An Architecture for Adaptable Wireless Networks

by

Sunil Kaliputnam Rao

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 18th, 2000

Author ……...
Department of Electrical Engineering and Computer Science
May 18th, 2000

Certified by……
John V. Guttag
Professor, Computer Science and Engineering
Thesis Supervisor

Accepted by ……..….
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Architecture for Adaptable Wireless Networks

by

Sunil Kaliputnam Rao

Submitted to the

Department of Electrical Engineering and Computer Science

May 18th, 2000

in partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis demonstrates the benefits of using a software based, adaptable wireless network protocol stack for voice applications. A design for a Controller Module which facilitates adaptation in the physical layer of wireless devices is presented. Using the Controller Module, designers may define a set of high level rules that govern when and what type of adaptation should take place in the physical layer. The design includes a protocol which enables Controller Modules to communicate physical layer configuration information to each other. To illustrate the use of the Controller, several Controller policies and their benefits are demonstrated.

The analysis shows that an adaptable physical layer can provide the best service to the application. In particular, voice applications are able to better react to changing channel conditions and optimize physical layer performance using application specific parameters. As wireless devices become even more pervasive, there will be an increased need for adaptable wireless systems which enable greater functionality and flexibility.

Thesis Supervisor: John Guttag
Title: Professor, Computer Science and Engineering, Head, Electrical Engineering and Computer Science Department.

Acknowledgements

I would like to thank the following individuals for their support:

- John Guttag, for his insight and advice throughout my time at the M.I.T. Laboratory for Computer Science.

- John Ankorn, for his guidance throughout my thesis research. I have learned a great deal from working with John.

- Matt Welborn and Steve Garland for their input and suggestions.

- My parents Raman and Rekha K. Rao who have provided nonstop encouragement and support.

Contents

Chapter 1

Introduction

1.1 Thesis

This thesis argues that an adaptable physical layer can provide better service for a wireless application relative to traditional, fixed physical layer configurations. Given goals and constraints from the application, an adaptable physical layer uses a set of high level rules to dynamically change its physical layer properties. This thesis presents the design, implementation, and analysis of a Controller Module that facilitates adaptation in the physical layer of wireless devices. The benefits of an adaptable physical layer are demonstrated for a voice compression application.

The Controller Module is part of the SpectrumWare [1] architecture, a software based approach to designing communication systems. The SpectrumWare approach to signal processing enables a system designer to build flexible and adaptable wireless devices. This level of flexibility is achieved by using software in place of specialized hardware throughout the protocol stack.

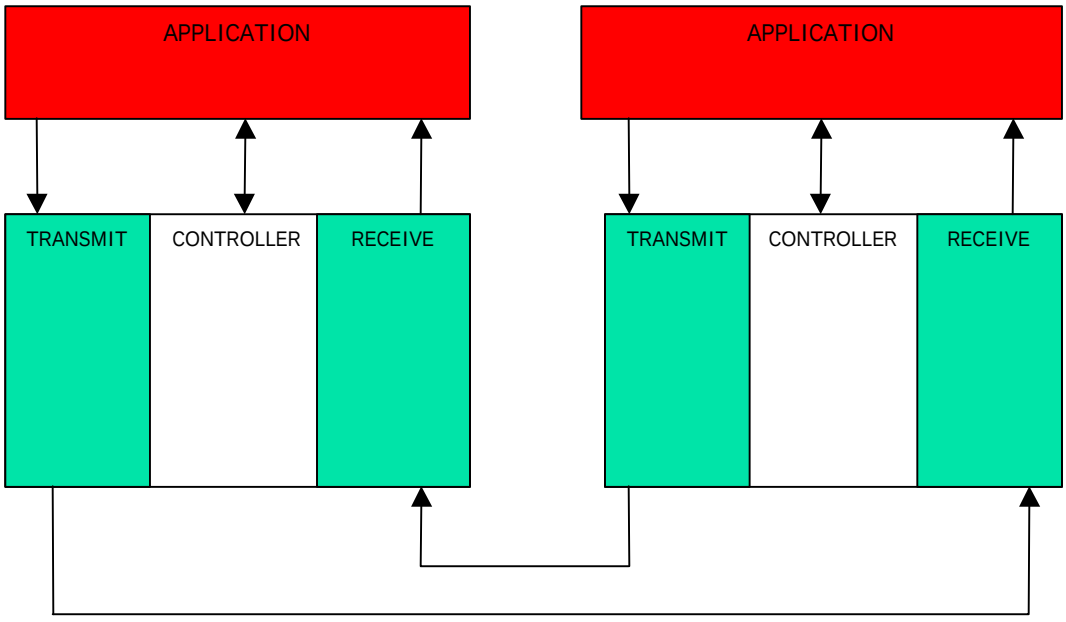

Figure 1-1. Two hosts communicating with adaptable physical layers. Given application constraints and goals, the Controller Module at each host manages adaptations to the Transmit and Receive Stack.

1.2 Adaptable Physical Layers

Since many wireless devices are implemented in hardware, they are designed to operate under worst case conditions. The channel bandwidth and power are often fixed in hardware. Additionally, the physical layer parameters such as the modulation format are generally set in hardware.

An adaptable physical layer can provide better service to a wireless application. First of all, an adaptable physical layer enables a wireless application to better react to changing channel conditions. Such a physical layer can inform the application of various parameters such as the bit error rate (BER) of the channel and the physical layer's power consumption. Secondly, an adaptable physical layer enables wireless applications to pro-actively optimize physical layer performance using some application-defined set of metrics. For example, an adaptable physical layer may be used to switch from a carrier frequency with a low signal to noise ratio (SNR) to a carrier frequency with a much higher SNR given that the BER needs to be reduced. Alternatively, the physical layer could use a different channel coder that increases the amount of error correction and thereby reduces the BER.

An adaptable physical layer also enhances the communication ability of a wireless device. Such a device can be communication protocol independent since any two devices' modulation format may be changed via a software upgrade. One could envision two wireless devices' scanning their database of modulation formats in order to find the best physical layer for the given channel.

1.3 Key Motivations

1.3.1 Higher Data Rates

With an adaptable physical layer, the data rate may be dynamically adjusted as the signal to noise ratio varies for a particular channel. An adaptable physical layer allows the modulation format to be changed as the signal to noise changes. For example, if the signal to noise ratio is very high at the receiver, then the Controller Module at the transmitting host could negotiate with the Controller Module at the receiving host to

change the modulation format to allow for a higher data rate. If the signal to noise ratio is low at the receiver, then the system could dynamically reduce the number of points in the signal constellation. Given a low signal to noise ratio at the receiver, the transmitter Controller module could also negotiate with the receiver to switch to a carrier frequency with a higher signal to noise ratio. After a successful switch to the less noisy channel, the transmitter Controller module could use a larger signal constellation to obtain a high data rate.

Each of the Controller Modules makes it's decision using input from the application. As an example, a voice compression application could use a larger data rate obtained by the use of a larger signal constellation to transmit higher quality, less compressed voice. Given that the Controller Module reacts to a lower signal to noise ratio at the receiver by decreasing the number of points in the signal constellation, the application could adjust by compressing the voice signal to a larger extent. Using this architecture, the application can both react to its environment and pro-actively request certain service from its physical layer.

1.3.2 Appropriate Source Coding and Channel Coding

An adaptable physical layer enables a communication system to dynamically choose appropriate source coding and channel coding modules for a given channel. The source coder and channel coder may be chosen independently.

The application chooses the source coder since it understands the nature of the data. With an adaptable physical layer, the application is able to make an informed choice as to which source coder to use. In particular, the Controller Module can inform the application of the latency of the channel, the data rate available, and the BER. Given these metrics, the application can determine the appropriate source coder module that introduces an acceptable amount of latency and operates within the specified data rate.

The application may also issue a request to the Controller Module to change the physical layer. For example, the application may request that the Controller Module optimize the physical layer to obtain a higher data rate. The application may either use a particular source coder given the current status of the physical layer or dynamically adjust the physical layer to accommodate a particular source coder.

While the application chooses the appropriate source coder, the Controller Module determines the best channel coder for a physical layer. The Controller Module is able to obtain the signal to noise ratio from the physical layer and may then determine the bit error rate. Given the bit error rate, the Controller Module can decide on the amount of error correction needed and pick an appropriate channel coder module to use. The Controller Module should consider the effect of the error correction overhead on the data rate. Additionally, the Controller Module must also consider the latency and computational complexity of the channel coder module that it chooses.

The Controller Module makes its decisions based upon input from the application and the Controller Rules, a component of the Controller Module which defines the rules for adaptation. For example, if the application requests a lower bit error rate from the physical layer, the Controller Module may either use more error correction, change the modulation format, change the channel altogether, or perform some combination of these operations.

With the appropriate source coder and channel coder, the application is able to obtain superior system level performance. In this manner, each module in the communication pipeline can work in concert to satisfy the application's objectives.

1.3.3 Efficient Use of Spectrum

An adaptable physical layer enables wireless devices to more efficiently use spectrum. An adaptable wireless network can transmit at a range of carrier frequencies. This enables the system to use under-utilized carrier frequencies to obtain a higher data rate. Aside from changing the carrier frequency, adaptable wireless devices can also increase their channel bandwidth given that portions of the spectrum are available. This can also provide the application with a higher data rate.

1.3.4 Power Management

With an adaptable physical layer, wireless devices can more efficiently manage their power consumption. For example, a Controller Module of the transmitter can respond to a high signal to noise ratio at the receiver by decreasing its power. Conversely, the Controller Module of the transmitter can increase the power at which it

transmits given a low signal to noise ratio at the receiver. As another example, the Controller Module of the transmitter may increase its power so that the system may obtain a lower bit error rate. The application's requests and the Controller Rules can facilitate this adaptability. Thus, while many wireless devices transmit at a fixed power, an adaptable physical layer allows for the appropriate amount of power consumption that satisfies the application's instructions.

1.3.5 Protocol Independence

One main motivation for adaptable physical layer technologies is the ability to implement new protocols on a device with just a simple software upgrade. For example, with new software an adaptable physical layer can implement a variety of medium access schemes such as code division multiple access (CDMA), frequency division multiple access (FDMA), and time division multiple access. Additionally, the software that determines the modulation format may be changed to allow for interoperability. In this manner, adaptable physical layers significantly enhance the functionality of wireless devices.

1.4 Thesis Scope

The major goal of this thesis is to develop the infrastructure in the SpectrumWare system that allows for an adaptable physical layer. The principle component that facilitates this adaptable physical layer is the Controller Module. The Controller Module has APIs to the application, the transmit stack, the receive stack. The Controller uses these APIs and a set of high level rules to govern when and what type of adaptation should take place in the physical layer. To illustrate the use of the Controller Module several Controller policies and their benefits are demonstrated.

Chapter 2

Controller Design

This chapter introduces the Controller design. The Controller Module has interfaces to the transmit stack, receive stack, and the application. Each host has a Controller Module that facilitates adaptation in its physical layer.

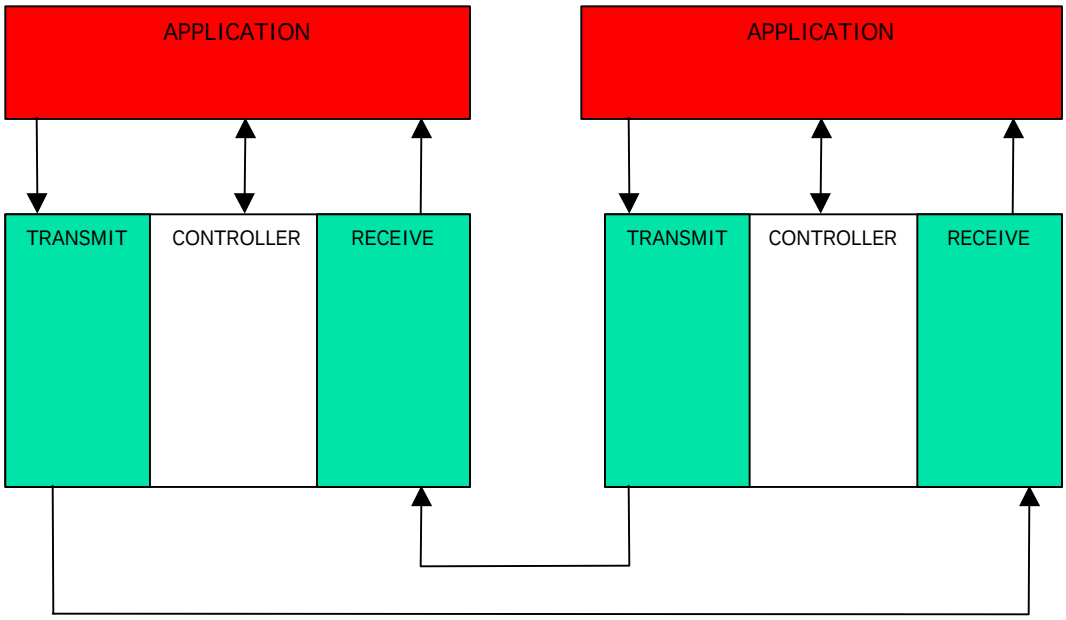

Figure 3-1. Two hosts communicating over an adaptable physical layer.

