the application can specify exactly which protocols should be used.

Two levels of configuration changes are defined. The first affects messages that are processed similarly. These messages are considered to use the same configuration. Changes to the processing of these messages is done at the protocol level, and will affect all messages that use this configuration. These types of changes generally involve modifying protocol parameters, but not changing which protocols

are being used. For example, consider a configuration that includes JPEG and DES. JPEG provides a few parameters that affect the compression processing. If we want to keep using JPEG, but want to change from fast JPEG to slow JPEG, we would keep the same configuration, and make changes at the JPEG protocol level.

The second level of configuration changes is used when messages need to be processed with a different set of protocols. These changes involve using a new configuration for the messages in question. If we consider our JPEG and DES example again, we may decide to change from DES encryption to IDEA encryption. In this case, we would create a configuration that contained JPEG and IDEA and switch to use this new configuration. Another example might be that we had some control information that needed to be processed with some control protocol and a different encryption key. In this case we would leave the original configuration alone and again create a new one that included the control protocol and DES. For this example, the image processing and the control processing could continue in parallel, each using its own configuration.

4.2 Protocol Infrastructure

The primary design objective of the protocol infrastructure is to provide two mechanisms: protocol function composition and multiplexing. This functionality is provided in a manner that supports various performance-enhancing techniques, while preserving modularity in some form. The idea is that these mechanisms should work with an extensible set of policies, in order to support a wide variety of applications, including those whose requirements are not yet fully understood. Our goal is a generic model of protocol processing, in which the protocol functions are separated from the details of the "glue" that binds them together.

Because protocol functions are not layered in the protocol infrastructure, they do not attach their headers directly to outgoing data units, nor extract them from incoming data units; instead, this is handled by the protocol infrastructure. The generic protocol model defines the interface between the protocol infrastructure and each protocol function. The requirement that this interface be specified represents an opportunity to reduce the costs of porting protocol implementations by isolating, as far as possible, the specification of a protocol's function from the "glue" used to combine it with other protocols. This is a key design motivation of the protocol infrastructure.

A logical block diagram of an implementation is shown in Figure 3. Each protocol function module is viewed as a (passive) transducer, which is given inputs (state information, control parameters, incoming header, and possibly data) and produces outputs (updated state information, control parameters, outgoing header, and processed data). The architectural "glue" is provided by the demux-and-dispatch function. It selects and coordinates between the protocol functions, providing them with inputs based upon external events, and collecting and passing on to the external environment (i.e. the user, the network, auxiliary functions such as timeout and buffer management) their outputs.

4.3 Metaheader Protocol

The metaheader protocol provides the necessary information for determining what protocols are involved in each instance of communication. The metaheader protocol implements the multiplexing

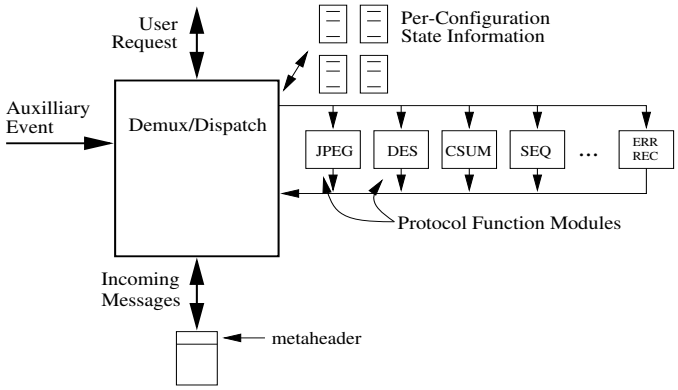

Figure 3: Architecture of Protocol Framework

and composition mechanisms. A metaheader is an extended message header that provides enough information for that message to be handled and processed correctly at the receiving end.

The metaheader is built from three building blocks: message header information, protocol header descriptors and individual protocol headers. The message header information provides the information necessary to understand how the rest of the metaheader was built. It includes header length, number of protocol headers, sender and receiver application identifiers, and sender and receiver configuration identifiers. The protocol header descriptors and the individual protocol headers come in pairs. The header descriptor determines which protocol is next and defines the length of the specific protocol header. The individual protocol headers are defined by the specific protocol. Combined, these three building blocks provide the receiving side with a map of how the message was processed at the sending side, and how it now needs to be processed at the receiving side. Figure 4 shows an example message with metaheader and header descriptors.

