IMPLEMENTATION OF AN 802.1X SUPPLICANT FOR INTERNET TELEPHONY

by

Jerry Wong

B.A.Sc., Simon Fraser University, 2000

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING

IN THE SCHOOL OF ENGINEERING SCIENCE

Simon Fraser University

Spring 2005

Approval

Name: Jerry Wong

Degree: Master of Engineering

Title of Project: Implementation of an 802.1X Supplicant for Internet Telephony

Examining Committee:

Chair: Dr. Dong In Kim

Senior Supervisor:

Dr. Jim Cavers
Professor
School of Engineering Science, SFU

Technical Supervisor:

Mr. Andy Fung
Engineering Manager
Broadcom Canada Ltd.

Date Approved: April 14, 2005

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENCE

Abstract

To prevent unauthorized devices from accessing private Local Area Networks (LANs), the IEEE 802.1X standard combined with the related Internet Engineering Task Force (IETF) RFC specification, the Extensible Authentication Protocol (EAP), provide an access control mechanism for IEEE 802 LANs. EAP-enabled networks allow administrators to ensure that devices such as PCs and IP phones are authorized and authenticated to access the enterprise's LAN environment. These devices are also known as supplicants in 802.1X terminology.

This is a report on the implementation of an 802.1X supplicant for the *BCM91101* reference IP phone platform at *Broadcom Canada Ltd.* This report includes a study of the 802.1X framework and the EAP, as well as a review of the supplicant software development process including design, implementation and testing. The test results indicate that the implemented supplicant software consumes *983kB* with the EAP-TLS method and *61kB* with the EAP-MD5 method, with negligible CPU usage.

Acknowledgements

First of all, I would like to thank Mr. Andy Fung for letting me to work on this exciting and challenging project, giving me all levels of freedom from design to implementation to testing. I also wish to thank Dr. Jim Cavers being my academic supervisor for this project and for giving his time and energy to review this work. I am grateful to Mr. Bob Lukas for inspiring me to work on the protocol field. I would like to thank my wife, Winnie, for her support and understanding. I want to thank my parents for their patience and encouragement on my academic studies. Last but not least, I would like to thank my friends, especially Nelson and Bennett, for their support.

Table of Contents

List of Figures

List of Tables

Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
DHCP	Dynamic Host Control Protocol
DNS	Domain Name Service
DSP	Digital Signal Processing
EAP	Extensible Authentication Protocol
EAPOL	Extensible Authentication Protocol over Local Area Network
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPSec	IP Security
LAN	Local Area Network
MAC	Media Access Control
NVRAM	Non-volatile Random Access Memory
OS	Operating System
PAE	Port Access Entity
PEAP	Protected Extensible Authentication Protocol
PEM	Privacy Enhanced Mail
PKI	Public Key Infrastructure
QoS	Quality of Service
RADIUS	Authentication Dial-In User Service
RFC	Request For Comments
RTOS	Real Time Operating System
SDRAM	Synchronous Dynamic Random Access Memory
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TLS	Transport Layer Socket
TTLS	Tunneled Transport Layer Socket
VoIP	Voice over Internet Protocol

Chapter 1: Introduction

In recent years, network security has become a major issue for network administrators. They are increasingly concerned about security threats such as virus attacks, worms and undesirable network activities such as music file sharing. There are also security breaches that either originated or aided by internal users putting corporate data into jeopardy. Both of these make network access control one of the most important means to defend a network.

The IEEE 802.1X standard combined with the related Internet Engineering Task Force (IETF) RFC specification, the Extensible Authentication Protocol (EAP), provides an access control mechanism for IEEE 802 Local Area Networks (LANs). EAP-enabled networks allow administrators to ensure that individual users are authorized to access the enterprise's LAN environment. 802.1X is particularly beneficial in a campus type network where users connect from random locations or where guests will access the network; however, it is becoming more prevalent in mainstream enterprises along with many other security initiatives.

1.1 The 802.1X Framework

The IEEE 802.1X standard specifies an architectural framework for authentication of devices attached to ports on IEEE 802 LANs such as the 802.3 for wired Ethernet, and the 802.11 for wireless Ethernet [1]. It also defines requirements for the protocol exchange between the entities involved in the framework. The IETF RFC 2284 [2] defines the EAP that fulfills the protocol requirements defined by 802.1X. The EAP is a generic protocol that supports multiple authentication methods such as the MD5 message-digest (EAP-MD5) [3] and the Transport Layer Security (EAP-TLS) [4] authentication methods.

The 802.1X framework defines three roles: a supplicant, an authenticator, and an authentication server. The supplicant is an entity, such as a PC or an IP phone, which

requires access to the network for use of its services. The authenticator is a network entry point that the supplicant physically or logically connects, typically a managed switch for Ethernet or a wireless access point for wireless 802.11, and acts as a proxy between the supplicant and the authentication server. The authenticator is responsible for controlling port access of the supplicant(s) based on the grant status from the authentication server. The authentication server, typically a Remote Authentication Dial-In User Service (RADIUS) server, performs the actual authentication, allowing or denying access to the network based on the supplied credentials from the supplicant. The 802.1X framework is depicted in Figure 1.

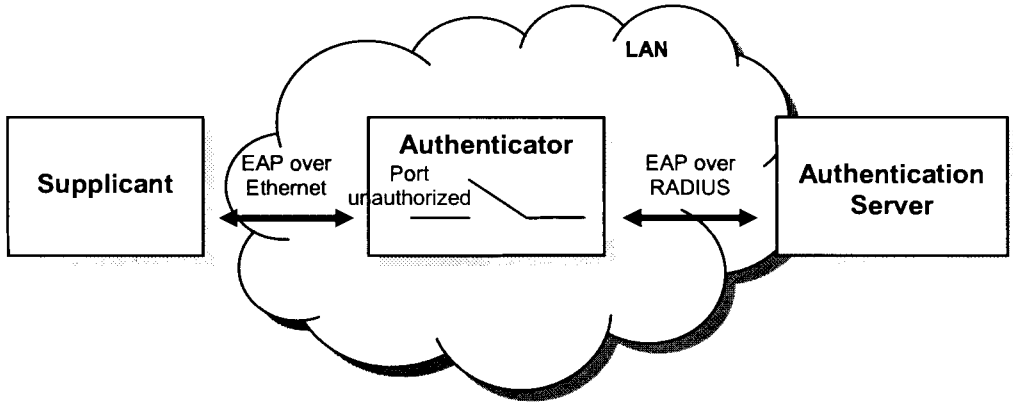

Figure 1: 802.1X framework

1.2 IP Telephony

As the name implies, IP telephony is a term that is used to describe the transport of telephone calls over the Internet, and IP phones are devices that transmit voice over the IP network. The voice over IP technology has been an emerging market as of late in both the residential and enterprise markets as it promotes data convergence. Unlike traditional enterprise phones which have separate wirings from the data network, IP phones can be operated in the same network within an enterprise, hence reducing the wiring cost of installing and running a phone system. IP telephony also promotes the support of enhanced call features such as video conferencing, web casting, etc, that is otherwise unachievable with traditional phones.

In light of this market, the *Broadcom BCM1101* silicon was developed for enterprise IP phones. This single-chip solution integrates a *150MHz MIPS* processor, a *108MHz* ZSP DSP processor, a two-port *100baseT* Ethernet switch, and controllers for peripherals such as SDRAM, keypad, etc. The *BCM91101* is a reference platform for the *BCM1101* silicon that serves as a proof of concept to potential customers and as well as a software development platform for developers.