When the application is compiled, the main() program instantiates the Controller Module and passes all of the physical layer module objects to the Controller Module as arguments. The fact that the rules for adaptation are in a Controller Module and not embedded in the application itself is a novel feature of the design. This separation between the application and the application adaptation mechanism provides an effective division of labor. With this configuration, application designers do not need to

understand physical layer adaptation and may instead focus on designing systems. Application designers may make use of a particular Controller Module without investigating the particular details of the adaptation rules. On the other hand, individuals interested in application adaptation may design a set of rules suited for a particular application. With this design of the Controller, application development and application adaptation development may take place independently.

After the Controller Module is instantiated and all of the modules are passed to it as arguments, the Controller Module registers itself with each of the modules. After this registration process has occurred, each of the modules that comprise the physical layer can send indications back to the Controller Module. The Controller Module can also change various parameters in each of the modules.

.

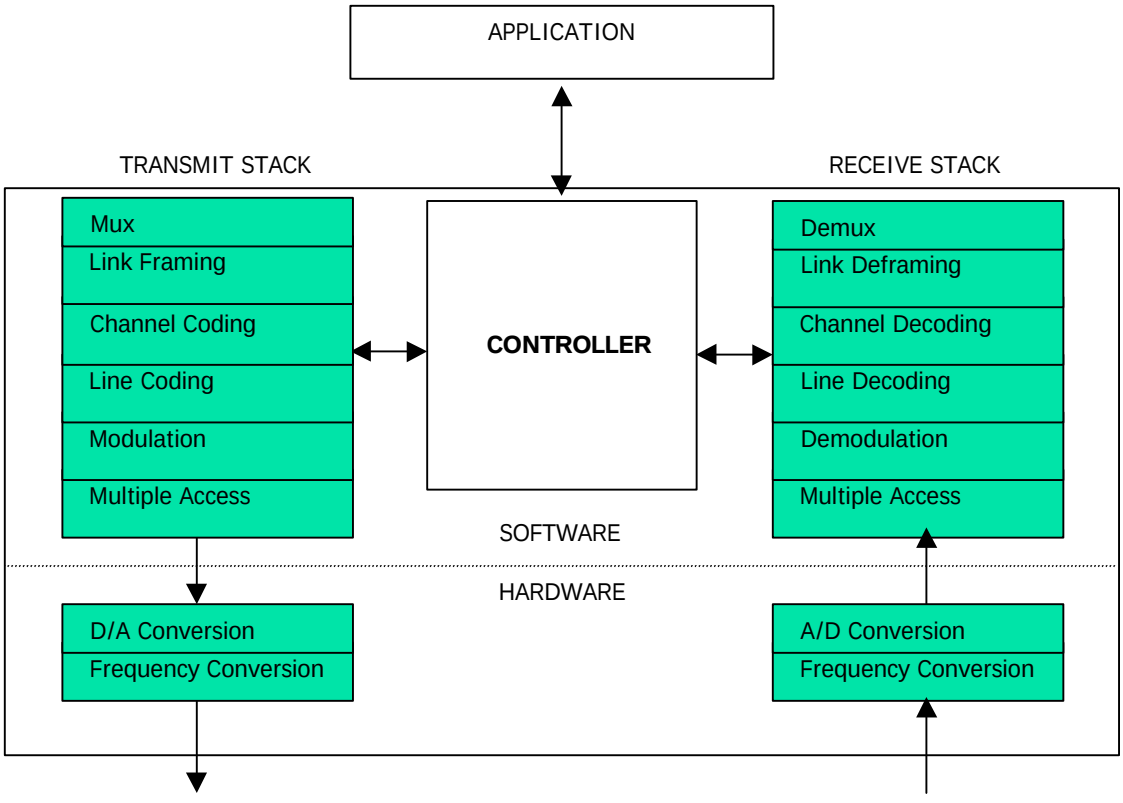

Figure 3-2. The Controller Module Architecture.

Each of the modules must maintain an **Attribute Table**. The Controller designer needs to have access to the Attribute Table of a module.

2.1 Transmit Side Controller

The Transmit Side Controller can change parameters in the Transmit Stack.

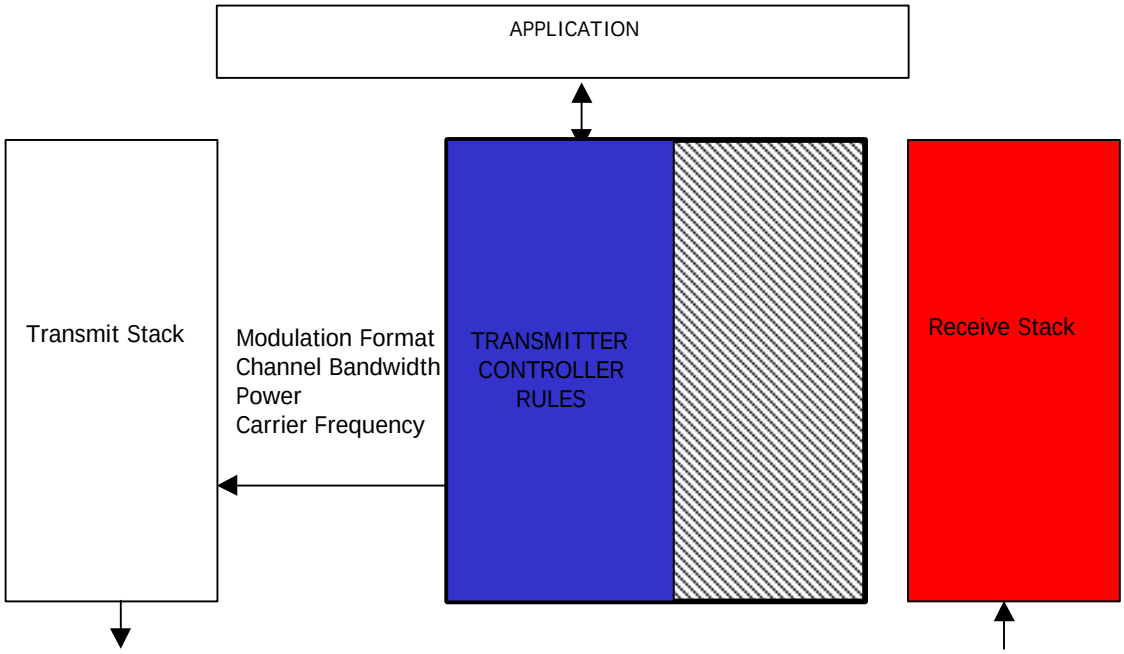

Figure 3-3. Transmit Side Controller.

The transmit side controller may modify the bandwidth, power, modulation format, or the carrier frequency.

Bandwidth - The bandwidth is defined as the width of the analog signal at the carrier frequency. The bandwidth is expressed in hertz.

Power - The power parameter denotes the amount of power used by the antenna. The power is expressed in watts.

Modulation Format – The modulation format may be any of the following: 2-PAM, 4-PAM, 8-PAM, 4-QAM, 16-QAM, 8-VSB. Additional formats could be included.

Carrier Frequency – The carrier frequency is the frequency at which the analog data is sent. This parameter is expressed in hertz.

2.2 Receive Side Controller

The Receive Side Controller is able to modify the receiver such that it may receive signals transmitted using a variety of physical layer configurations.

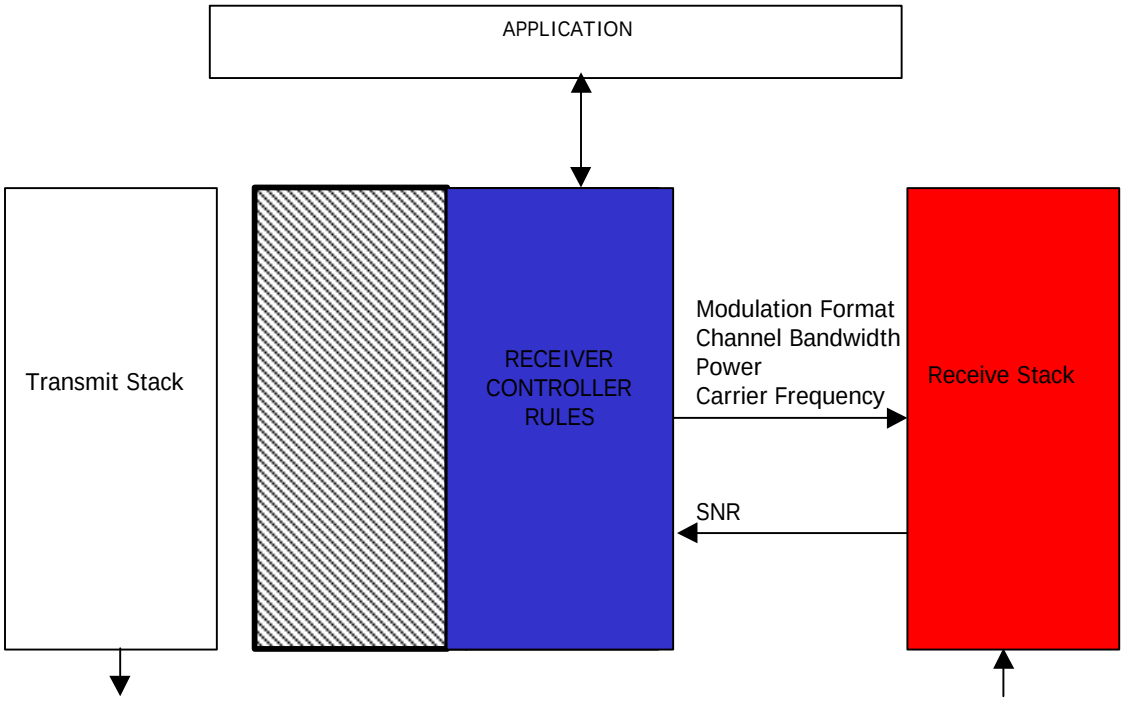

Figure 3-4. Receive Side Controller.

The Receive Side Controller can modify the modulation format, bandwidth, power and carrier frequency that it receives. The physical layer can also provide the current SNR to the Receive Side Controller.

2.3 Quality of Service Interface (QOS) to Application

The QOS interface allows an application to make requests to change the data rate, power, bit error rate, and latency of either the transmitting channel or receiving channel.