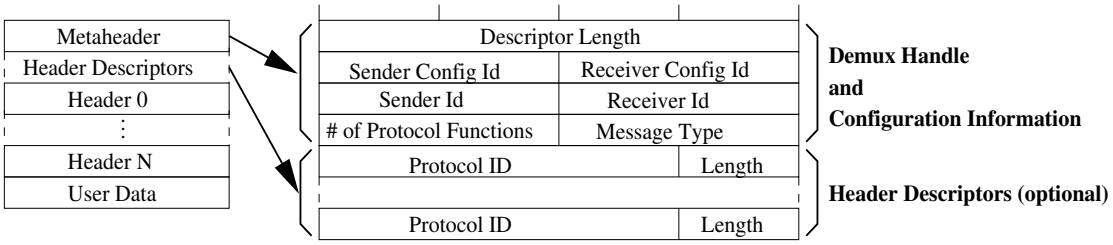

Figure 4: Example Message using Metaheader Protocol

4.4 Protocol Functions

Each protocol function defines a specific function that can be processed independent of other protocol functions. This functionality can vary in complexity from simple checksumming to the entire TCP protocol. The protocol functions together provide the "menu" from which an application can choose the services it desires. We envision communication services implemented by composing atomic single-function protocols from a "menu of functionality", as have others [OP92, ZST93, Haa91, PPVW93, SBS93]. For example, a service for a reliable, secure image application could be implemented with

JPEG, DES, a sequence numbering function, and two different reliability functions (one for request retransmission and one for response error detection and retransmission).

5 Interfaces

The design of our framework leaves a clean distinction between three interfaces (see figure 5). The application has a set of functions that it uses to communicate with the framework. These include all the entry points necessary for configuration creation and for sending and receiving data. In addition, the application can retrieve information about a specific protocol in a specific configuration. In order to access this information, the processing passes transparently through the framework. This design allows for the application to access the protocol directly, but without having any knowledge of the design of the framework. The final set of functions provides a generic interface between the framework and the protocol functions. Each protocol provides a set of standard entry points that are accessible by the framework. Our current implementation does not have an interface defined for communicating with the network. The next stage of our design will include this specification.

Figure 5 depicts the control functionality to be a separate entity from the application. In our initial experiments, this control functionality was part of the application. Our design allows for the control functionality to be either place.

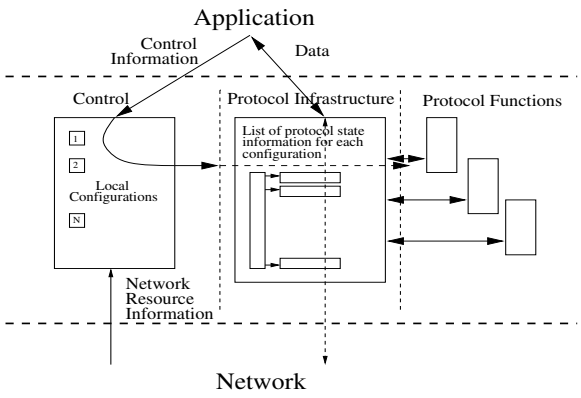

Figure 5: Three Interfaces between Application, Protocol Infrastructure and Protocol Functions

5.1 Application to Protocol Infrastructure Interface

In order to create a configuration, an application follows these steps:

1. Create the configuration : `ConfigId createConfiguration()`

   This function returns a handle to a new, empty configuration. This handle is used by the application to specify this configuration in the future.

2. Add protocols to the configuration: `addProtocol(ConfigId, ProtocolId, ProtocolState *)`

   This function adds a protocol to the configuration, allocates memory for the protocol's state and header information, and initializes the protocol. This generic interface allows the Communication

Layer the ability to set up any necessary information for a protocol without having to know the specifics of that protocol.