The *MIPS* processor software contains device drivers, network services and call-signaling software, and is responsible for managing voice calls while the DSP software manages the voice codec algorithms. New features are implemented based on marketing needs. As security initiatives such as access control becoming more prevalent, it becomes natural that 802.1X should be supported for the reference software.

The objective of this project is to implement an 802.1X supplicant on the *BCM91101* reference IP phone platform, supporting the EAP-MD5 and EAP-TLS authentication methods.

1.3 Document Outline

This document describes the design, implementation and testing of the 802.1X supplicant software. Chapter 2 provides the background information for the 802.1X standard and an overview of the IP phone software architecture. Chapter 3 describes the design and implementation of the supplicant software integrated to the IP phone software. Chapter 4 describes the test plan and presents the test results. Finally, Chapter 5 concludes the report by summarizing the supplicant software implementation and the test results.

Chapter 2: Background

This chapter provides the background information for the project and is organized into two sections. The 802.1X protocol along with the EAP would be described and explained in Section 2.1. The IP phone software architecture would then be explored in Section 2.2.

2.1 802.1X and EAP

The 802.1X standard specifies encapsulation methods for transmitting EAP messages over different 802 media types such as over wired LAN 802.3 and over wireless LAN 802.11. As mentioned earlier, the 802.1X standard defines the supplicant, the authenticator and the authentication sever roles. The supplicant is the device that requires accessing the network. The authenticator is typically a managed switch that is physically or logically connected to the suppliant acting as a proxy between the supplicant and the authentication server. The authentication server typically a RADIUS server provides authorization.

The encapsulation method, called the EAP over LAN (EAPOL), is used between the supplicant and the authenticator. The authenticator and the authentication server communicate using the RADIUS protocol with EAP messages transmitted over RADIUS. The authenticator is not aware of the content of EAP messages; it just supports the 802.1X process by relaying EAP frames.

In remaining of the section, the authentication process would be described. The EAPOL packet format and the supplicant state machine would then be presented. Finally, the EAP authentication methods would be studied.

2.1.1 Authentication Process

As mentioned previously, the authenticator controls the supplicant access to the network. Once the 802.1X authentication function is enabled on the authenticator, a successful

authentication must occur before any network traffic is allowed to transmit from the supplicant. That includes critical traffic such as DHCP[1] requests regardless of whether link is established between the supplicant and the authenticator. This strict constrain ensures that unauthorized devices would not gain network resources until authentication is granted.

2.1.1.1 Before Authentication

To ensure that no unauthorized traffic is transmitted before a successful authentication, the authenticator's port is set to uncontrolled. The only packets that will be accepted from the client are EAP requests which will be forwarded to the authentication server (see Figure 2).

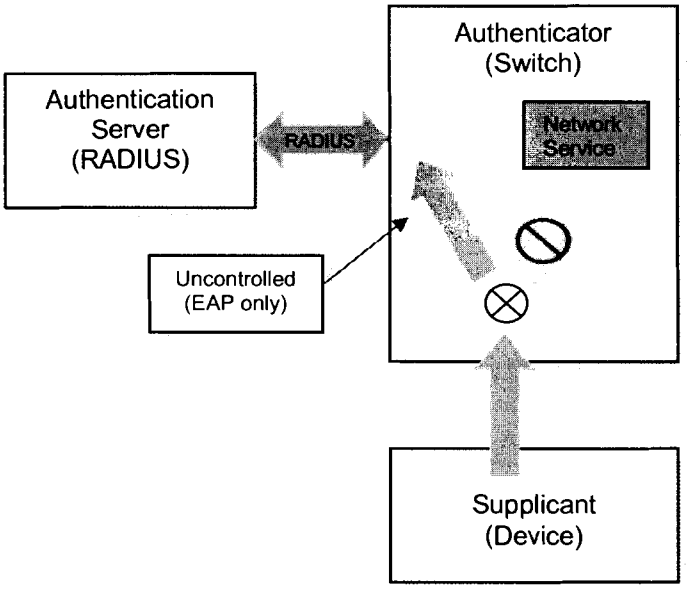

Figure 2: Before authentication

2.1.1.2 Authentication

Upon receiving the supplicant's EAP messages, the switch will forward the request to the authentication server without changing its content. Even though the EAP contents are not

[1] DHCP, or Dynamic Host Control Protocol, is a protocol for dynamically assigning IP addresses and related network information to devices.

changed, the encapsulation must be translated from the originating EAP message to a RADIUS request.

Upon receiving the RADIUS request, the authentication server will either grant or deny access to the network. A RADIUS response will then be transmitted back to the switch which keeps the port in an uncontrolled state if access is denied, or changes to a controlled state if access is granted (see Figure 3).

Although the supplicant is the device that requires authentication, either the supplicant or the authenticator can initiate the process. This decision is based on timers which are set in both the supplicant and the authenticator's state machines.

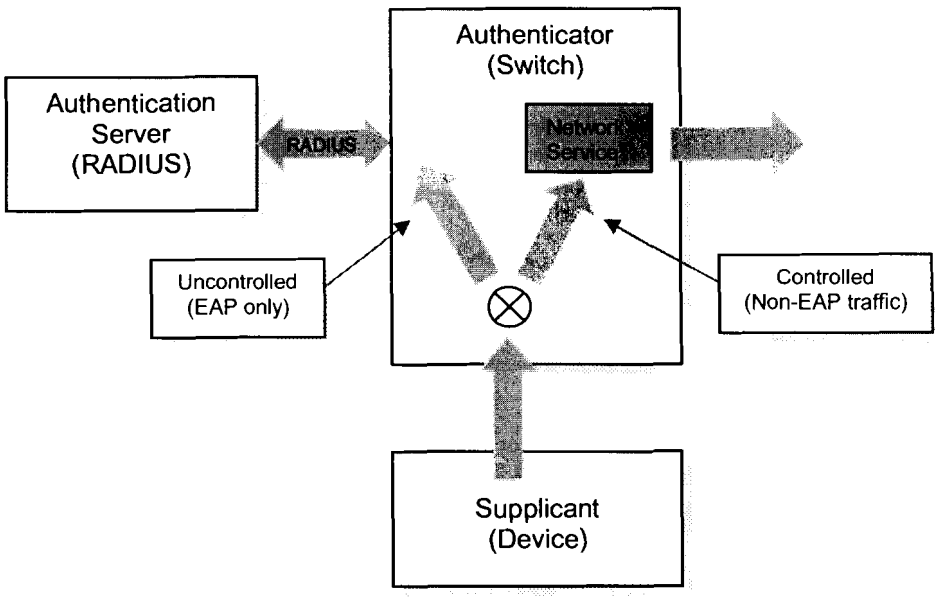

Figure 3: After successful authentication

2.1.2 EAP Encapsulation over LANs (EAPOL)

EAP packets are encapsulated by Ethernet frame when they are transmitted through Ethernet between the supplicant and the authenticator. The EAPOL frame format is shown in Figure 4. The important fields such as the Ethernet payload type, packet type and the packet body are described in the next sections.

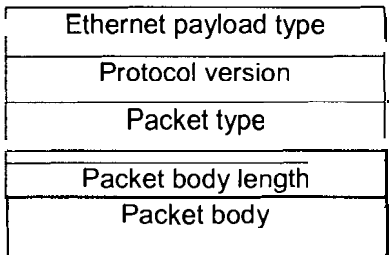

Figure 4: EAPOL frame format for Ethernet

2.1.2.1 Ethernet Addressing

An EAP packet is encapsulated on an Ethernet header that contains source and destination Ethernet MAC addresses of the packet. For media where the MAC address of the authenticator is not previously known, a standard 802.1X MAC address *01-80-C2-00-00-03* would be used as the destination address. This special address is one of the reserved addresses that switches should not forward [8]. The Ethernet type contains the protocol type of the EAP packet, which is set to *0x888E*. An 802.1X capable authenticator would limit network traffic to only packets with Ethernet payload type set to *0x888E* before the supplicant is authenticated.