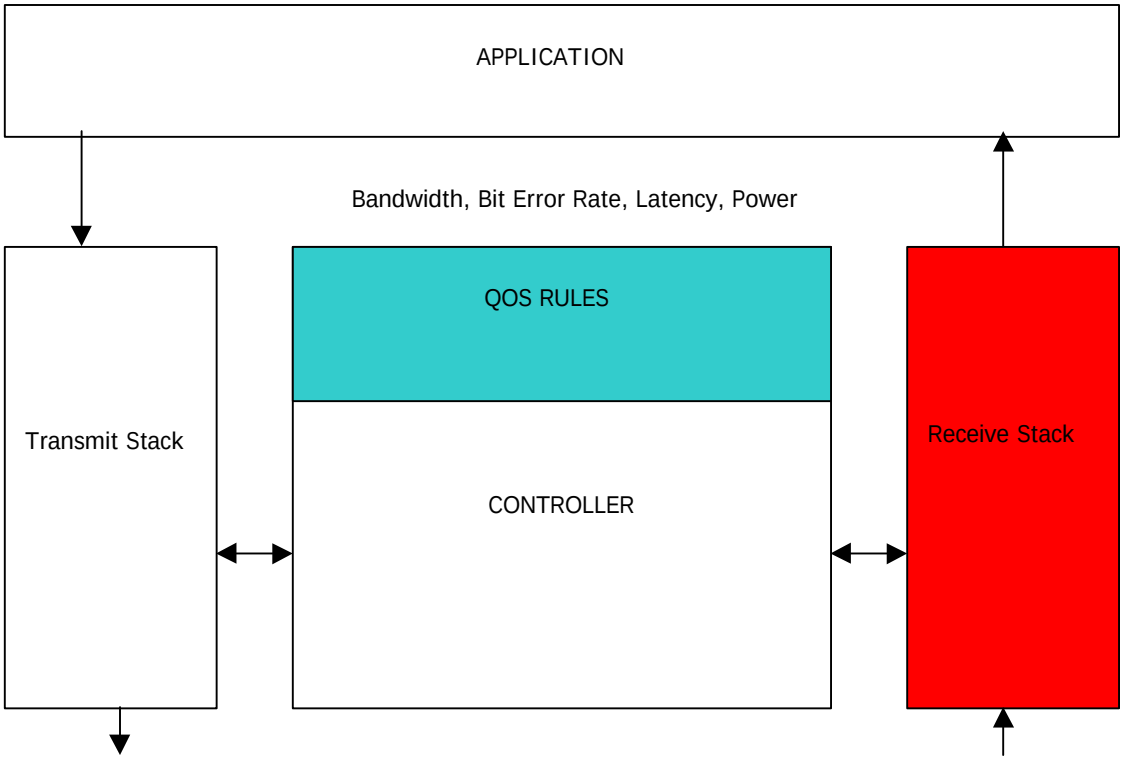

Figure 3-5. QOS Controller.

2.4 Adaptation Example

Consider an application that determines that the quality of its input data stream is unacceptable. The application could obtain the current bit error rate from its receive side Controller Module. The application might then send a resource request to the receive side Controller to have the bit error rate reduced to a particular value.

After receiving this resource request, the receive side Controller Module would then determine the physical layer parameters that need to be changed in order to achieve

a lower bit error rate. The receive side Controller Module cannot make changes to its physical layer unilaterally. The receive side Controller Module must enter into negotiations with the transmit side Controller Module to make the appropriate changes to its physical layer. Once the receive side Controller and transmit side Controller have reached a consensus, the two Controllers use their APIs to the physical layer modules to make the changes. Communication temporarily stops when the changes to the physical layer are made. The communication resumes when both Controllers have made the appropriate changes to their physical layer.

In summary, the following steps take place:

1. Application monitors QOS parameters using the Controller Module's indications API. Controller Module monitors physical layer parameters using the physical layer/controller indications API.
2. Application decides to submit a resource request to the Controller Module.
3. Controller Module uses the its set of high level rules, called the Controller Rules, to determine the physical layer changes that need to be made.
4. Controller Module enters into negotiations with its peer Controller Module.
5. Once consensus is reached, communication temporarily stops while each Controller Module makes the necessary changes to its physical layer.
6. Communication resumes.
7. Application continues to monitor QOS parameters using the Controller Module's indications API. Controller Module continues to monitor physical layer parameters using physical layer/controller indications API. The application may initiate step 2 again.

2.5 Controller Rules

The Controller Rules (CR) are a component of the Controller Module. The CR decides what changes to the physical layer need to be made in order to satisfy a particular set of resource requests from the application. The CR are the logic that govern physical layer adaptation.

The CR use five data structures:

1. An array of pointers to module objects: The CR may use this data structure to access particular modules and modify their internal parameters.
2. An array of pointers to completed physical layer indications queries: Physical layer modules may store their answers to the CR physical layer indications queries in this data structure.

3. An array of pointers to completed application indications queries. The CR stores completed application indication queries in this data structure.

4. An array of pointers to application resource request structures: The CR attempts to satisfy each of the resource requests that the application has placed in the application resource request data structure.

5. An array of pointers to negotiation data structures: The CR uses this data structure to obtain negotiation information from peer Controller Modules.

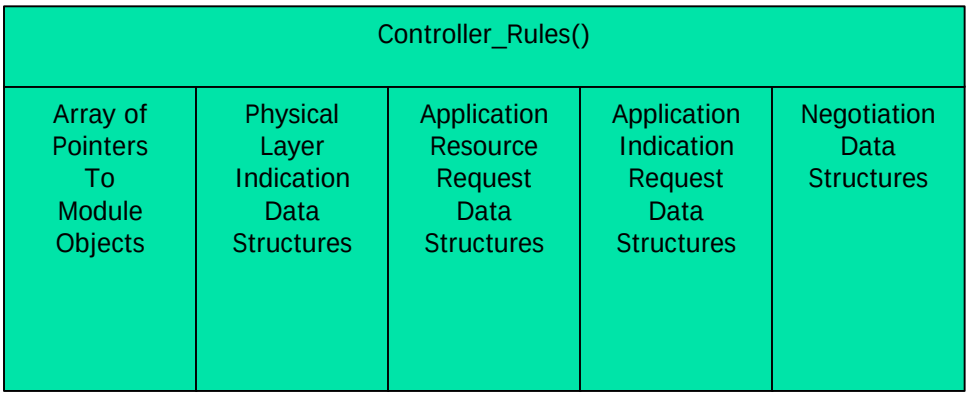

Figure 4-1. Data Structures that the Controller_Rules() method use.

2.6 Controller/Physical Layer API

The Controller uses the controller/physical layer API to change parameters in the physical layer and receive indications from the physical layer modules.

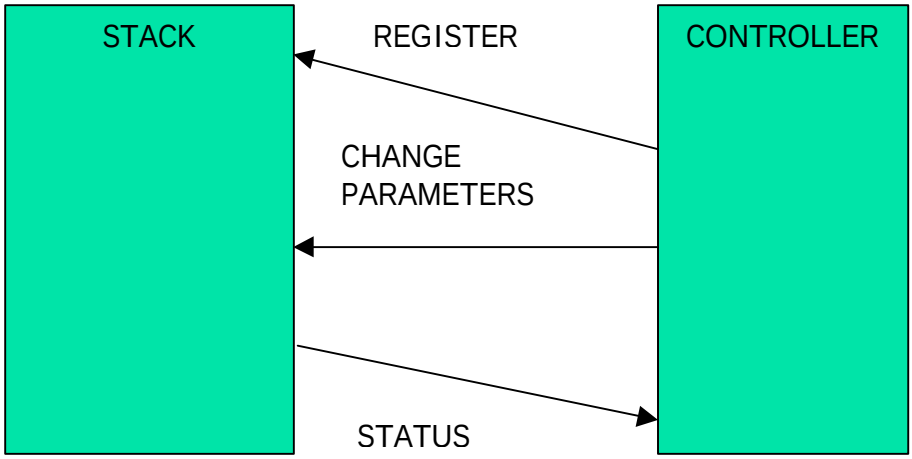

Figure 5-1. Physical layer stack and Controller interactions.

2.6.1 Startup

The main() program instantiates the Controller Module with the list of modules which comprise the physical layer.

```
// The modules are instantiated
DigitalModulator mod = new DigitalModulator(amplitude, number_of_symbols,
bandwidth, carrier_frequency);
ChannelCoder coder = new ChannelCoder(coding_rate);

// The module objects are passed to the Controller Object
Controller Controller_Module = new Controller(mod, coder);
```

The Controller Module maintains an array of pointers to these module objects. This array of pointers is termed the **Module Array**. Using the Module Array, each object can be identified through its index in the array.

2.6.2 Registration

After the Controller module has been instantiated, it registers with each of the physical layer modules.

```
Module_Array[1]->register(Controller_Module);
Module_Array[2]->register(Controller_Module);
```

Since the Controller Module registers with each of the physical layer modules, the module designer does not need to know the name of the Controller Module that may use it. The Controller Module does not make use of the objects in the main() file directly and therefore provides a layer of abstraction between the Controller Module and the main program. This abstraction provides a separation between the physical layer's adaptation rules and the physical layer itself.

2.6.3 Change Parameters

Each of the physical layer modules implements a procedure called "change". This procedure contains an **Attribute Table**, which is a mapping between an integer index and each of the parameters in the module. There is a separate Attribute Table for each module. To make the process of writing a Controller Module user friendly, a designer may use #define statements to replace the attribute index with meaningful variable names.

The change procedure is included in every module. Its specification looks as follows:

```
int change(int attribute_index, float value);
// returns 0 if successful, -1 if unsuccessful
// sets the attribute which corresponds to the attribute_index to the "value"
```

The change procedure includes a case statement that maps the attribute number inputted to the various attributes in a module.

```
// Internals of the change procedure:

case(attribute_number)
{
        0: amplitude = value;
        break;
        1: number_of_symbols = value;
        break;
        ...
        default:
        break;
}
```

The above mapping between physical layer attribute number and the attribute comprises the Attribute Table.

In the following example, the Controller designer replaces the physical layer attribute number with a name. The AMPLITUDE variable is represented by the number 0. This indicates that the zeroth element of the Attribute Table for the DigitalModulator Module refers to the amplitude. Several other #define statements are also used. The Controller designer must have access to all of the module's Attribute Tables in order to develop a naming scheme.

```
#define AMPLITUDE 0
#define NUMBER_OF_SYMBOLS 1
#define OUTPUT_SYMBOL_BITS 1
```

On the fourth line, we can see a simple Controller Rule. The statement indicates that if the signal to noise ratio passes below 10 dB the amplitude of the modulator is set to 50, the number of symbols should be fixed to 2, and the number or bits used in the channel coder should be 1.

```
if(SNR < 10)
{
                int a = mod->change(AMPLITUDE, 50);
                int b = mod->change(NUMBER_OF_SYMBOLS, 2);
                int c = coder->change(OUTPUT_SYMBOL_BITS, 1);

                if(a && b && c)
                {
                printf("success\n");
                }
                else
                {
                printf("error\n");
                }
}
```

2.6.4 Physical Layer/Controller Indications

To receive an indication from a physical layer module the Controller issues a request to the particular module. The request is a structure that identifies the module of interest, the attribute being studied, a timeout counter expressed in milliseconds, a variable in which the value of the attribute is stored, and low and high values which comprise the acceptable window of values. The module is identified by its index in the Module Array described above.

```
typedef struct {
int Module_Array_index;
int physical_attribute_number;
int timeout;
float attribute_value;
float low;
float high;
} pl_indication_request;
```

The Controller Rules set the various parameters of this structure. In this case, our pl_indication_request structure is named request_it.

```
pl_indication_request request_it;
```

The Indications procedure of the particular module is passed this structure as an argument. In this case, the 1st module in the Module_Array Indications method is being called with a particular pl_indication_request structure, request_it.

Module_Array[0]->Indications(request_it)

When the attribute falls out of the tolerable range of values in the window or when the timeout expires the structure is added to the pl_indications array of pl_indication_request structures. Next, the Controller_Rules() method is called. The Controller_Rules() method will investigate all of the data structures and can then make appropriate changes to the physical layer.

2.7 Quality of Service Interface to the Application

The point of a Controller Module is to separate the management of the physical layer from the application. A flexible system could be designed which implemented all of the protocol stack in one large level. However, the key to the SpectrumWare Controller design is that it preserves the layering abstractions that are central to networking systems. In this vein, the Controller Module maintains a simple interface to the application.

The Controller Module takes parameters that are of interest to the application. For example, the Controller Module provides the application with a data rate metric, a power metric, a bit error rate metric, and a latency metric. A designer can extend this quality of service interface for other metrics by using the conventions described.

As previously mentioned, the application is able to receive indications on these four parameters and make requests to change these four parameters. The combination of indications and requests provide the application with the tools to manage its requirements for the physical layer. The application's requests provide the Controller Rules with the direction that enables it to facilitate the adaptation of the physical layer.

2.7.1 Controller/Application Indications

Using indications from the Controller Module, the application can receive status updates on the QOS parameters. The Controller provides the application with current information regarding the latency, bit error rate, power, and data rate. To receive an indication regarding one of these parameters the application must first complete an indication request.

Each of the four factors or attributes is assigned a number. In this implementation, the following mapping was used:

```
#define data_rate 1
#define latency 2
#define power 3
#define bit_error_rate 4
```

The application_indication_request structure describes the attribute of interest, a timeout value, a variable in which the current value is stored, and low and high values which comprise the acceptable window of values. If either the timeout expires or the value of the parameter falls outside the window then the application_indication_request expires and the Controller Module calls the app_indications_handler function of the application.