3. Add a transport protocol: `addTransportProtocol(ConfigId, ProtocolId, ProtocolState *)`

   This function sets the transport protocol (i.e TCP or UDP) for this configuration. This particular distinction is necessary in our current implementation. A similar distinction may need to be made in the future depending on what platform the framework is built on.

The key idea behind protocol configurability is that the application should be allowed to choose what protocols it needs for a specific communication channel, stream, or even message. Once an application has set up the appropriate configurations, it can now use a specific configuration handle to tell the communication layer which configuration is to be used for the current message. The order of processing of the protocols is determined by the order in which they are added to the configuration. Outgoing messages are processed in order; incoming messages are processed in reverse order.

Messages are sent using the function `sendData (Data *, Length, ConfigId, Label *)`. This simple call allows the application the freedom to choose a specific configuration on a message by message basis. The function of the label is to provide the application with the ability to send control information out-of-band. The framework takes this label and includes it in the message similarly to a protocol header. Providing this label allows the application to support the concepts of application level framing (ALF) [CT90]. ALF suggests that since the application has the most knowledge about the data that it needs to send, the communication system should respect application specified data boundaries. To this end, sufficient information should be included in each (application) data unit to enable the receiving application to deal with it, regardless of its order with respect to other data units. Our distinctions between label and data allows the framework to correctly combine processing the data with placing the data in its final location.

When an application wants to receive a message, it makes the call `recvData (ConfigId *, Data *, Length *, Label *)`. Since the application does not have foreknowledge of what type of message it might be receiving, or with what configuration id might be used, the framework provides this information. As new configurations are used on the sending side, new configurations are dynamically built on the receiving side. In general, the application may or may not need to know what configuration was used to process an incoming message. By providing the ConfigId, the application has access to application specific state information about this message if necessary.

5.2 Application to Protocol Function Interface

A set of special purpose functions are provided to allow the application the ability to communicate directly with the protocols. These functions allow the application to set and retrieve application specific parameters.

- Setup the appropriate parameters: `resetProtocol (ConfigId, ProtocolId, ProtocolState *)`

  This function allows the application the ability to inform the protocol that certain application specific parameters need to be changed. The protocol takes this information and makes any

changes it deems necessary.

- Check protocol parameters: `getProtocolState (ConfigId, ProtocolId, ProtocolState)`

  This function allows the application the ability to check certain application specific protocol parameters during the run of the application.

These functions are used when the application needs to inform a protocol of parameter changes, or the application needs information about the state of a protocol. If we consider the example of changing the JPEG compression parameter described in section 4.1, the application would use the function `resetProtocol`. In a situation where a flow control protocol is being used, the application may want to query the flow control protocol to see if the current message can be transmitted. In this case, the application would use the function `getProtocolState`.

5.3 Protocol Infrastructure to Protocol Function Interface

This interface is defined in terms of a set of entry points corresponding to various events. Currently, each protocol must provide the following entry points:

- Initialization
- Send Processing
- Receive Processing
- Acknowledgment Processing
- Timeout Processing
- Reseting or Changing of Protocol Parameters

When control is passed to a protocol function, it receives a set of parameters which may include some or all of the following:

- local control information (e.g., data size, user parameters, destination application identifiers)
- remote control information (the header associated with an incoming message)
- user data
- persistent state information relevant to the connection or endpoint.

6 Experience

We have observed that there can be a high degree of variability in run time protocol behavior for complex distributed application. In an effort to take advantage of this variability through the use of configurable protocols, we built an adaptable application that makes use of our framework and protocol suite. This experiment was built to demonstrate the uses and benefits of configurable protocols. The goal is to show that by using this dynamic protocol framework, our application can improve its performance.

Our efforts have moved beyond simple examples and standard benchmarks to show the effects of the use of configurable protocol systems on cost and performance. In order to understand the demands

on high performance protocols, we built an applications with characteristics representative of today's high performance distributed applications.

The following sections also describe a new protocol we implemented. This protocol provides the application with the ability to specify the importance of different pieces of data. The application described represents functionality from real world applications and use of our variable reliability protocol.