2.1.2.2 EAP Packet Types

The various EAP packet types defined by the 802.1X standard are listed in Table 1.

Table 1: EAP packet types

EAP packet type	Description
EAP-Packet	EAP packets (*EAP-Request*, *EAP-Response*, *EAP-Success*, *EAP-Failure*).
EAPOL-Start	Packet sent by supplicant initiating the 802.1X handshake process.
EAPOL-Logoff	An explicit logoff request issued by the supplicant. Set the authenticator back to uncontrolled state.
EAPOL-Key	Optional frame used to transmit session key information to encrypt the data channel. Typically used for wireless LAN only.
EAPOL-Encapsulated-ASF-Alert	Used for allowing alerts (e.g., specific SNMP traps, warning messages, etc) to be forwarded through a port that is in the unauthorized state.

A typical call flow of the 802.1X authentication showing how a supplicant is granted network access is shown in Figure 5. The supplicant issues an *EAPOL-Start*, followed by a number of *EAP-Request / Response* pairs, and finally an *EAP-Success* (or *EAP-Failure*) to complete the process. For wireless LAN where packets can easily be eavesdropped on open air, a symmetric session key could optionally be delivered from the authenticator through the *EAPOL-Key* frame. The key can then be used to encrypt the data channel.

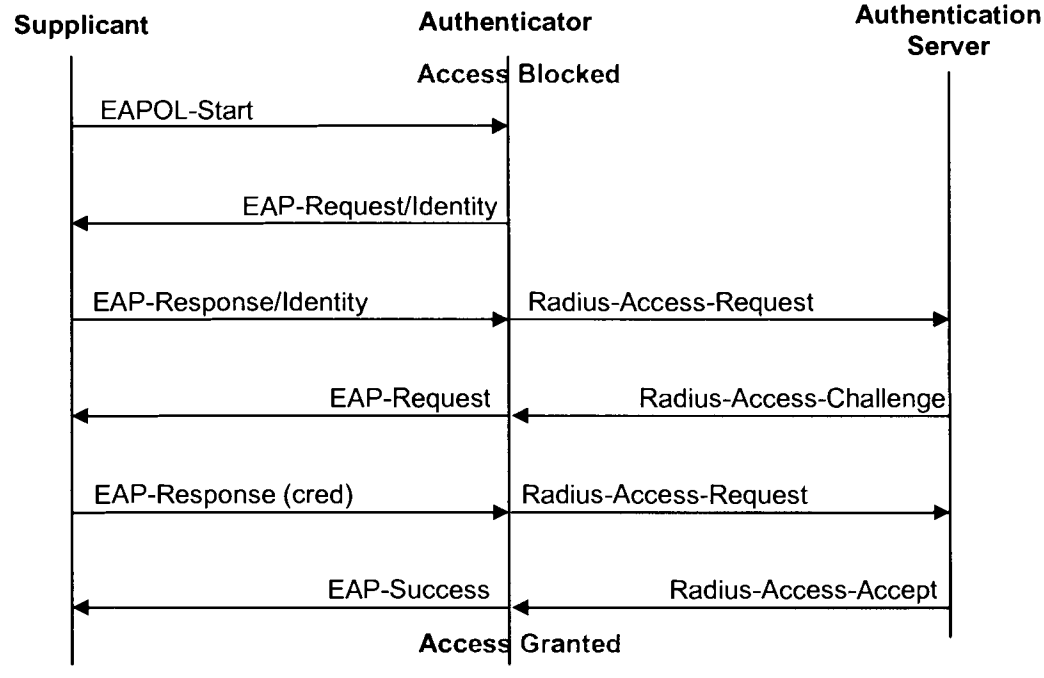

Figure 5: Basic 802.1X call flow

2.1.2.3 Supplicant State Machine

The IEEE 802.1X-2001 specification defines a set of state machines that a supplicant implementation should comply with. The state machines define a set of timers and events that the supplicant should react to. For example, at start up in the *Connecting* state, the supplicant sends the *EAPOL-Start* packet *maxStart* number of times each with *startPeriod* duration. When the counter has reached *maxStart*, the supplicant changes the state to *Authenticated*, meaning that the supplicant is not in an 802.1X network. Table 2 summarizes the operations of the state machine in various states. For completeness, the core supplicant state machine is included in Appendix A.

Table 2: Supplicant state transition table

State	Description / Action	Next state
Disconnected	Initial state. Associated port is operable.	• *Connecting*
Connecting	Attempting to acquire an authenticator. An *EAPOL-Start* packet is transmitted to the authenticator. Set the *startWhile* timer to *startPeriod*. Set *maxCount* to *maxStart*.	• *Connecting* (the *startWhile* timer expires) • *Authenticated* (number of *EAPOL-Start* frame sent has passed *maxStart*) • *Acquired* (received an *EAP-Request/Identity*)
Authenticated	Authenticated by authenticator, or network is not 802.1X aware.	• *Acquired* (received an *EAP-Request/Identity*)
Acquired	Sends an *EAP-Response/Identity* to authenticator. Set the *authWhile* timer to *authPeriod*.	• *Connecting* (the *authWhile* timer expires) • *Authenticating* (received an *EAP-Request* that is not *EAP-Request/Identity*)
Authenticating	Supplicant is authenticating to the authenticator. Sends an *EAP-Response* frame. Set the *authWhile* timer to *authPeriod*.	• *Connecting* (the *authWhile* timer expires) • *Held* (received an *EAP-Failure*) • *Authenticated* (received an *EAP-Success*) • *Acquired* (received an *EAP-Request/Identity*) • *Authenticating* (received an *EAP-Request*)
Held	Authentication failure signaled by the receipt of an *EAP-Failure*. This state provides a delay period before the supplicant will reattempt to acquire an authenticator. Set the *heldWhile* timer to *heldPeriod*.	• *Connecting* (when the *heldWhile* timer expires) • *Acquired* (received an *EAP-Request/Identity*)
Logoff	User has requested an explicit logoff. *EAPOL-Logoff* packet is transmitted.	• *Disconnected*

2.1.3 Authentication Methods

An important policy decision with authentication is which EAP authentication method to deploy. The supplicant and the RADIUS server must support the same authentication method. (The authenticator is not aware of the EAP authentication method; it only has to

support the 802.1X authentication process.) If the supplicant supports more than one EAP protocol, the RADIUS server must select an initial EAP protocol with which to proceed when receiving an authentication request with EAP credentials. Selecting an incorrect EAP protocol is not fatal; the client simply sends and *EAP-NAK* in response to the server's *EAP-Request* with the selected protocol and suggests an alternative one. After one additional network round trip, the correct EAP protocol becomes active.

2.1.3.1 EAP-MD5

The MD5 message-digest algorithm takes an input message of variable length to create a 16-byte "digest" or fingerprint of the input. The algorithm is computationally infeasible to reproduce the original message given a 16-byte digest. The MD5 is designed and intended for digital signature application to authenticate the identity of the sender.

EAP-MD5 is based on the Challenge-Handshake Authentication Protocol (CHAP) [5] for message exchange and Message Digest 5 (MD5) for authentication. The authentication method operates in a challenge-response manner, with the supplicant responding to the authentication server's challenge. The authentication server sends a challenge within an EAP packet along with a variable stream of octets called *Challenge Value*. Upon receiving the challenge, the supplicant responds with clear text of the user name, along with the user name, password and challenge message digested using MD5 into a 16-byte text.