To make a request the application completes an application_indication_request which is detailed below:

```
typedef struct {
int attribute_number;
int timeout;
float attribute_value;
float low;
float high;
} app_indication_request;
```

Next, the application stores the application_indication_request in the app_indication array. For example, the following structure could be defined and stored in the app_indication array:

app_indication_request check_latency;

The application can then call the Controller_Rules() method.

int status = Controller_Rules();

The Controller_Rules method will look at all of the data structures and attempt to satisfy to indication request. After the Controller_Rules completes the app_indication_request structure, it can then submit this structure to the application's app_indications_handler function:

int app_indications_handler(app_indication_request req1) {}
// returns 1 if successful, -1 if unsuccessful

Here the app_indications_handler method is called with the app_indication_request structure check_latency.
app_indications_handler(check_latency);

The app_indications_handler function must be implemented by every application.

The handler may use the structure completed by the Controller_Rules() to obtain the current value of the attribute of interest. It may then decide to submit a new indication request or it may change some application feature.

To modify an application_indication_request, an application can simply modify the relevant structure in the app_indication array and call the Controller_Rules() method. An app_indication_request may be canceled by an application by removing the structure from the app_indication array and calling the Controller_Rules() method.

2.7.2 Requests

The application can set goals for the data rate, latency, power and bit error rate. For each of these metrics, the application sets a target value, a maximum value, a minimum value, and a timeout. When the timeout expires, the resource request is terminated.

The application must first complete a resource request structure or app_resource_request. Subsequently, the application adds the app_resource_request structure to the app_resource array of structures. Next, it calls the Controller_Rules() method which attempts to satisfy all of the resource requests contained in the app_resource array. The Controller rules may then optimize the physical layer according to the application's resource request.

The resource request is captured in the following structure:

```
typedef struct {
int up_or_down;
int attribute_number;
int timeout;
float target_value;
float low;
float high;
int priority;
} app_resource_request;
```

The up_or_down parameter indicates whether the request is for the downstream (transmit) or upstream (receive) path. As described above, the data rate, latency, power, and bit error rate are each indexed by an attribute number. This structure also contains a timeout value, which indicates the number of milliseconds the resource request should be active, a target value for the parameter of interest, low and high values for the parameter which comprise a window of acceptable values, and a priority indicator.

Various priority levels schemes may be defined. The designer of the Controller Rules can determine how many priority levels are required. A possible implementation defines three different priorities:

Priority 0:
The application does not care about the request.
Priority 1:
A request by the application.
Priority 2:
A requirement for the application.
Priority 3:
A strong requirement for the application.

When optimizing the physical layer, the Controller Rules first optimize priority 3 strong requirements, then priority 2 requirements, then priority 1 requests. Priority 0 requirements are not included in the optimization process.

The application may monitor a resource request using the indications infrastructure described above.

If no physical layer optimizations are in progress or being negotiated with a peer Controller module, an application may modify or cancel its resource request. If an application attempts to modify a request during these times, the Controller_Rules() function call may fail.

To modify a request, the application simply stores a new app_resource_request in the app_resource array and calls the Controller_Rules() method. To cancel a request the application can remove the particular structure from the app_resource array and call the Controller_Rules() method.

2.8 Link Layer Communication

The Controller Rules may decide to make modifications to the physical layer given high level directions from the application. However, such modifications may not be made unilaterally. These modifications involve communicating and negotiating with a peer Controller Module.

The control messages that flow between these two Controller Modules can change the medium of communication and consequently it is extremely important that there be a reliable channel between the two Controller Modules.

In this design, the link layer communication between the two wireless hosts uses the Point to Point protocol (PPP) with high level data link coding (HDLC). PPP was

chosen because of the powerful semantics included in the specification. In particular, PPP includes a protocol that enables effective negotiation of link layer parameters. The semantics included in the PPP specification allow for new negotiation parameters to be defined. PPP was also chosen because of the fact that it considers two wireless hosts to be peers.

When two hosts communicate over PPP, they first establish their link layer connectivity. They use special Link Layer Control Packets to negotiate the link layer configuration. Subsequently, the peers use Network Layer Control Packets to negotiate the network layer protocol. At this point, standard packets may flow through the connection.

FRAME FORMAT

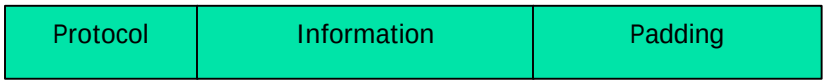

Figure 6-1. Link layer frame format.

Protocol:

The protocol field can be either a Link Control Protocol which establishes connectivity for the link layer, a Network Control Protocol which establishes the network protocol to be used, or standard IP packets and datagrams. The bit representations of these protocols are specified in RFC 1661.

Information:

This field protocol specific headers and the payload.

Padding:

Additional padding may be included in a link layer frame.

2.8.1 Link Layer Connectivity

In typical Point-To-Point connections, the link layer control protocol (LCP) is used to negotiate the data frame sizes, detect errors in the configuration, activate header compression, and terminate the link. LCP may also be used to establish authentication between two peers. This implementation extends LCP so that it allows for the negotiation of physical layer parameters for wireless devices. In particular, the carrier frequency, power, bandwidth, and modulation format may be negotiated.

Eleven types of LCP packets are defined in RFC 1661. A peer can send a LCP packet to another host to propose a link layer option. The receiver can accept or reject the options proposed. The receiver also has the ability to reject certain options and suggest alternative values. The receiver may also refuse to negotiate with the initiator of a LCP packet.

2.8.2 LCP Commands

The Link Control Protocol Packet format is depicted below:

Figure 6-2. LCP packet format

Code:
The Code indicates the type of LCP packet. Several types of LCP packets are listed below.

Identifier:
Allows the system to manage requests and replies.

Length:
Denotes the length of the LCP packet.

Data:

The LCP data is sent in the Data section of the packet. The format of the data depends on the type of LCP packet sent.

2.8.3 LCP Packet Types

Configure Request

Using a Configure Request LCP packet, the initiator can suggest various link layer parameters to the receiver. In our setup, an LCP packet which includes the carrier frequency, power, bandwidth, and modulation format can be sent to the receiver. A Configure Ack from the receiver must be received before the Restart Timer at the transmitter expires or else a subsequent Configure Request LCP packet is sent to the receiver.

Configure Ack

A Configure Ack is sent by the receiver of a Configure Request to indicate that all of the options sent by the initiator in its Configure Request are acceptable. The Configure Ack includes the original parameters sent by the Configure Request.

Configure NAK

A Configure NAK is sent when the Configure Request sent by the initiator is understandable but not entirely acceptable. The Configure NAK returns the unacceptable options in the Configure Request and also includes a list of potential values for these parameters which would be acceptable. The Configure-NAK may also include additional options that need to be negotiated.

Configure Reject

A receiver of a Configure Request sends a Configure Reject LCP packet either when it was not able to understand the original Configure Request or when some fields suggested by the Configure Request are considered by the receiver to be non-negotiable. In the latter case, the Configure Reject includes those non-negotiable configuration options in its LCP packet response.

Terminate-Request

Terminate request packets may be sent by the initiator to request the termination of a link. These packets may be sent in succession.

Terminate Ack

After a terminate ack is received the connection is considered to be closed.

Protocol-Reject

When a peer receives a PPP packet with an unknown protocol field, a Protocol-Reject LCP packet is sent to indicate to the initiator that the protocol is not supported.

Echo-Request and Echo-Reply

An Echo-Request LCP packet can help in testing the quality of the link. The peer sends back an Echo-Reply. This operation can help determine packet loss and round trip times.

Discard-Request

This command lets the sender put a packet out on the link. The receiver is supposed to silently discard the PPP packet.

2.8.4 Link Control Protocol Uses

Negotiate Physical Layer Adaptation for Wireless Devices

Given the architecture described herein, link layer control packets can be used to negotiate the carrier frequency, power, bandwidth, and modulation format for wireless devices. The LCP packets also includes the signal to noise ratio of the receiver of the communication system.

Authentication

In a configure-request a sender can specify that it is requesting authentication from the peer. The two hosts can negotiate on an authentication protocol. Authentication

of Configure Request packets is extremely important. Without authentication, attackers could send harmful Configure Request packets that initiate Controller negotiation of parameters. Such attackers could disrupt the communication between two peers.

Quality Monitoring

This configuration option can allow the peers to negotiate a protocol for link quality monitoring. In this manner, the peers can obtain link quality metrics from one another.

2.9 Negotiating Physical Layer Adaptation

2.9.1 Startup

In order for two hosts to initiate communication, they must begin to communicate over a control channel. Such a control channel defines a set of default carrier frequencies and the default modulation format, power, and bandwidth to be used. Subsequently, the Controller Modules could negotiate new physical layer parameters that would differ from the control channel.

2.9.2 Failure Cases

If the link between the peers degrades to the extent that very little data can flow, then one or both of the peers may attempt to terminate the link. PPP terminates a connection if no response is received after several retransmissions of a control packet such as a LCP or NCP packet. The Controller Module may send such control packets periodically to determine the quality of the link.

After the communication is terminated by both peers, each may attempt to initiate communication again by using the default control channel and default physical layer parameters.

2.9.3 LCP Packet Types

Each of the LCP packet types are denoted according to their LCP index number. The following table provides the mapping between the index numbers and the LCP packet types.

0 Configure Request
1 Configure Ack
2 Configure Nak
3 Configure Reject
4 Terminate Request
5 Terminate Ack
6 Protocol Reject
7 Echo Request
8 Echo Reply
9 Discard Request

2.9.4 Composition of an LCP Packet for Physical Layer Configuration Changes

The physical layer parameters are embedded in the body of the LCP packet. This thesis presents one possible allocation of bits in the payload:

FIGURE LCP PACKET

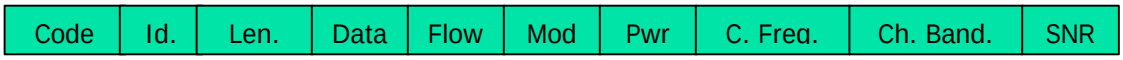

Figure 7-1. LCP packet format for wireless, physical layer negotiations.

Flow:

Eight bits are allocated to denote whether the packet refers to the upstream or downstream flow.

00000000 upstream flow
11111111 downstream flow

Mod:

Eight bits are allocated to denote the modulation format.

00000000 - BPSK
00000001 - 4-PAM
00000010 - 8-PAM
00000011 - 4-QAM
00000100 - 16-QAM
00000101 - 8-VSB
00000110 - 64-QAM

Power

Twenty four bits represent the amplitude. In this manner, any value in millivolts from zero to 33,554.431 volts may be specified.

Carrier Frequency

Forty bits represent the carrier frequency. Any value from zero to approximately 1.099e12 hertz may be expressed.

Channel Bandwidth

Thirty two bits represent the channel bandwidth. Any value in hertz from zero to approximately 4.2949 Gigahertz may be expressed.

SNR:

Sixteen bits represent the signal to noise ratio. Any value in dB from zero to 65.535 dB may be represented.

2.9.5 Negotiating

The negotiation structure specification is:

```
typedef struct {
int[][] array_of_attributes;
// this two dimensional array includes the attributes and an array
// which can include the proposed value in the case of the initiator
// or which may include a set of acceptable values in the case of the
// receiving peer
} negotiation_structure;
```

When the initiating peer's Controller_Rules desires to make a change to the physical layer, it completes the negotiation structure with the parameters to be negotiated and proposed values. The Controller_Rules method then calls the Frame_Sender() method with a LCP index number and the negotiation_structure. The Frame_Sender() method converts the negotiation structure into an LCP frame that is sent to the receiver.

When a LCP frame is received, the receiving peer's Frame_Receiver() method is called with the data frame as an argument. The frame is decoded, converted into a negotiation_structure, and a pointer to the structure is stored in the negotiation_array. The receiving peer's Controller_Rules method is then called. The receiving peer's Controller_Rules method processes the contents of the negotiation_array. At this point, the receiving peer's Controller_Rules may determine whether the proposed changes are acceptable. If the receiving peer's Controller_Rules decides to propose alternate values, it may modify the negotiation_structure using the pointer provided to it by the Frame_Receiver().

Next, the receiving peer's Controller_Rules calls the Frame_Sender() method with an LCP index number and the negotiation structure. The Frame_Sender() uses these inputs to generate a frame ready for transmission.

Negotiation may iterate until either a successful conclusion is reached or until the MAX_FAILURE counter expires.