6.1 Variable Reliability

During the development of our applications, it became clear that the two types of reliability provided by TCP and UDP were not sufficient for our applications. There were other situations where the applications could withstand some threshold of loss that was chosen by the application. This led us to provide a mechanism that can be used to create a variable reliability protocol, in which the specific policy for determining what reliability means can be determined by the application that is using it. Our protocol is built upon a very simple abstraction which can be expanded depending on the desired policy. This abstraction is based on a concept called reliability classes. A reliability class groups together application data that has similar reliability requirements. Reliability classes are defined by the application. The determination of which data belongs in which reliability class is also defined by the application.

The mechanism used for providing variable reliability is a simple counter for each specified class. The sender passes data to the protocol and specifies which class it should be sent in. The protocol maintains state that keeps track of the counters for each class. As a new message is transmitted, the sending protocol includes the counters for all classes in a protocol specific header. When the message is received at the other side, the receiving protocol can compare all of the counters and determine if any messages were lost.

For applications which only send data sporadically, a "heartbeat" message can be included. This heartbeat would include the counters for all classes. When data is being sent fast enough, no heartbeat is necessary, which removes the problem of the heartbeat causing congestion or delays.

When the receiving protocol notices a lost message in a class, it calls a function that implements the policy for that reliability class. This policy is determined by the receiving application, and can be transparent to the sending application. In other words, the sending application determines what class a specific piece of data is sent in, but the receiver determines its policy for that class. This allows for different receivers to implement different policies for the same reliability class. On the sending side, the application must inform the protocol what policy should be used for each reliability class when data has been lost.

Studies have shown that for some applications, packet loss can often be tolerated. Dempsey, Liebherr and Weaver [DLW94] provide some insight into the usefulness of allowing the application to determine when retransmission is a viable option. Recent work by Marasli, Amer and Conrad [MAC96] shows some analytical studies for retransmission-based reliability.

6.2 Distributed Robot Simulation

The class of complex distributed interactive systems combines a mix of human, simulated and mechanical control. These different parts have differing requirements for latency, reliability, consistency, and bandwidth. Some elements of these types of future systems are examined in this paper. The specific application we implemented is a distributed robot simulation. This application involves the sharing of a world view between distributed robots. The goal for this application is the use of the variable reliability protocol by a complex dynamic application.

The world view in this application is processed as an image and passed back and forth between the robots as it is being updated. The consistency and updates are handled similarly to a distributed shared memory (DSM) model. The world is broken up into blocks, where each block has an owner which is responsible for coordinating reads and writes as well as sending updates to the other robots. Each robot moves through the world, changes the data in the world as it moves and receives updates about other changes.

Two degrees of configurability were explored in this experiment. The first is the dynamic determination of what data is important to each robot. The second is to introduce the concept of the variable reliability protocol.

The amount of network traffic and processing resources can be reduced through the use of simple window abstraction, while still providing acceptable accuracy of the world for the robot. The most important pieces of the world view for each robot are those surrounding that particular robot. The robot needs the most recent information about any updates to these pieces. If each robot always receives updates about any changes, it actually may be receiving information that it is not interested in. Through the use of the window abstraction, we allow each robot to register itself for updates about data that it is interested in. The robot now only receives updates about the data inside that window (see figure 6). The window is always centered on the robot, and, as the robot moves, the window will move with it.

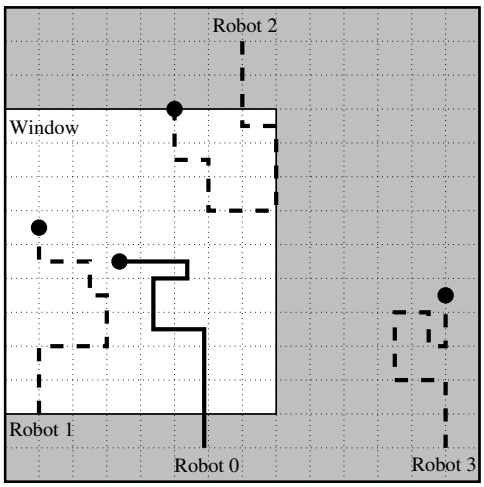

Figure 6: Snapshot of Distributed Robot Simulation with Window

We also experimented with the use of the variable reliability protocol. This protocol lends itself well

to be used in a configurable protocol system, due to the fact that each piece of data potentially has a different reliability class. In this application, the detection of a lost message is passed on to the sending application. This puts the burden of buffering the data on the side of the application, not the protocol. The protocol needs only to keep some label for each data that is sent. When data is lost, the protocol passes the label to the application, and the application can then decide what to do. This can be very efficient in the case of data that is constantly being updated. If the protocol were to buffer the data and then retransmit it, it may be retransmitting old data. By allowing the application to decide what to do upon data loss, the application can make use of the very explicit information that it has about its own data.