The advantage of the EAP-MD5 method is that it is fast and simple; the CHAP transaction only takes a round-trip of handshake. However, EAP-MD5 suffers from brute force attach that the message digested text can be guessed eventually. It is important that the password be changed periodically if the EAP-MD5 authenticated method is selected. Also, EAP-MD5 is vulnerable to man-in-the-middle attack[2] because the authentication is one way and a malicious authenticator could trick the device to redirect all the packets.

---

[2] Man-in-the-middle attack is a terminology commonly used in cryptography to describe a security vulnerability which an entity in between the supplicant and authenticator could silently hijack all the packets sent from the supplicant.

To get around this problem, more secured methods such as the certificate based EAP-TLS authentication are developed.

2.1.3.2 EAP-TLS

The EAP-TLS method uses the Transport Layer Security, version 1 (TLSv1) protocol that is based on concept of Public Key Infrastructure (PKI) to provide mutual authentication. The PKI is a management system designed to administer asymmetrical cryptographic keys and public key certificates.

2.1.3.2.1 Asymmetric Cryptography

With symmetric encryption, both parties share the same key (or password) to encrypt and decrypt data. Conversely, with asymmetric encryption, a pair of keys called public and private keys is used; data encrypted with a public key can only be decrypted with its private key and vice versa. The primary disadvantage of symmetric key algorithms is that the security system is compromised once the key is exposed, making key distribution difficult. With a public and private key pair, only the public key for encrypting data is distributed. Since only the private key can decrypt the data and the key is never distributed, the encryption would not be compromised.

Asymmetric key encryption still suffers from man-in-the-middle attack; someone could intercept the public key and replaces the key, and would then be able to read the messages between the two parties. Public key infrastructure and digital certificates are developed to solve this problem.

2.1.3.2.2 Public Key Infrastructure and Digital Certificates

The PKI introduces the concept of having a trusted third party called the Certificate Authority (CA) to issue digital certificates that associate a person's or device's identity with a public-private key set. The certificate is signed by the issuer's private key with a validity period (typically in order of years), and anyone with the issuer's public key can verify the certificate's authenticity. For the 802.1X, the CA and the authentication server can possibly be the same entity.

2.1.3.2.3 The TLS Protocol

Developed by the IETF networking working group, the TLSv1 protocol is specified in RFC 2246 [4] and is currently a draft standard. The TLS protocol is inherited heavily from the Secure Socket Layer, version 3 (SSLv3), which is a standard developed by *Netscape* to secure HTTP connections for web pages [6]. Even though the TLS protocol was designed to operate above the transport layer using the Transmission Control Protocol (TCP), the same mechanism can be used for the 802.1X / EAP. The TLS protocol establishes a secured tunnel with the client and server mutually authenticating each other's certificates by verifying PKI signatures of certificates and the validity period. Both sides also agree on an encryption method used for the session. Figure 6 illustrates the TLS connection establishment operation with the TLS client initiating the handshake.

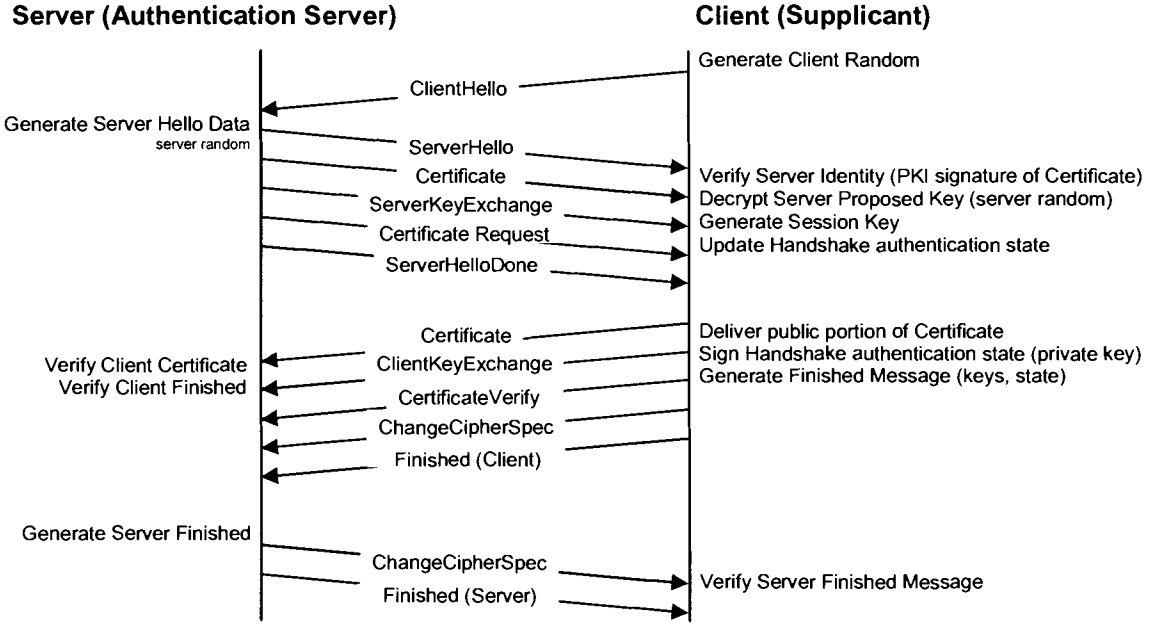

Figure 6: The TLS handshake process

2.1.3.2.4 Advantages and Disadvantages of EAP-TLS

The EAP-TLS method has an advantage that mutual authentication can be supported (comparing with EAP-MD5's one-way authentication). The use of certificates and keys,

however, make it difficult to provision offline (e.g., close impossible to provision through keypad). A related consequence with the provisioning problem is that unless the certificate is preloaded to the supplicant, it is bc impossible to provision the certificates over an 802.1X nctwork because the supplicant is not yet authenticated. Finally, the extra message exchanges with the supplicant and the authentication server slows down the authentication process comparing with simpler method such as the EAP-MD5 method described earlier.

2.2 PhonexChange Architecture

PhonexChange is the IP phone software suite for the *BCM91101* reference IP phone platform. The reference platform includes an *8MB* SDRAM, a *2MB* flash, a *2 x 16* LCD display, a keypad, indicators, a handsct and a speaker / microphone set, as shown in Figure 7. The reference platform, along with *PhonexChange*, serves as a reference implementation of an enterprise IP phone for the *BCM1101* silicon.

Figure 7: The *BCM91101* reference IP phone

The *PhonexChange* software consists of a set of device drivers, voice processing software, call signaling protocol stacks, provisioning software and reference call client applications. The software runs on a multitasking environment provided by the *VxWorks* embedded Real-time Operating System (RTOS). Most of the software was written in *ANSI C* although some of the third-party call-signaling protocol stacks were written in *C++*. The timing critical drivers and DSP software were respectively written in *MIPS* and *ZSP* assembly languages.

Figure 8 illustrates a high level overview of the *PhonexChange* software architecture. The *PhonexChange* architecture is presented as a set of interdependent tasks interacting with each other via the set of well-defined callback functions and data buffers.