2.9.6 Link Quality Information

The peer Controller Modules may exchange information about the quality of the link. For example, one Controller Module can send an echo-request packet to another Controller Module with its current signal to noise ratio. The Controller Module which is the receiver of this echo-request packet may use this information to gauge how much power to use in transmitting data to the sender of the echo-request packet. The receiver of an echo-request packet must submit an echo-reply with its signal to noise ratio. The "Magic Number" field contains a random number that prevents pathological conditions such as loops.

This signal to noise ratio is represented in the first 16 bits of the data portion of the LCP packet. The echo-request and echo-reply commands can also be used to estimate the round trip time between the sender and the receiver.

Figure 7-2. Echo-Request and Echo-Reply LCP packet formats.

Chapter 3

Voice Compression Application

A voice compression application was used to demonstrate the Controller Module's functionality. Voice compression is a suitable application since there are varying levels of performance, latency, and quality that can be achieved with different compression algorithms. With this application, we demonstrate that there are significant gains in voice quality that can be obtained by changing the degree of source coding in conjunction with the modulation format.

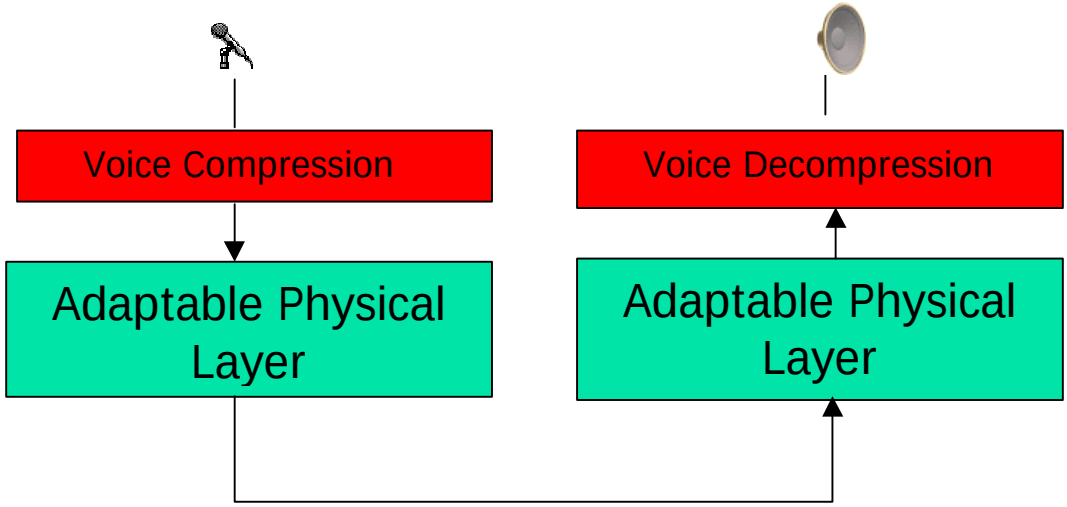

Figure 8-1. Top level diagram of voice application.

The voice application specifies its goals to the Controller Module through the quality of service interface described above. For example, the application may use the quality of service API to specify that it is trying to obtain the highest bandwidth possible. Alternatively, the application may specify that it desires a physical layer that introduces as little latency as possible.

The application receives feedback from the physical layer through its quality of service interface. For example, it may be informed that its current bandwidth has been halved. It may then compress the data to a larger extent in order to cope with the reduced

bandwidth. Under more favorable conditions, the application may compress less to deliver higher quality data. More compression often requires more computation and may result in a lower quality output.

To study the voice application, the following simulation pipeline was constructed:

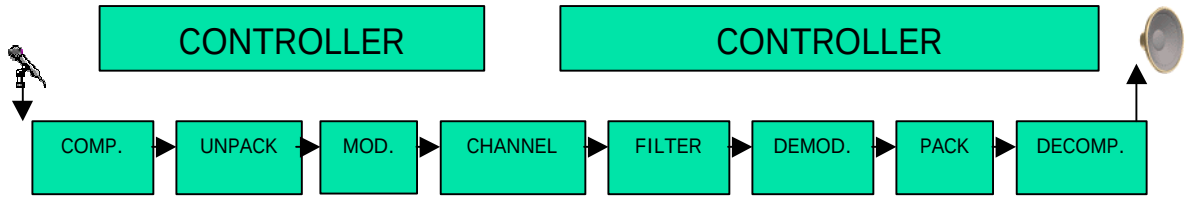

Figure 8-2. Simulation pipeline.

An audio source module (not shown) receives input from the microphone and is connected to a voice compression module. This compressed voice data is then unpacked from full data bytes into symbols that can be transmitted by the modulator. The output data from the modulator goes into a channel module that introduces additive white gaussian noise into the data stream. At the receiving system, the data stream is filtered and demodulated. Next, the symbol bytes are packed into full data bytes. This data is then decompressed and sent to an audio sink module (not shown) that sends its output to a speaker. All of the aforementioned steps are performed in software.

CHANNEL SNR:

The channel introduces additive white gaussian noise into the system. The signal to noise ratio is used to assess the quality of our channel.

Using the SNR as the figure of merit is reasonable since the SNR is directly related to the bit error rate. The symbols are chosen such that neighboring regions in the constellation only have a one bit difference. Thus, as the SNR decreases, the bit error rate increases.

Overall Signal to Error Ratio:

The signal to error ratio of the output of the sink module relative to the output of the source module is termed the "overall SER". Errors are introduced by both the voice compression modules and the channel. The compression modules introduce errors when they quantize the signal into discrete levels. The errors that the compression modules introduce are carefully designed to achieve high voice quality. On the other hand, the channel introduces random, additive white Gaussian noise. The contribution of both the errors due to noise through the channel and the errors due to the compression algorithm's quantization levels is represented in the overall SER ratio.

The overall signal to error ratio provides a possible metric for measuring voice quality. Since no standard metric for measuring voice quality exists, the overall SER metric should only be considered as one possible way for determining voice quality.

Since the symbol errors over the channel do not depend on whether the symbol forms the most significant bits of the byte being transmitted or the least significant bits of the byte being transmitted, some symbol errors will contribute more to the overall SER. Coding techniques such as unequal error protection have been shown to mitigate this decrease in overall SER.

The overall SER is an appropriate metric since the magnitude between the transmitted byte and received byte is measured. This metric makes sense for PCM code words. In PCM, the most significant bits are also the most important bits. The first bit represents the polarity, the next three bits represent the particular chord the value falls under, and the final four bits represent the sixteen steps within a chord. (Refer to Appendix A) Using a metric such as the bit error rate would not take into account the importance of the more significant bits. Therefore it would not be appropriate as a metric for measuring voice quality.

In this analysis, three different types of voice compression modules were studied:
PCM u-law, 64 kbps
ADPCM, 32 kbps
ADPCM, 16 kbps

We use 8-PAM modulation for the ulaw coder, 4-PAM modulation for the ADPCM 32kbps coder, and BPSK modulation for the 16kbps coder. Thus our test cases are:

Case 1: PCM 64kbps, 8-PAM
Case 2: ADPCM 32kbps, 4-PAM
Case 3: ADPCM 16kbps, BPSK

If we compare Case 1 and Case 2, the number of symbols transferred per second has been halved in Case 2. To offset this lower physical data rate, the amount of compression has been doubled. Thus, each of the test cases in our analysis maintains a consistent application data rate.

The graph on the following page plots the signal to noise ratio of the channel versus the overall SER ratio for each of the test cases.

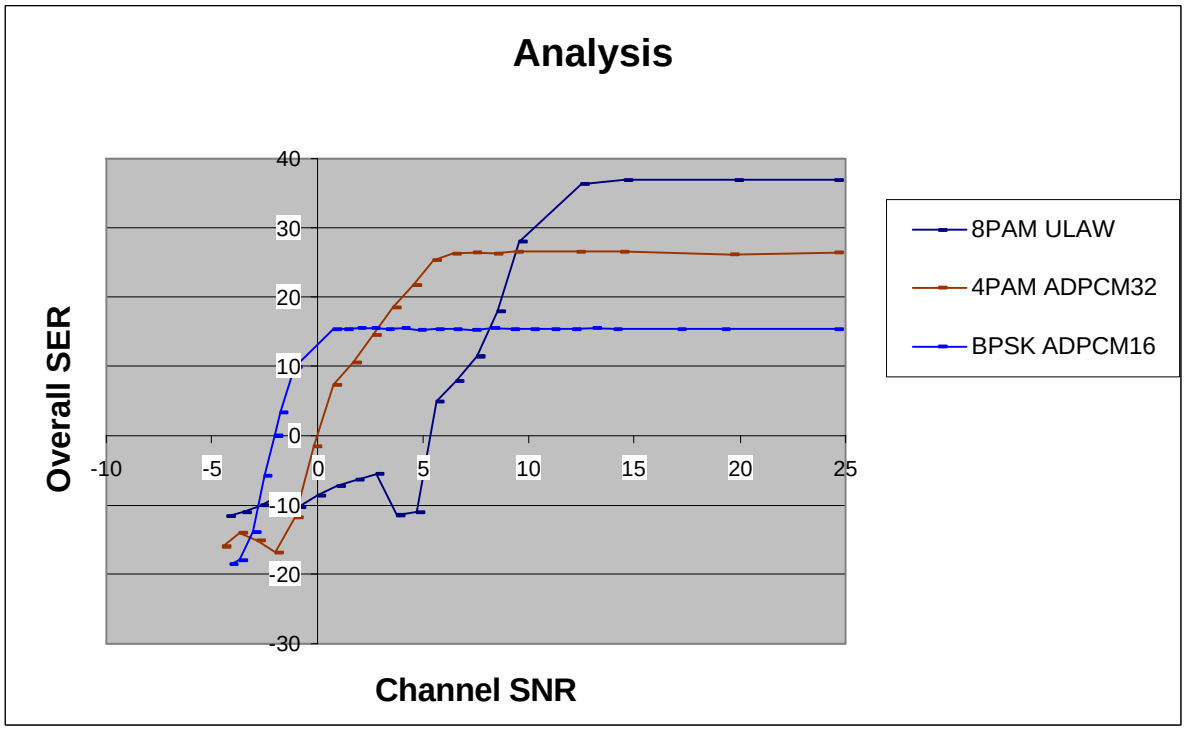

Figure 8-3. Analysis Graph.

As the channel SNR decreases, we can see that the 8-PAM, 64kbps curve begins to decline after the channel snr is less than 12. The decline occurs when noise causes a symbol in the signal constellation to be shifted to an incorrect symbol region. After the points in the signal constellation begin to fall out of their correct region due to noise, the error rate rapidly increases.

At a channel SNR of 10, the overall SER for the 8-PAM, 64kbps curve is almost the same as the 4-PAM, ADPCM32 curve. After this juncture, the 8-PAM, 64kbps curve declines rapidly. At a channel SNR of 8.5 the overall SER for the 8-PAM, 64kbps curve is around 18. To prevent the significant loss in overall SER or voice quality, the adaptive system switches to the 4-PAM, ADPCM32 curve at a channel snr of 10. If the system had not switched to the 4-PAM, ADPCM32 curve at a channel signal to noise ratio of 4.7 dB, it would have experienced a 32 dB (-150.4%) smaller overall SER. Without the adaptation, intelligible speech could not be transferred over the channel.

By switching to the 4-PAM, ADPCM32 curve, the overall SER could remain at 26.5 dB until a channel snr of 4.5 dB is reached. At this point, the overall SER declines for the 4-PAM, ADPCM32 curve as lower channel signal to noise ratios are seen.

At a channel SNR of about 2.65 the adaptive system switches to the BPSK, ADPCM16 curve. This allows the system to obtain an overall SER of 15 until a channel snr of 0.76 is encountered. At lower signal to noise ratios than 0.76, the BPSK, ADPCM16 curve also declines. It still provides the system with the highest overall SER relative to the other curves at these low channel signal to noise ratios.

As can be seen in the analysis, the adaptive wireless system can provide good service to an application by picking the modulation format and other physical layer parameters depending on the channel conditions. Other compression modules and modulation formats may provide better overall signal to error ratios at various points in the graph. Additional analysis could study how to deliver the highest voice quality by changing the compression algorithm, channel coder, and modulation format in concert under various channel conditions. This analysis shows that the ability to switch all of these physical layer characteristics allows an adaptive wireless device to deliver superior performance relative to traditional fixed schemes.