The reliability class of an update is determined by the update's proximity to the robot receiving the update. As we move further away from the robot, the importance of those updates lessens. The sections that are out of the robot's view may only require periodic, unreliable updates, or no updates at all. The application can dynamically set the degree of reliability that it wants for a specific reliability class. The application can dynamically change the assignment of data to a specific reliability class. This allows the application to chose what data is most important to it, and pay the overhead of reliability for that data.

For this application, we combine the idea of the window view with the variable reliability protocol. The window is divided into concentric "rings" around the location of the robot (see figure 7). Each of these rings corresponds to a reliability class. The loss tolerance in the distant rings is higher than that of the closer rings. In other words, as the updates come from further away, the reliability of the update is relaxed.

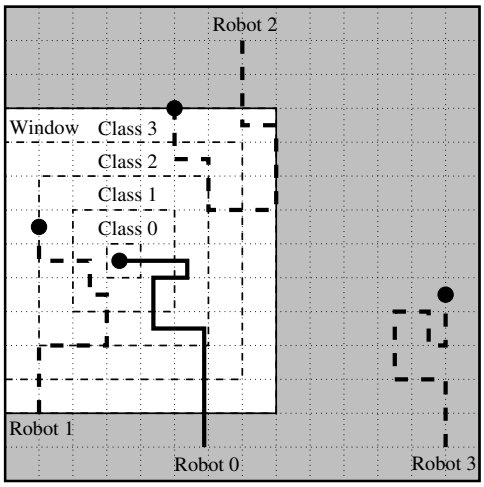

Figure 7: Snapshot of Distributed Robot Simulation with Window and Reliability Classes

6.3 Experiment Setup

The experiment was run with the following variables:

- Image Size = 512 x 512 pixels
- Block Size = 32 x 32 pixels ($\Rightarrow$ 1024 byte data size)
- Image had 16 x 16 blocks

- Simulated Loss Rate = 1 in 100 messages
- JPEG compression of image data

The loss rate was chosen to be relatively high in order to produce the necessity for retransmitting data. JPEG compression was used to incur some processing overhead for each data message.
During a run, each robot takes 5000 steps and then calculates its statistics. Since the time of the run is also dependent on how much each robot must process incoming messages from the other robots (i.e. how much time the other robots read and write to the blocks this robot owns), the times vary for each robot.
Three sets of experiments were run. The results are averages from 4 runs of each type.

- **Total View**: Each client is registered for all blocks. A retransmission is asked for any lost message.
- **Simple Window**: Each client has a dynamic window for which it is registered. The window size is 8 blocks in each direction. Within this window, a retransmission is asked for any lost message.
- **Window with Classes**: Each client has a dynamic window for which it is registered. The window size is 8 blocks in each direction. Within this window there are four reliability classes. Retransmission is determined by the class in which the loss was perceived.

6.4 Results

6.4.1 Simple Window

As would be expected, we could see a sharp reduction in the number of updates processed by the robots when the window size was smaller than the entire world (see table 1). There is a certain amount of trade off in this system. In order to keep track of the window for each robot, control messages were sent each time the robot window changed (see table 2). In our observations, the cost of processing the control messages still made it beneficial to use the window, in that processing the control messages was still less costly than sending all updates (see table 3).

6.4.2 Window with Reliability Classes

Although, through the use of the variable reliability protocol, we did see a drop in the number of retransmissions requested(see table 4), the penalty for retransmission was too insignificant to show a significant improvement in performance (see table 3). We are currently looking into the use of the variable reliability protocol for applications which have a much higher penalty for retransmission, but can still handle some degree of loss.