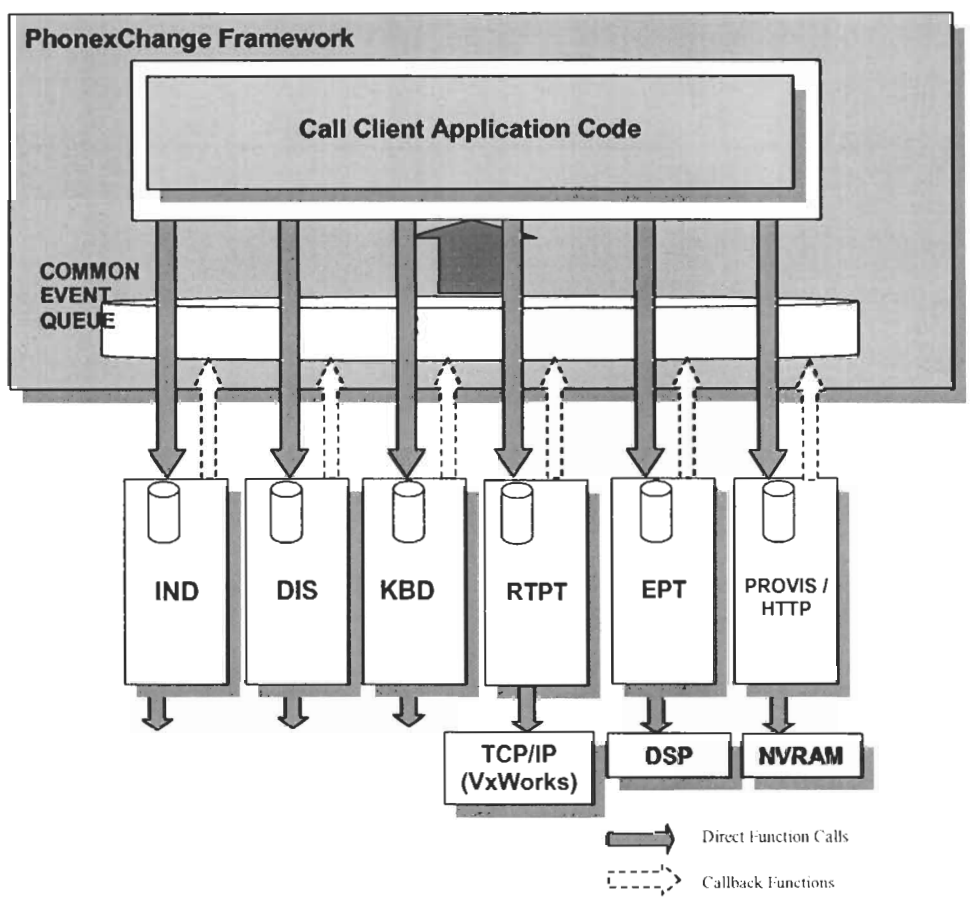

Figure 8: ***PhonexChange*** **architecture**

The call client module can be considered as the brain of the *PhonexChange* software. It contains the necessary callback functions and set the callback function pointers when initializing with the corresponding software components. The call client module directly synchronizes the activity of the following *PhonexChange* software components.

- PROVIS – NVRAM / DHCP configuration provisioning module
- HTTP – HTTP server provisioning module
- RTPT – RTP/RTCP voice stream stack.

- EPT – Endpoint module. DSP resource management.
- DIS – Display driver
- KBD – Keypad driver scans keypad events
- IND – LED indicator driver

Driver and library callback functions may execute in the context of an unknown task defined in the corresponding software component. To provide synchronization, each callback function provided by the call client posts the function's type and parameters as a message to a common call client event message queue. The main task loop in the call client then dispatches events to appropriate handlers as new messages arrive. A timeout on the queue provides for periodic timer processing.

2.2.1 VxWorks

The *PhonexChange* software runs on the *Wind River VxWorks* operating system. *VxWorks* is a *Linux*-like RTOS that comes with a complete set of network services such as a TCP/IP stack, a Telnet server, a FTP server, etc. The faster context switches and interrupt processing comparing with the *Linux* operating system makes *VxWorks* a more suitable operating system for embedded systems [7]. The OS comes with cross-compiling tools allowing users to develop software on *Windows* and other contemporary operating systems.

The latest *VxWorks* offering, the Platform for Consumer Device (PCD) version 2.0, bundles additional features that are suitable for consumer devices such as IP phones, printers, handset, etc. The bundled features include a fault-tolerant file system, an 802.1X supplicant stack (*Wind NET* supplicant), a graphics library for rendering fonts and graphics (*Wind ML*), an IPSec / IKE security stack, etc, that help shorten development time. Most of the components are written in *C* and are delivered as source format so custom changes can be implemented quite easily.

2.2.2 Boot up sequence

Both the boot up code and the phone application code operate on the *VxWorks* OS. The boot up code is responsible for decompressing the application to SDRAM and executing it at the start address. The code also contains the TCP/IP network stack, an Ethernet driver and a file system. When the device is booted up, depending on the configuration, the application code could be downloaded via TFTP or via the flash memory to the SDRAM for decompression. Once the application code is executed and upon initialization of the operating system, the call client application would gain ownership of the system. The call client first initializes the event queues and buffer pools. After that, it initializes the drivers, hardware subsystems, and begins the provisioning process.

2.2.3 Provisioning

The PROVIS module is responsible for provisioning and retrieving system configuration parameters via the non-volatile flash memory (NVRAM), key pad (user input) or DHCP. Configuration parameters such as network settings (IP address, subnet mask, default gateway, DNS IP, etc) and application settings (phone number, call server address, Type of Service (TOS) settings, etc) can be provisioned.

At startup, the call client application prompts the user for provisioning over keypad. If the user chooses not to modify the settings, then the PROVIS module may retrieve data from either the NVRAM or DHCP, depending on the configuration. The call client application then proceeds with starting up the rest of the system such as initializing the DSP software, drivers, signaling protocol stack, etc. Typically upon successful registration with the call server, the phone is ready for phone calls. The PROVIS module is not designed to be accessed beyond startup.

The HTTP module implements a HTTP server that allows users to provision the configurations via web forms with a HTML browser on a PC, as shown in Figure 9. Multiple web forms have been setup to configure network and application settings as for the case with the PROVIS module. Additionally, it is capable of performing silicon self-test and application firmware upgrade.

The previously stored configuration parameters are retrieved directly from the NVRAM and are displayed on the web forms. The user can choose to modify the default parameters and submit the form. The modified data would then be validated and stored in the NVRAM, overwriting the old configuration. The newly provisioned parameters would be used upon the next boot cycle.

Web-based provisioning is advantageous because web forms are versatile. A web form could easily be extended to provision configurations that involve large data that are impossible or at least inconvenient to provision using the phone's keypad. User certificates and private-public key pairs used for 802.1X make good use of web forms. On the other hand, web-based provisioning could introduce security threats. For example, someone could capture the web traffic on the network to obtain the credentials of the phone. Nevertheless, secure provisioning methods vary with customers and therefore is out of the scope of this project.