Chapter 4

Related Work

4.1 SpectrumWare Project, M.I.T. Laboratory for Computer Science

The Controller Module is a part of the SpectrumWare platform [1] [18] that enables the construction of software radios or wireless devices that perform all of their signal processing in application software. The architecture pushes the software boundary as close to the antenna as possible.

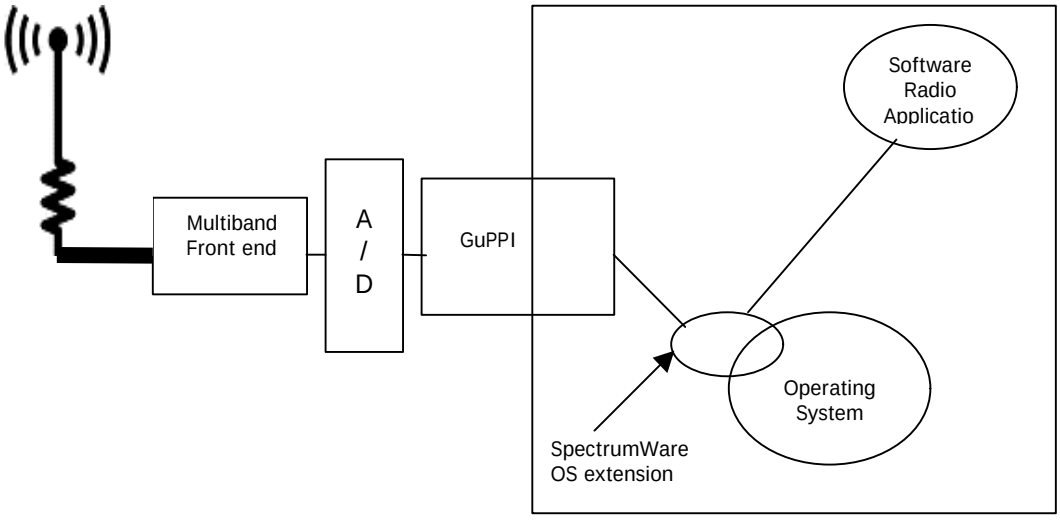

Figure 2-1. The hardware in the SpectrumWare system includes a receiver that converts a radio frequency band to intermediate frequency, analog to digital converters which digitize these samples, and a General Purpose IO card (GuPPI), which feeds these digitized samples through the computer's PCI bus into host memory. Extensions to the Linux operating system minimize costly kernel to user space transitions and ensure fast data transfer. [1]

Using SpectrumWare, functions typically implemented with specialized hardware such as digital modulation and channel coding are instead built in C++ application software. This approach provides several key advantages. First of all, since the properties of the physical layer itself are flexible, applications will be able to optimize their entire protocol stack depending on the channel conditions. For example, the

physical layer could dynamically change its modulation format or even change its carrier frequency and channel bandwidth.

The SpectrumWare system pushed the software boundary as close to the analog to digital converter as possible. In this manner, a SpectrumWare device is to a large extent transmission format independent. A device only needs a software upgrade to morph into a different wireless device.

SpectrumWare's unique data pull architecture enables efficient, real-time signal processing for a variety of applications. In the SpectrumWare system, the Sink module, the last module in a signal processing pipeline, requests a block of data from its nearest upstream module. This process continues for each module in the application up until the Source module. SpectrumWare allocates a certain amount of buffer space between each of these modules so that computation can begin.

The signal processing is not performed sample by sample since such a system would incur a high overhead because of function call processing time and would not make good use of caching effects. Rather, the signal processing is performed on blocks of data. One goal is that many of the blocks of data can be placed in the cache, and computed all at once for greater performance. After the data has been "marked" ready by each of the modules, the computation of these samples begins. In this manner, the sink determines the rate at which the entire application runs. Thus, data is computed at the rate that it is required and no faster. For example, an audio sink would be able to specify that it requires the computation of 8000 samples per second.

4.2 Odyssey Project, Carnegie Mellon University

The Odyssey project has studied application-aware adaptation, which allows mobile applications to adapt to their changing resources. The application's API to the Odyssey system allows it to make resource requests and receive indications of parameters of interest. The Odyssey system could be used to manage resources such as the bandwidth or the power of a mobile client. The Odyssey system manages resources at

the operating system level. Thus, it is not able to dynamically make physical layer modifications.

4.3 Point to Point Protocol (PPP) RFC 1661, RFC 1662, RFC 1663

These RFCs describe the details of link layer control packets including bit level descriptions of LCP packets. The details of link layer negotiations are also discussed.

4.4 Voice and Data Internetworking

This text by G. Held describes details of the pulse code modulation and adaptive pulse code modulation compression techniques. Detailed information on voice compression can be found in this resource.

4.5 A Dynamic Bandwidth Allocation Algorithm for MPEG Video Sources in Wireless Networks

This paper by Y. Iraqi and R. Boutaba describes an algorithm that enables mobile devices to provide good quality MPEG video despite varying bandwidth levels. The algorithm adjusts the level of reserved resources between the mobile device and base station so that the required quality of service is obtained.

4.6 Fundamental Challenges in Mobile Computing

This article by M. Satyanarayanan discusses the motivations for application adaptation. The article discusses several of the particular constraints involved in mobile computing and suggests that adaptation is superior to static, fixed architectures.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis demonstrates the benefits of an adaptable physical layer for wireless devices. Some of the key strengths of this architecture include the capability to obtain higher data rates through modulation format changes, optimally pick the source coder and channel coder, manage power more efficiently, use more of the available spectrum, and achieve interoperability over a wide range of protocols. The infrastructure developed in this thesis enables designers to describe adaptation for wireless devices at a high level. This thesis also details a protocol that allows Controller Modules, which facilitate the adaptation in the physical layer, to negotiate wireless physical layer parameters. The analysis illustrates the power of an adaptable physical layer for voice compression applications.

5.2 Future Work

Future work on adaptable physical layers could focus on the Controller Rules needed for a variety of applications or on new communication protocols between Controller Modules.

As an example, future work could more deeply study motivations for adaptable physical layers. Such work could detail the adaptation rules needed for dynamically choosing source coders and channel coders for a particular wireless application.

Future work could also extend the Point to Point Protocol extensions described herein to Point to Multipoint paradigms. In a Point to Multipoint topology, physical layer negotiations may be more challenging. Such work could also describe the fairness algorithms needed so that the various Controller Modules can cooperate to achieve less congestion in the network and greater utilization of the available spectrum.

Appendix A
Background on Voice Compression

A.1 Speech

Air that passes through our vocal cords and along our vocal tract creates human speech. The vocal cords are the two pairs of folds of mucous membrane that extend into the cavity of the larynx. The vocal tract is considered to be the entire pipe from the vocal cords to the mouth. Air from the lungs can open the vocal cords and this opening and closing of the vocal cords produces sounds such as human speech. The frequency and pitch of the sound can be controlled by the opening of one's mouth and the tension in one's vocal cords.

Speech can be classified into three categories: voiced sounds, unvoiced sounds, and plosive sounds.

A.1.1 Voiced Sounds

Voiced sounds are created when the vocal cords open and close due to air flow from our lungs. The frequency of the opening and closing of our vocal cords determines the pitch of the speech. Voiced sounds generally show a great deal of periodicity. The frequency of vibration of our vocal cords depends on the tension of the cords and the length of our cords.

A.1.2 Unvoiced Sounds

Unvoiced sounds occur when the vocal folds are already open. In this phase, air from our lungs passes unhindered to our vocal tract. Unvoiced sounds have a higher power level across a broader frequency range and do not exhibit much periodicity.

A.1.3 Plosive Sounds

When the vocal tract is completely closed, high air pressure typically builds up behind our vocal tract. When the vocal tract opens, significant low frequency energy results. This type of sound is termed plosive.

Some speech does strictly fall into any of these three categories but rather is a combination of one of these three types of sounds.

A.2 Requirements for Voice Applications

Voice applications have stringent timing requirements. Data that arrives significantly after 250 milliseconds generally sounds unacceptable. When compressing data, we must consider the additional voice coding delay that we introduce into our system. Since voice applications are extremely sensitive to delay, we must tradeoff compressing data further and further with the amount of time such compression takes.

A.3 Voice Compression Algorithms

There are three main voice compression techniques: waveform coding, voice coding or vocoding, and hybrid coding.

A.3.1 Waveform Coding

Waveform Coding takes advantage of the regular nature of voice signals. In waveform coding, the voice signal is sampled at the necessary intervals. The coding essentially preserves the amplitude of the sample and enables reconstruction of the original signal at the receiver.

A.3.1.1 PCM

The most common type of waveform coding is Pulse Code Modulation. Pulse Code Modulation (PCM) is utilized in standard telephone networks. The communication carriers use PCM compression to provide toll-quality voice which has a certain clarity, latency, and quality.

There are three steps performed in Pulse Code Modulation. The first step is sampling the analog waveform. A voice channel is often filtered such that it produces a passband of frequency from 300 Hz to 3300 Hz. Since no filter has an ideal cutoff, the passband may extend to 4000 Hz. According to the Nyquist rate, an analog waveform must be sampled at twice its frequency

in order to obtain reasonable reconstruction of the waveform. Thus, the analog waveform is sampled 8000 times a second which amounts to once every 125 microseconds.

Next, the analog samples must be quantized and thereby converted to discrete digital values. If 5 bits are used for quantization then a total of 2^5 or 32 quantization levels can be used to characterize the analog voice sample. Sampling at 8000 samples per second with 5 bits per sample amounts to a bandwidth of 40,000 bits per second.

Voice over an analog telephone line ranges to approximately 60 dB in power. Often times 12 bits are used in linear quantization producing a total of 2^12 quantization levels. 12 bits per sample results in a bandwidth of 96 kilobits per second. Linear quantization ensures that the distance between two quantization levels is constant. However, non-linear quantization can be effective since there is often a higher probability of lower power signals than of higher power signals in human speech.

PCM systems have also been enhanced by compressor-expandor techniques termed companding. The compressor end of the compander reduces the power range of higher power signals and increases the power range of lower power signals. Increasing the power of lower intensity signals enables the voice signal to be less affected by noise and crosstalk in a system. Decreasing the power of higher power signals, prevents these signals from creating more crosstalk in the system. With this reduced power range, only 8 bits are needed for uniform quantization.

The expandor function performs the inverse of the compressor operation. The expandor operation increases the power of the higher power signals and decreases the power of the lower power signals. The compression and expansion functions follow either the A-law standard or the u-law standard. Both of these standards use non-linear quantization. In u-law companding, the analog waveform is dividing into 255 distinct levels. The levels can be categorized into chords and steps. In each cord there are 16 steps spaced linearly. Chords are spaced logarithmically. This non-linear quantization has more granularity for lower power signals which occur with more frequency.

PCM code words are composed of a polarity bit, 3 bits which represent the chord, and 4 bits which represent the particular step with in a chord. PCM coding takes very little processing power relative to more complex hybrid coding techniques.

A.3.1.2 ADPCM

Adaptive Differential Pulse Code Modulation takes advantage of the fact that speech is regular and often repetitive. ADPCM coders make predictions of speech samples given the history of previous samples. The error between the predicted speech sample and the actual speech sample generally are less randomly distributed and hence have a lower variance. Thus, fewer quantization levels are required to encode the error.

The predictor and quantizer adapt depending on the particular speech samples at hand. The decompressor takes the quantized error value and adds it to its own prediction value to obtain a reconstructed version of the original. ADPCM does not perform extremely well given that the signals vary widely over a small range. Thus sharp cliffs such as clipping are not encoded well by ADPCM. The quantizer adapts such that the ranges between samples are more spread out if samples differ over short ranges.

Various flavors of ADPCM use 2 bits, 3 bits, 4 bits, or 5 bits for the quantization of the errors. Thus, ADPCM can generate near-toll quality voice at bandwidths of 16kbps, 24kbps, 32kbps, and 40kbps.

A.3.2 Vocoding

Vocoding involves modeling speech signals and synthesizing speech. The key characteristics of speech such as its energy, tone, pitch, are coded. The actual encoded waveform is not represented exactly. This enables vocoders to achieve very low bit rates.

A.3.3 Hybrid Coding

Hybrid coding is a combination of Waveform Coding and Vocoding. In hybrid coding, a small subset of the waveform is investigated and key parameters are obtained from this data. The hybrid coder then performs its synthesis as done in vocoding. Hybrid coding differs from vocoding in that it then compares the synthesized waveform with the original waveform. It then uses the difference between the original and synthesized signal to further tune its synthesis parameters. This process iterates until the error falls

below a threshold. Therefore hybrid coding is more computationally intensive than vocoding.