7 Conclusions and Future Work

Our goal in this research is to highlight the connection between configurable protocols and adaptable applications. Our experience is consistent with the theory that given the ability to configure application protocols during runtime, applications can simplify their design and improve their performance.

	Robot 0	Robot 1	Robot 2	Robot 3
Total View	17739	15662	14177	12732
Simple Window	12555	10285	9493	7463
Window with Classes	12326	10345	10433	7413

Table 1: Number of Updates Received

	Robot 0	Robot 1	Robot 2	Robot 3
Total View	1	1	1	1
Simple Window	144	126	132	152
Window with Classes	144	126	132	152

Table 2: Number of Registration Messages Sent

	Robot 0	Robot 1	Robot 2	Robot 3
Total View	228	234	233	233
Simple Window	210	199	194	197
Window with Classes	204	197	190	191

Table 3: Total Running Time (in seconds)

	Robot 0	Robot 1	Robot 2	Robot 3
Total View	80.75	140.50	171.75	221.75
Simple Window	57.25	90.50	111.25	162.25
Window with Classes	20.75	33.50	47.50	76.75

Table 4: Number of Messages Retransmitted

Beyond these experiments, we are considering how to provide applications with feedback from the network. This information could include any statistics that can be gathered from the protocol functions, or it might be be information provided from an external monitoring source. Network feedback will enable the application to be able to make better conclusions about what type of configuration it requires.

We are also looking into the design of our intermediary communication control monitor (CCM). The purpose of this CCM would be to monitor control information from various sources and use that information to modify protocol parameters. The CCM could get control information from various sources, including the network and the application itself. Network information could include throughput levels, number of messages lost, or congestion levels. Information from the application could include future transmission rate requirements or reliability requirements.

References

[BS95] Nina Bhatti and Richard Schlichting. A system for constructing configurable high-level protocols. In *sigcomm95*, August 1995.

[Cal93] Kenneth L. Calvert. Beyond layering: Modularity considerations for protocol architectures. In *icnp93*. Georgia Institute of Technology, September 1993.

[CKK96] K.L. Calvert, R.H. Kravets, and R.D. Krupczak. An extensible end-to-end protocol and framework. Technical Report TBA, Georgia Institute of Technology, 1996.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *sigcomm90*, pages 200–208, September 1990.

[Dio95] Christope Diot. Adaptive applications and QoS guarantees. In *IEEE Multimedia Networking*, sep 1995.

[DLW94] Bert Dempsey, Jorg Liebherr, and Alfred Weaver. On retransmission-based error control for continuous media traffic in packet-switching networks. Technical Report CS 94-09, Computer Science Department, University of Virginia, 1994.

[FV90] Domenico Ferrari and Dinesh Verma. A scheme for real-time channel establishment in wide-area networks. *jsac*, 8(3):368–379, April 1990.

[GP94] R. Gopalakrishnan and Guru Parulkar. Application level protocol implementations to provide quality-of-service guarantees at endsystems. In *9th IEEE Computer Communication Conference*, 1994.

[Haa91] Zygmunt Haas. A protocol structure for high-speed communication over broadband ISDN. *IEEE Network Magazine*, pages 64–70, January 1991.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *tse*, 17(1):64–76, Jan 1991.

[LKAS93] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. An architecture and toolkit for parallel and configurable protocols. In *icnp93*, September 1993.

[MAC96] Rahmi Marasli, Paul Amer, and Phillip Conrad. Retransmission-based partially reliable services: An analytical model. In *infocom96*, 1996.

[OP92] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *tocs*, 10:110–143, May 1992.

[PPVW93] Thomas Plagemann, Bernhard Plattner, Martin Vogt, and Thomas Walter. Modules as building blocks for protocol configuration. In *icnp93*. Swiss Federal Institute of Technology Zurich, September 1993.

[SBS93] Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda. Adaptive: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, June 1993.

[ZDE+93] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, pages 8–18, Sep 1993.

[ZST93] Martina Zitterbart, Burkhard Stiller, and Ahmed N. Tantawy. A model for flexible high-performance communication subsystems. *jsac*, 11(4), May 1993.