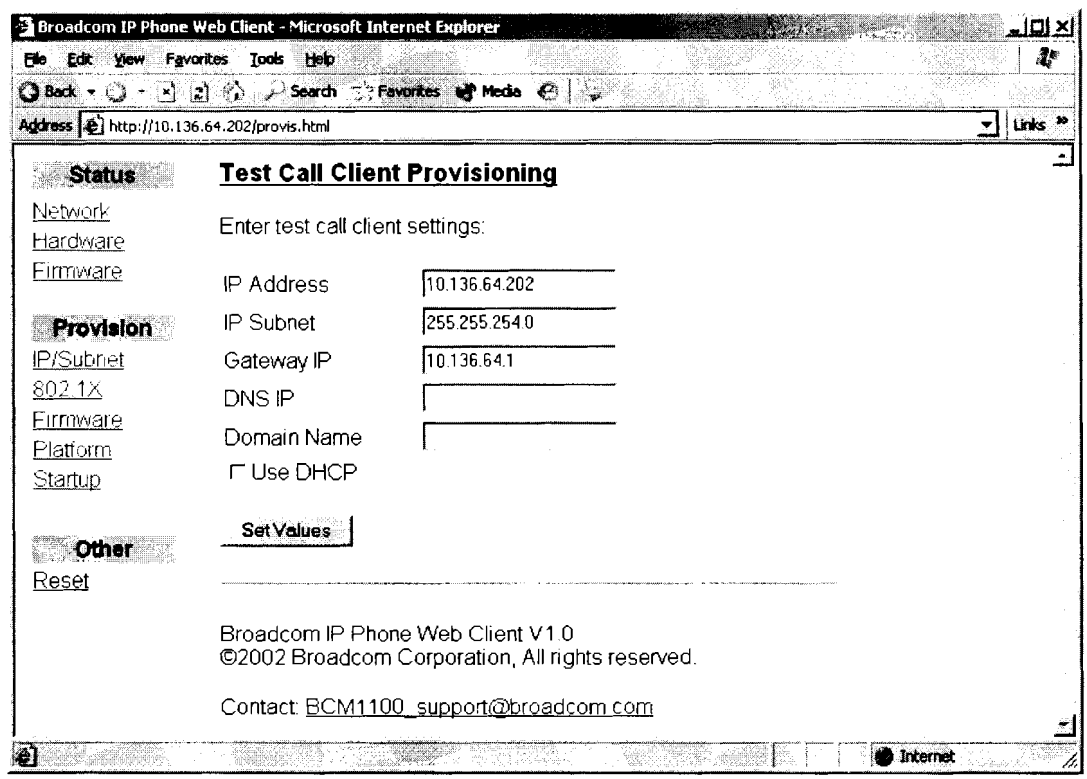

Figure 9: Wed-based provisioning with the reference design IP phone

2.2.4 The Two-port Ethernet Switch

The *BCM1101* chip contains a two-port Ethernet switch that is designed to have one of the ports connected to the LAN while the other one is connected to the PC. The Ethernet driver within *PhonexChange* is responsible for sending and receiving Ethernet frames between the *VxWorks* TCP/IP stack and the two-port switch. The switch manages most of the packet routing on its own, except when the destination MAC address matches the list of reserved switch specific MAC addresses including the 802.1X special address that is defined in the 802.1D specification [8]. The Ethernet frames with the special MAC address as destination are forwarded to the Ethernet driver for further processing. The Ethernet driver currently tosses all these packets.

Chapter 3: Design and Implementation

Based on the design requirements, decisions were made when developing the supplicant software. This chapter presents the design of the supplicant in a top-down approach. Section 3.1 discusses the design considerations. Section 3.2 presents the high level design of the supplicant for the *PhonexChange* software. Section 3.3 describes the interface design of the supplicant software. Section 3.4 details the internal architecture of the supplicant software. Finally, Section 3.5 summaries this chapter.

3.1 Design Considerations

The goal of the project is to implement the 802.1X supplicant for the *PhonexChange* software on the *BCM91101* reference IP phone platform. The software design is constrained by the hardware platform (memory, processor speed, etc), standard protocol specifications, customer requirements and existing software conventions. This section discusses these factors.

3.1.1 802.1X Requirements

The software would support the standard 802.1X-2001 supplicant specification. The requirements include supporting the supplicant state machine as well as the timers and counters that the state machine defines. The EAP-MD5 and EAP-TLS authentication methods would be supported.

3.1.2 Configurations

The user should be able to enable or disable the supplicant software both at compile time and at run time. Also, the authentication method and credentials should be configurable at run time.

3.1.3 Memory Footprint

The *BCM91101* reference platform includes *8MB* of SDRAM and *2MB* of non-volatile flash memory. The application code and data without the 802.1X supplicant occupy *4MB* of SDRAM and *1MB* of flash. The supplicant code must fit into the remaining memory.

3.1.4 Programming Language

Similar to the majority of the *PhonexChange* software, the supplicant software would be written in standard *ANSI C*.

3.2 High-level Design

This section presents the high level design of the supplicant software for the *PhonexChange* architecture. The interaction between the supplicant software with the rest of the system will be introduced.

3.2.1 Supplicant Module

The supplicant software fits into the *PhonexChange* architecture by following its convention. In particular, a supplicant (SUPP) module dedicated to the supplicant functionality is designed and is shown in Figure 10. The supplicant module hides the protocol details while giving the application the flexibility to control (such as proper error handling) and interact with it. Similar to the other modules, the supplicant module interacts with the call client application to access the other *PhonexChange* components such as the provisioning module.

The supplicant module contains well-defined public function calls or Application Program Interfaces (APIs) and event notification functions. While the supplicant module was integrated with the example reference design, well-defined interfaces help customers as their applications would be using it. Defining the interface also minimize software maintenance effort and improve software portability. The supplicant module APIs are protected by semaphores to ensure that the module is thread-safe[3]. The design of the supplicant module interfaces are described in Section 3.3.

---

[3] Thread-safe is a term to describe that the software contains a mutex such that it is possible for multiple threads to access the supplicant module at the same time without causing data inconsistency.

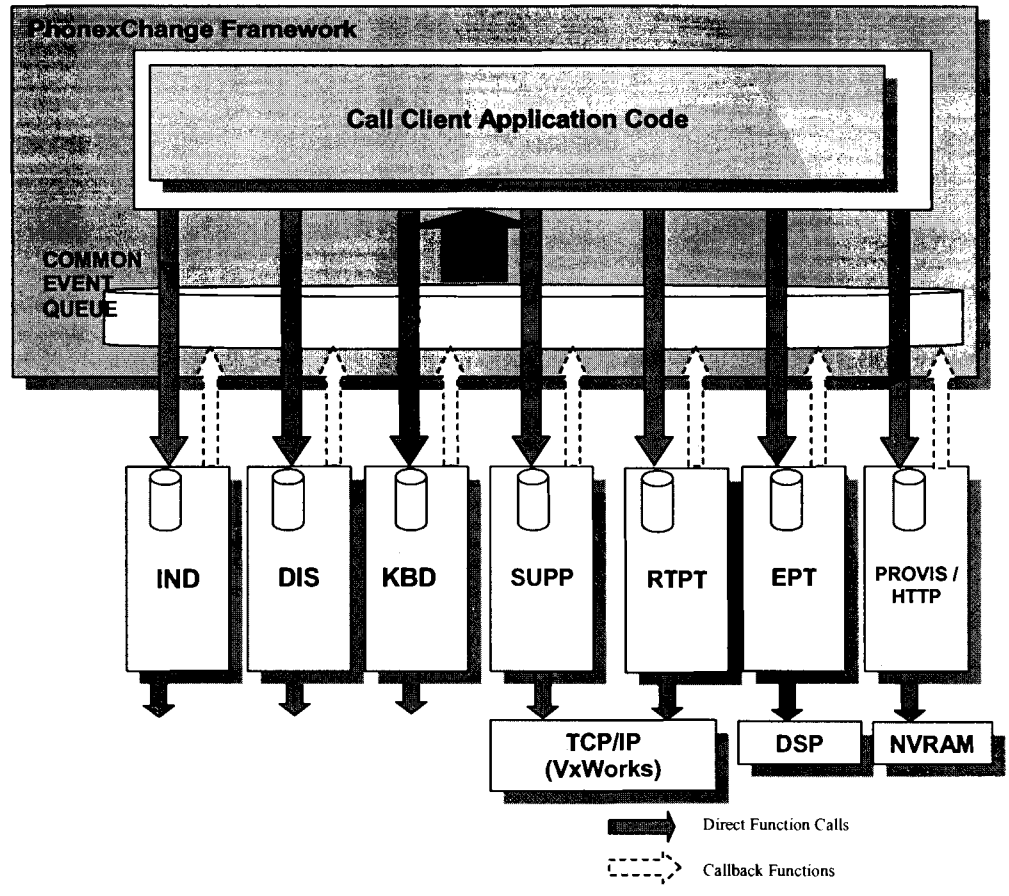

Figure 10: ***PhonexChange*** **architecture along with the supplicant module**

The supplicant module accesses the *VxWorks* TCP/IP stack directly to send and receive EAP packets. The software invokes appropriate API calls to bind itself with the TCP/IP stack so that Ethernet packets with the EAP payload type would be forwarded to the supplicant.