Appendix B
SpectrumWare Example: Packing/Unpacking

B.1 Purpose

Since a modulator and a demodulator work with symbols, the Packing and Unpacking modules are used to group bits into symbols and vice versa. These modules demonstrate the level of flexibility that has been designed in to the SpectrumWare system.

B.2 Unpacking

The Unpacking Module translates bits from the data stream into symbols that may be modulated. For example, a 4-Pulse Amplitude Modulated Transmitter has a constellation consisting of four different symbols or symbol regions. Each of these symbol regions is assigned a binary code.

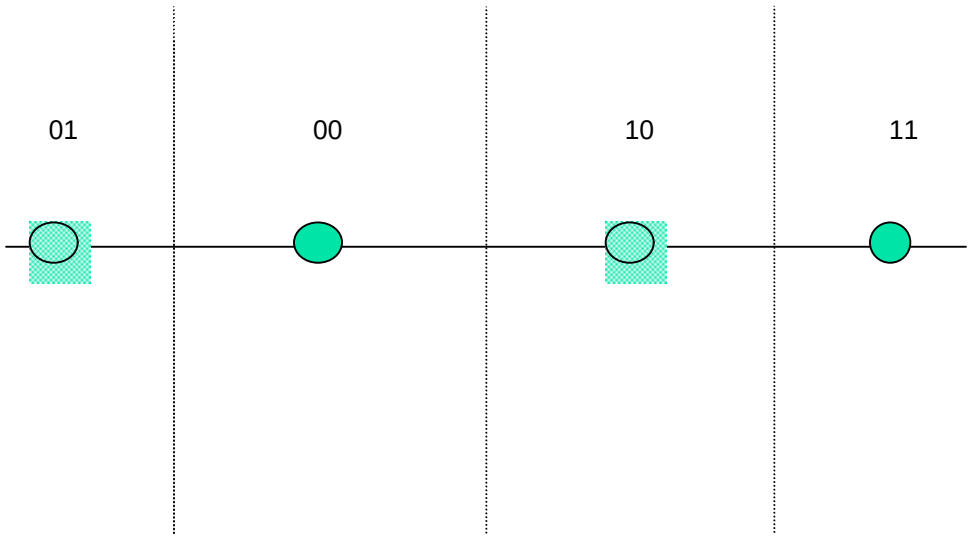

Figure B-1. 4-PAM Constellation

The symbol regions may be chosen such that errors due to noise generally only cause a single bit error.

The Unpacking Module unpacks the data from the bit stream into symbols which can be properly modulated. In the case of 4-PAM modulation, a data byte is sent in to the Unpacking Module is translated into 4 distinct data bytes each having 2 data bits from the original and 6 padded zeros.

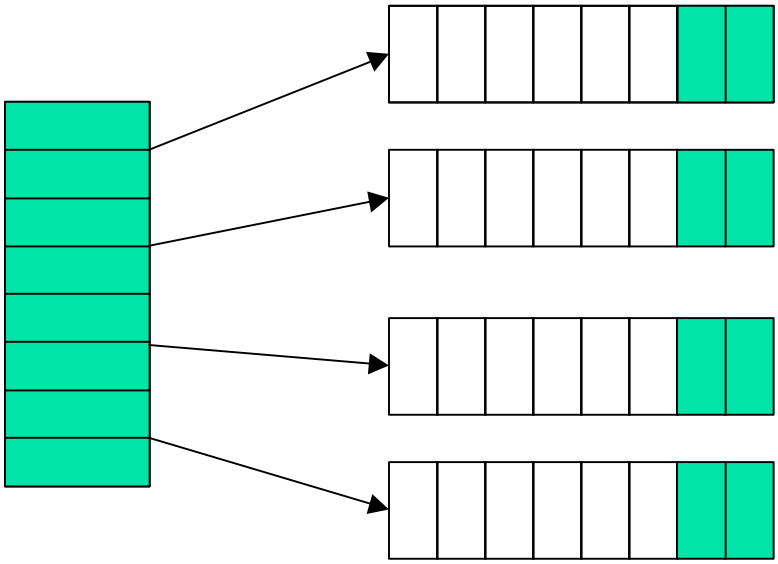

Figure B-2. One input data byte unpacked into four symbol bytes.

Each of the output bytes represents a symbol since the two lower bits have information that can be expressed in a signal constellation.

After these symbols are expressed in a signal constellation, they are modulated and transferred to the receiver.

B.3 Packing

The demodulator at the receiver translates the analog waveform it sees into points which make up a signal constellation. Using its symbol decision regions, it then translates these points in the signal constellation into bytes which contain the correct symbol. The Packing Module translates symbols outputted by the demodulator into bits.

In the case, of a 4-PAM modulated signal such bytes will only have 2 data bits expressed in them. Consequently, the symbol bytes need to be packed to obtain full data bytes.

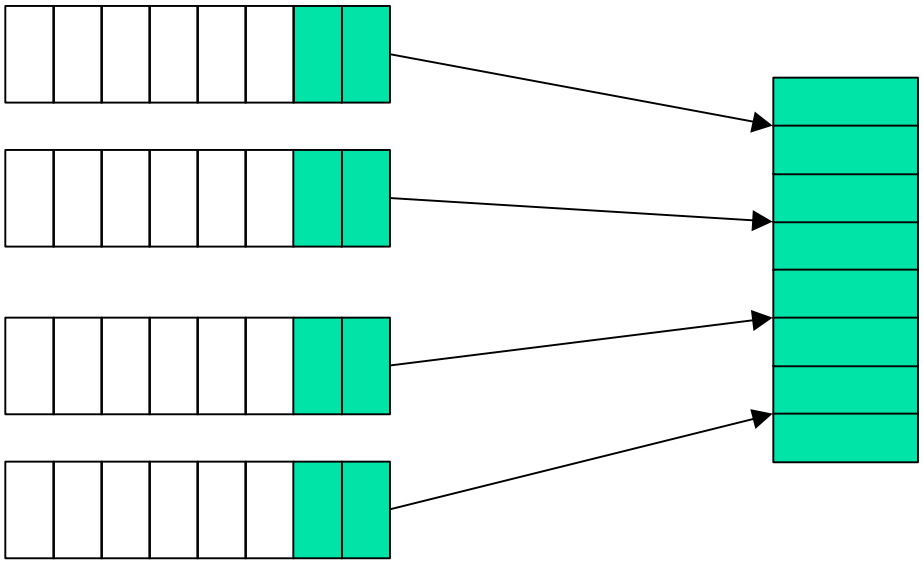

FIGURE B-3. Four symbol bytes being packed into a single data byte.

Since the SpectrumWare system allows the Controller Object to switch modulation formats, it must also possess the ability to switch the amount of unpacking and packing that takes place at the line coding module in accordance with the number of symbols used in the modulation format. The number of data bits in a symbol is log base 2 of the number of symbol regions used in the modulator or demodulator.

B.4 Ratios

In our case above, we have a ratio of 1:4. Whereby, one data byte corresponds to 4 symbol bytes. Such a ratio requires a simple Packing Module since no remainders need to be preserved. If a ratio of 1:8/3 is considered the problem of packing the data becomes more complex. With an 8 PAM modulation format, we would need to pack three data

bits into a single symbol output byte. This would lead to a ratio of 1:8/3. This suggests that one input byte produces two and 2/3 output bytes.

One approach to dealing with the remainder expressed above, would be to accumulate enough input bytes such that any fractions are eliminated. In this case, a system could accumulate 3 data bytes and output 8 symbol bytes achieving a ratio of 1:8/3. However, processing the input data in such blocks assumes a serial method of computation. SpectrumWare allows for multiple parallel threads to run on various parts of the data. The system cannot maintain the significant amount of state which informs it of the correct 3 bytes or portions of bytes that may be used to produce 8 symbol bytes. At any given point in the output stream, the system must be able to determine the correct input bytes needed for computation.

A better solution which fits into the SpectrumWare architecture assigns the forecast procedure the task of finding the appropriate point in the input stream where data should be computed and the work procedure the bookkeeping of verifying which portions of the input data are valid remainder bits.

B.5 VrUNPACK

This section describes the internals of the VrUNPACK module.

B.5.1 Forecast Procedure

The forecast procedure provides a mapping between the output sample stream and the input sample stream. A forecast procedure is given a point in the output named an output.index and is charged with finding the appropriate place in the input stream. It is also required to correlate the
size of the number of output samples requested with the size or number of input samples required. This mapping between the output sample stream and the input sample stream illustrates the data-pull architecture of SpectrumWare.

In this example, we calculate how many full bytes in the input stream have been successfully used and set the input.index to the first byte which has either been completely or partially unused. Initially, the offset and input_offset variables are zero. The output.index tells us how many symbols have been outputted. Our first task is to

determine how many full input bytes the previous outputs have used. This can be accomplished by multiplying the output.index by the number of data bits in an output and dividing by 8. We round down during these computations and thus we arrive at the first input byte that has either completely unused data or partially unused data.

When a change to the packing rate is made, we store the number of input bytes that have been used as our input_offset and we set our offset to the output.index at the time of the change. Subsequently, the current output.index minus the old output.index multiplied by the number of data bits in the symbol will tell the program how many full bytes have been written. This number is added to the input offset to arrive at the appropriate place in the input stream.

Thus, the only state that needs to be maintained when a switch of unpacking rates occurs is the output.index at the time of the switch denoted by the "offset" variable and the input.index at the time of the switch denoted by the "input offset".

```
template<class iType,class oType> int
VrUnPACK<iType,oType>::forecast(VrSampleRange output, VrSampleRange inputs[])
{
// body

if(change_request == 1)
  {
                min_output_bits = new_min_output_bits;
                offset = output.index;
                change_request = 0;
                input_offset = beginning_next_input + input_offset;
}

inputs[0].index= ((output.index - offset) * min_output_bits)/8 + input_offset;
inputs[0].size= (output.size * min_output_bits)/8 + 2;

beginning_next_input = inputs[0].index + (output.size * min_output_bits)/8;

return 0;
}
```

B.5.2 Work Procedure

The work procedure is presented with the appropriate place in the input stream and the corresponding point in the output stream. The work procedure is charged with

collecting data from the input, computing with this data, and setting the results equal to the appropriate point in the output data stream.

For most applications, the entire byte of the input data is valid. In our example, only part of an input byte may be unused data. To determine the portion of the byte that has valid, unused data the work procedure determines the number of bits that have been used since the last packing ratio change to generate output bytes. This number is the quantity of the output.index minus the offset times the number of bits used in each output byte. The remainder can be determined by taking this number modulus 8. This remainder represents the number of used bits in the byte at hand. The number of unused bits is simple 8 minus the number of used bits.

The implementation above maintains two counters: a valid bits and a ready bits. When an input byte is accepted, the number of valid bits is set to 8. Each of the 8 bits is shifted one by one on to an output byte. Each time a bit is shifted onto an output byte the number of valid bits is reduced by one and the number of ready bits is increased by one. When the number of ready bits equals the number of data bits that should be included in an output symbol byte, the output byte is mapped to the output stream and the ready bits counter is set back to zero. When the number of valid bits is zero, another input is taken leading to 8 more valid bits. This process continues until the size of the output buffer is reduced to zero.

Given a byte with used bits, the system initializes the number of ready bits to the negative of the number of used bits. In this manner, bits may be shifted on to an output byte but the output byte will not be mapped to the output stream until the number of ready bits equals the exact number of bits needed to represent a symbol byte.

```
template<class iType,class oType>
int VrUnpack<iType,oType>::work(VrSampleRange output, void *ao[],
                                VrSampleRange inputs[], void *ai[])
{
  iType **i = (iType **)ai;
  oType **o = (oType **)ao;
  int size = output.size;
  int remainder = ((output.index - offset) * min_output_bits) % 8;
  num_outputted_bits = remainder;
  ready_bits = -remainder;
```

```
  u_char out = 0x00;
  u_char new_bit;
  u_char current_data;

  while(size > 0) {
    current_data = *i[0]++;
    valid_bits = 8;
    while(valid_bits > 0)
                    {
                      new_bit = (char) (current_data >> (valid_bits - 1)) & 0x01;
                      out = (out<<1) + new_bit;
                      valid_bits--;
                      ready_bits = ready_bits + 1;
                      if(ready_bits == min_output_bits)
                        {
                          // Zero the top bits of the output
                          out = out << (8 - min_output_bits);
                          out = out >> (8 - min_output_bits);
                          *o[0]++ = out;
                          out = 0x00;
                          num_outputted_bits = num_outputted_bits + min_output_bits;
                          ready_bits = 0;
                          size--;
                          if(size <= 0)
                               {
                               break;
                            }
                       }
          }
                        beginning_next_input = beginning_next_input + (num_outputted_bits
/ 8);
                       return output.size;
}
```

B.6 Example

As an example, assume an initial bit packing ratio of 1:8/2 for the unpacking procedure. Two bits from each data byte form an output symbol byte. Consequently, four output symbol bytes fully use up the data in one input byte. Thus, when the forecast procedure is given the output index of 3 (the index starts at zero) it will map this to byte zero of the input stream. In our example, the bit packing ratio remains at 1:8/2 until byte 7 in the output stream.