3.2.1.1 The 802.X Protocol Stack

Due to time and resource constrain, third party implementations are preferred over in-house, given that the third party implementation match the requirements and justify the cost.

The *Wind NET* supplicant from PCD 2.0 implements the supplicant state machine from the 802-1X specification. The timer constants and retry counts with the supplicant state machine are configurable at run time during startup. The supplicant supports only the

LEAP authentication method[4], though the software can be extended to support additional EAP methods.

An alternative is the *x-supplicant* software, which is a GNU Public License-based open source implementation that supports 802.1X along with many EAP methods such as EAP-MD5, EAP-TLS, EAP-TTLS, PEAP, LEAP, etc. The software is primarily written for *Linux*. Due to licensing issue and the fact that the software has to be ported to *VxWorks*, the *Wind NET* supplicant was chosen to be used instead.

3.2.1.2 The TLS Protocol Stack

In additional to supporting the 802.1X protocol, the supplicant module supports the EAP-MD5 and EAP-TLS authentication methods. The EAP-TLS authentication in particular needs to support the TLS protocol. There are a few third party implementations available (such as the *OpenSSL* and *MatrixSSL* software) each with their advantages and drawbacks. Therefore, a TLS interface was defined to decouple the dependent of the supplicant module with a specific TLS protocol stack. For the *PhonexChange* software, the *OpenSSL* software was chosen as the TLS stack because it is field-proven.

The *OpenSSL* software is an open-source software written in *ANSI C*. The *OpenSSL* software implements the Secure Socket Layer (SSLv2/v3) and Transport Layer Security (TLSv1) protocols, along with full-strength standard crypto algorithms such as RSA, SHA, AES, Blowfish, IDEA, etc [10]. Originally developed by Eric A. Young and Tim J. Hudson, the *OpenSSL* project is managed by a worldwide community of volunteers. The software comes with a command-line based tool for the crypto operations, as well as static libraries and header files that an application can be linked and built with. While some of the crypto algorithms such as the Blowfish are subjective to export restrictions, the *OpenSSL* software itself is licensed under an *Apache*-like license, that is, the software is free for distribution in both source and binary forms as long as the distributor acknowledges the use of the software in the documentation.

---

[4] LEAP or light weight EAP is a proprietary EAP method developed by Cisco.

3.2.2 Provisioning

At startup, the call client initializes the device drivers and retrieves the 802.1X configurations through the PROVIS module. Depending on the configuration, the call client may choose to enable with a specific EAP method, or to disable 802.1X supplicant. The call client application then proceeds with the rest of the startup process.

Due to the large size of the key and certificates, the HTTP server on the IP phone is chosen for provisioning the 802.1X configurations. As shown in Figure 11, an html page was created for the 802.1X supplicant that allows the user using a web browser to enable the supplicant with an EAP method (MD5 or TLS), or to disable it. The EAP method parameters, such as the password, the private key, the certificates, are available for configuration once an EAP method is selected.

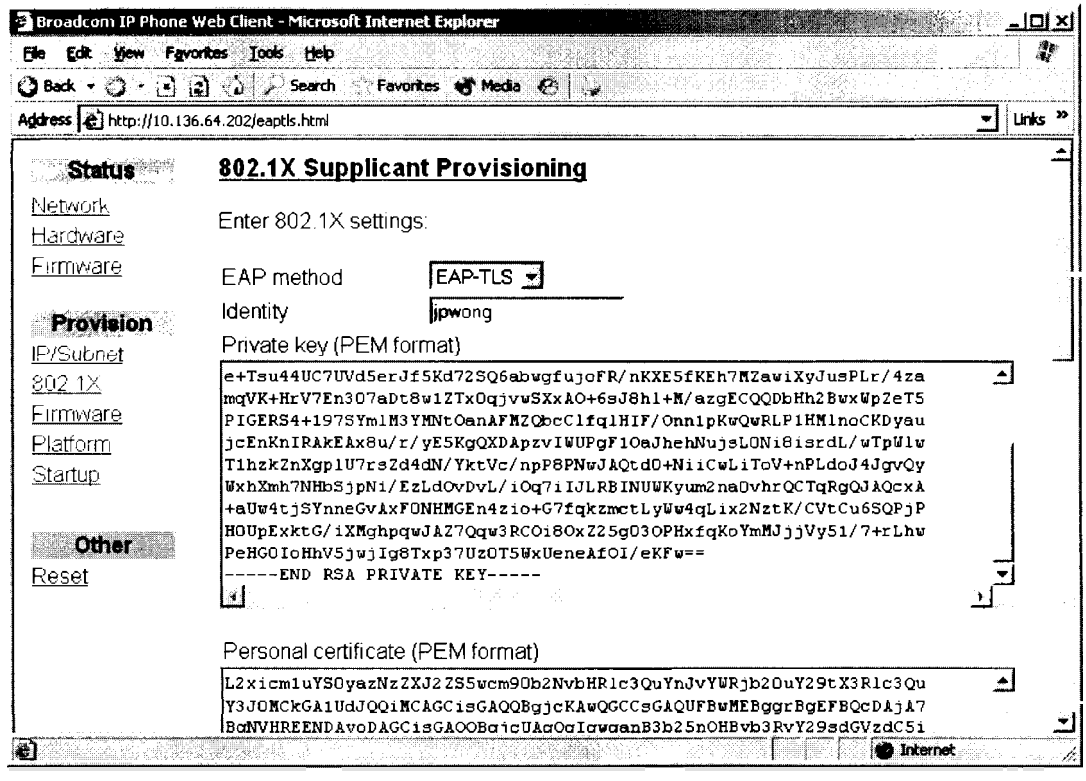

Figure 11: 802.1X supplicant provisioning (EAP-TLS)

Table 3 lists the 802.1X supplicant parameters to provision. To simplify the provisioning process for the reference design, the local certificate is assumed to inherit directly from

the root certificate with only one trusted certificate. The limited provisioning is sufficient to verify the supplicant software's operation and is a typical setup for a private CA in a corporate LAN. It is expected that customers may further enhance the provisioning for their products[5].

Once the user has submitted the 802.1X html form, the phone stores the new configurations to the NVRAM, which would be effective after reboot.

Table 3: 802.1X supplicant configuration parameters

Category	Parameter	Description
General	802.1X supplicant feature	Choices: Disable / Enable (EAP-MD5) / Enable (EAP-TLS).
	Identity	Identity to be sent as part of the *EAP-Response/Identity* frame.
EAP-MD5	Password	MD5 password to be message digested for the *EAP-Response*.
EAP-TLS	Local certificate	Phone's local certificate in Privacy Enhanced Mail (PEM) format.
	Private key	Private key of the local certificate in PEM format.
	Root certificate	Root certificate of the local certificate in PEM format.
	Trusted certificate	Trusted certificate in PEM format.