After byte 7 in the output stream, the bit packing ratio is changed to 1:8/3. According to the forecast algorithm above, the input index of 2 is stored in the input_offset variable and the output index of 8 is stored in the offset variable. This ratio means that three data bits are needed to form one symbol byte. If the forecast procedure were given an output index of 8, it would map this to an input index of 2. Only three bits of the 3rd input data byte will then be used.

Now consider an output index of 10. This maps to an input index of 2 also since there are 2 unused input data bits in the third input byte. The work procedure may now determine which of these bits are unused. It accomplishes this by understanding how many bits have been written at the current rate. In our case, (10 - 8) or 2 output symbol bytes have been written at the current rate. This amounts to 2 * 3 or 6 data bits. Thus, 6 data bits in the third data byte have been used. This leads us to conclude that the last two bits of the point in the data stream with an input index of 2 are valid data. The work procedure is then able to continue adding valid data bytes on to its buffer for use in developing symbol bytes.

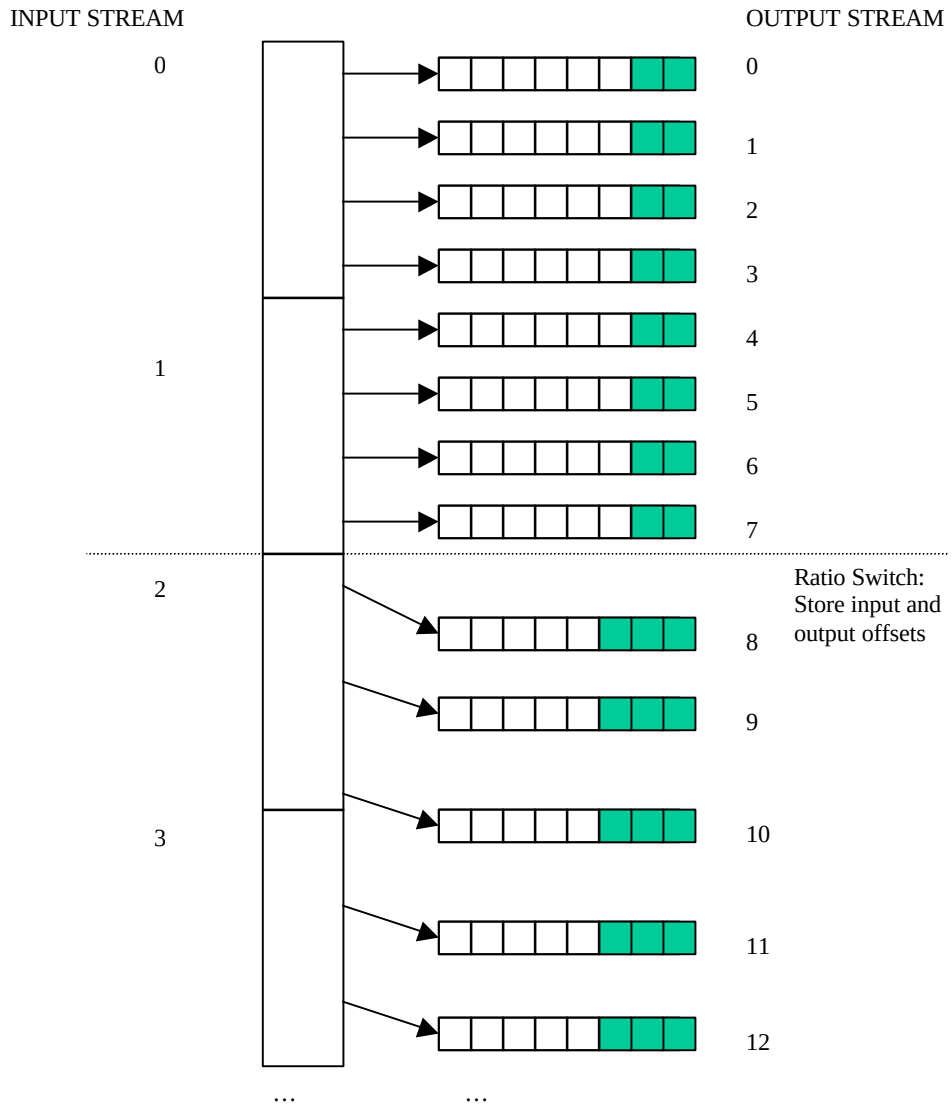

Figure B-4. Example scenario of unpacking ratio switch.

B.7 VrPACK

The packing routine is used by the receiver to translate symbols into bits. This module allows any ratio of packing. This means that some portion of a symbol may be used in one data byte and the remaining portion may be used in a separate data byte.

B.7.1 Forecast Procedure

The forecast routine maps the input sample stream which are the symbol bytes to the output sample stream which are the full data bytes.

```
template<class iType,class oType> int
VrPACK<iType,oType>::forecast(VrSampleRange output,
                              VrSampleRange inputs[]) {

   if(receive_coder_change_ok == 1)
   {
    receive_coder_change_ok = 0;
    min_input_bits = changed_bits;
    offset = output.index;
    change_request = 0;
    input_offset = beginning_next_input;
   }

  inputs[0].index= ((output.index - offset) * 8)/min_input_bits + input_offset;
  inputs[0].size= (output.size*8)/min_input_bits + 2;

  beginning_next_input = inputs[0].index + (output.size*8)/min_input_bits;
          return 0;
}
```

The forecast function is provided with the output index and output size and is charged with mapping these parameters to the input index and input size. With the offset and input_offset parameters initially equal to zero, the algorithm multiplies the output index which represents the number of full data bytes by 8. This produces the total number of output data bits. This total number of output data bits is divided by the number of bits represented in each symbol byte. This operation yields the number of symbols it takes to produce an output index number of output data bytes. Any remainders will be zeroed by the integer division calculation. Thus, we will arrive at the first symbol byte with either a fully unused set of bits or a partially unused set of bits.

If a change is made to the packing rate, then the input index at which this change occurs is stored in the input_offset and the output index at which this change occurred is stored in the offset variable. In this manner, further advances in the input index at the new packing rate can be accurately found relative to its starting point while the prior advances in the input index are captured by the input_offset parameter.

B.7.2 Work Procedure

The work procedure is provided with an input sample stream and an output sample stream. The input sample stream is composed of symbol bytes. The output sample stream is composed of full data bytes.

The work procedure must first determine how many bits of the first input symbol byte are unused. It does a similar calculation as that performed above. It multiplies the output index by eight to determine the total number of data bits completed. It then computes the total number of data bits modulus the number of bits in each symbol byte. This result is the **remainder** or the number of bits of the last symbol byte that have been used. The work procedure may use the remainder to isolate the new, unused bits in any particular symbol.

In this implementation, we continue to place unused bits from the symbol bytes onto a buffer until we have accumulated over 8 unused bits. The number of bits placed on the buffer is denoted with the valid_bits variable. After we have accumulated over 8 valid bits, we shift one of these valid bits on to our output byte iteratively. Each time we add another bit from the buffer to our output byte the ready_bits variable is incremented. When the ready_bits variable equals eight bits we map our output byte to the output sample stream and set the ready_bits parameter to zero. This process continues until the size of our output buffer equals zero.

To exclude used bits from the output, we can set the ready bits equal to negative one times the remainder. In this way, the ready_bits parameter will only be triggered when these bits have been shifted out and eight, fresh bits have been added to the output byte.

```
template<class iType,class oType> int
VrSymbols2Bits<iType,oType>::work(VrSampleRange output, void *ao[],
```

```
                                VrSampleRange inputs[], void *ai[])
{

 iType **i = (iType **)ai;
 oType **o = (oType **)ao;

 int size = output.size;
 int remainder = ((output.index - offset) * 8) % min_input_bits;
 num_outputted_bits = remainder;
          ready_bits = -1 * remainder;
 valid_bits = 0;
 u_char out;
 out = 0x00;
 u_char new_bit;
 u_char current_data;
 u_char temp;

 while(size > 0) {
  while(valid_bits < 8)
    {
                current_data = *i[0]++;
                buffer = buffer << min_input_bits;
                temp = current_data << (8 - min_input_bits);
                temp = temp >> (8 - min_input_bits);
                buffer = buffer + temp;
                valid_bits = valid_bits + min_input_bits;
            }
          while(valid_bits > 0)
                {
                new_bit = (char) (buffer >> (valid_bits - 1)) & 0x0001;
                out = (out<<1) + new_bit;
                valid_bits--;
                ready_bits++;
                if(ready_bits == 8)
                  {
                    *o[0]++ = out;
                    ready_bits = 0;
                    out = 0x00;
                    size--;
                    num_outputted_bits = num_outputted_bits + 8;

                    if(size <= 0)
                      {
                           break;
                      }
                  }
```

```
                }
  }
  beginning_next_input = beginning_next_input + (num_outputted_bits / min_input_bits);
  return output.size;
}
```

B.8 Summary of Packing/Unpacking Modules

Given such a setup, one is able to seamlessly switch the bit packing ratio. Multiple changes of the bit packing/unpacking ratio may occur and the only state needed is the input offset which represents the point in the input stream at which the switch occurred and the output.index value which represents the point in the output stream at which the switch occurred.

References

[1] V. Bose, M. Ismert, M. Welborn, J. Guttag, "Virtual Radios", JSAC issue on software radios, 1998.

[2] B. Noble, M. Price, M. Satyanarayanan, "A Programming Interface for Application-Aware Adaptation in Mobile Computing", Second USENIX Symposium on Mobile And Location-Independent Computing, 1995.

[3] J. Proakis and M. Salehi, "Communication Systems Engineering", Prentice Hall, Upper Saddle River, NJ, 1994.

[4] V. Bose, A. Shah, M. Ismert, "Software Radios for Wireless Networking", Infocomm 1998.

[5] L. Peterson and B. Davie, "Computer Networks: A Systems Approach", Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[6] Y. Youssef and R. Boutaba, "A Dynamic Bandwidth Allocation Algorithm for MPEG Video Sources in Wireless Networks," ACM, 1999.

[7] G. Held, "Voice and Data Internetworking", McGraw-Hill, New York, NY, second edition, 1998.

[8] V. Bose, A. Chiu, D. Tennenhouse, "Virtual Sample Processing: Extending the Reach of Multimedia", Multimedia Tools and Applications, Volume 5, No. 3.

[9] K. Lee, "Adaptive Network Support for Mobile Multimedia", Mobicom 1995.

[10] W. Simpson, RFC 1661, "The Point-to-Point Protocol (PPP)", Networking Working Group, 1994.

[11] W. Simpson, RFC 1662, "PPP in HDLC-like Framing", Networking Working Group, 1994.

[12] D. Rand, RFC 1663, "PPP Reliable Transmission", Networking Working Group, 1994.

[13] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications", ACM, 1999.

[14] A. Chiu, "Adaptive Channels for Wireless Networks", Master's Thesis, MIT, June 1999.

[15] B. Noble, M. Satyanarayanan, D. Narayanan, et. al., "Agile Application-Aware Adaptation for Mobility", 16th ACM SOSP.

[16] J. Woodard, "Speech Coding", http://www-mobile.ecs.soton.ac.uk/speech_codecs

[17] A. Duel-Hallen, S. Hu, H. Hallen, "Long-Range Prediction of Fading Signals: Enabling Adapting Transmission for Mobile Radio Channels", IEEE Signal Processing, Vol. 17, No. 3, May 2000

[18] B. Vasconcellos, "Parallel Signal Processing for Everyone", Master's Thesis, MIT, December 1999.

[19] M. Satyanarayan, "Fundamental Challenges in Mobile Computing", Carnegie Mellon University