[5] Depending on the configuration, the local certificate may be inherited from one or more intermediate certificates in a hierarchy along with the root certificate. In order for the authentication server to validate the local certificate, the certificate's signature, which is signed by the issuer's private key, is examined using the issuer's public key. The validity period of the certificate is also verified. The root certificate must be within the list of trusted certificates.

3.2.3 The Ethernet Driver and the two-port Switch

Since EAP packets, at least for the initial ones, contain the same standard 802.1X destination MAC address (*01-80-C2-00-00-03*), the topology creates an interesting scenario with both the IP phone and the PC act as a supplicant seeking for an authenticator on the LAN port. The switch routes all frames with the special MAC address as destination to the Ethernet driver and three special rules are created.

1. EAP packets received from the PC port are sent only to the LAN port.
2. EAP packets received from the phone are sent only to the LAN port.
3. EAP packets received from the WAN port are copied to both the phone and the PC port.

The last rule is necessary because it is unknown if the packet is intended for the phone supplicant or for other supplicants on the PC port.

3.2.4 Data Store

The supplicant software uses data storage for storing EAP credentials, states, and counters. With the exception of the third party software, all the data structures were chosen to be static allocated. Using static data store has an advantage that the data store is readily available. The disadvantage is that static data that is not used otherwise would be wasted.

3.3 Interface Design

This section presents the interface specification and the API function calls of the supplicant module defined based on the high-level design. A typical usage of the supplicant API would then be described.

3.3.1 Interface Specification

The interfaces of the supplicant module can be categorized into three types: the application interfaces (upper-edge), the transport interfaces (lower-edge), and the control interfaces (side-edge) (see Figure 12). The application interface enables the application

to notified with state change with the supplicant state machine. It allows the application to send command (i.e., Sending off an *EAPOL-Logoff* request). The transport interface provides the ability for the supplicant module to send and receive EAP packets over Ethernet. This interface would be interacting directly with the TCP/IP stack. Finally, the control interface enables run-time initialization and configuration of the supplicant module.

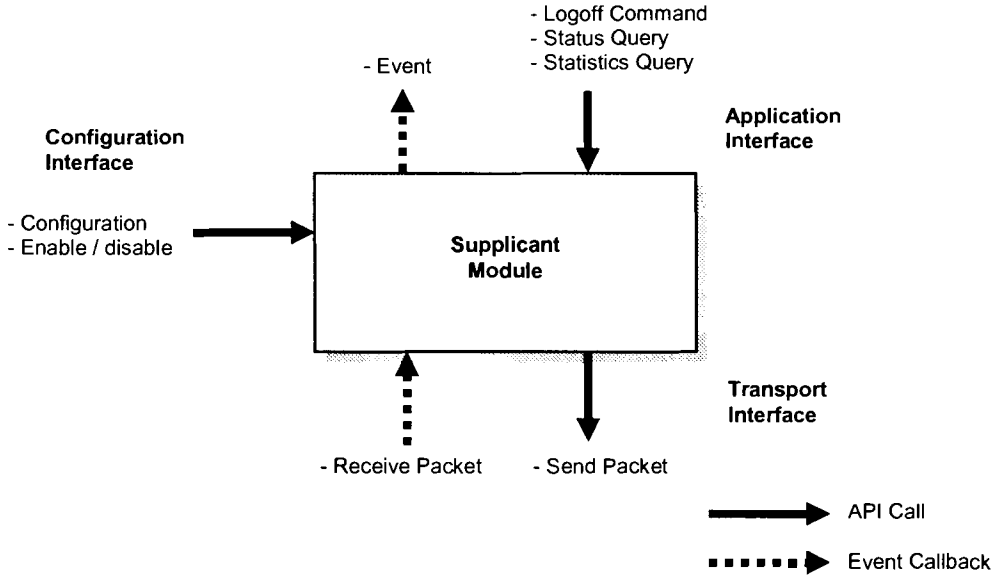

Figure 12: Supplicant module - interface specification

Each bullet in Figure 12 represents an interface with the supplicant module. A solid line represents an API call whereas a dotted line represents an event callback. In order to avoid the overhead of having the application handle the various supplicant states, the supplicant module would be hiding this to within the module while notifying the application whenever the supplicant state changes.

The APIs that realize the interfaces are listed in Table 4. Note that the supplicant module accesses the *VxWorks* TCP/IP stack directly and therefore no interfaces were designed for the transport interface.

Table 4: APIs correspond to the interfaces

Interface	API
Enable / disable supplicant	*suppInit*() – Initialize supplicant *suppStart*() – Start service *suppStop*() – Stop service
Configuration	*suppSetParm(XXXCREDITS)* – Set credentials *suppGetParm(XXXCREDITS)* – Get credentials
Statistics query	*suppSetParm(RESETSTATS)* - Reset statistics *suppGetParm(STATS)* – Query statistics
Status query	*suppGetParm(STATUS)* – Query status
Logoff command	*suppStop*() – Send *EAPOL-Logoff* before stopping service
Event	SUPPEVT(*prevEvent*, *newEvent*)

3.3.2 Detailed Interface Description

This section describes the typical usage of the supplicant API. The detailed API definition for the supplicant module is included in Appendix B.

3.3.2.1 Configuration

suppInit() is expected to be the first API to invoke. The API initializes the 802.1X supplicant service with an EAP method (EAP-MD5 or EAP-TLS). An event callback function pointer is also supplied for notifying the client application when there is a state change in the EAP supplicant state machine as defined in 802.1X-2001 and listed in Table 2. The routine also starts a timer that wakes up every second to check for the three timers defined by the state machine.

When the event callback function is invoked, both the original and new states are reported. Monitoring the state change gives the application an idea of the current state of the 802.1X service. It is up to the application how to react to the state change. For example, the application might hold on to the rest of system start-up until it received a state change from *Authenticating* to *Authenticated* (i.e. supplicant authenticated) or a

state change from *Connecting* to *Authenticated* (i.e. authenticator not responding / does not exist).

suppSetParm() provisions the authentication information for the specific authentication method. For example, when EAP-MD5 is used, it is expected that **suppSetParm**(SUPPPARM_MD5CREDITS) be called with the identity / password credentials before the supplicant service is started.

3.3.2.2 Operations

suppStart() enables the 802.1X supplicant service. In particular, the function associates the supplicant module with the *VxWorks* TCP/IP stack so that 802.1X packet (i.e., packets with Ethernet payload type set to the EAP packet type *0x888E*) would be routed to the service. The supplicant service then begins the supplicant state machine in the *Connecting* state. As discussed in Section 2.1.2.3, the supplicant starts sending *EAPOL-Start* frames, either to wait for the retransmission timer *startWhen* to expire, or to wait for the authenticator to send an *EAP-Request/Identity* frame. Note that the EAP method parameter must be provisioned with **suppSetParm()** before the supplicant service is started so that the supplicant could supply the identity and credential information for the specific EAP method.

suppGetParm(BSWPARM_STATUS) can be used to query the state of the supplicant state machine as well as the (default) value of the *startWhile*, *authWhile* and *heldWhile* timers, and the *startWhen* counter. In addition, **suppGetParm**(BSWPARM_STATS) can be used to query the supplicant statistics such as packet sent / received, source MAC of the last received 802.1X packet, etc. **suppSetParm**(SUPPPARM_CLEARSTATS) resets the collected supplicant statistics.

Finally, **suppStop()** disables the 802.1X supplicant service and removes the 802.1X service from the TCP/IP stack. An *EAPOL-Logoff* is sent before the supplicant service is stopped. Then, the timer tick is turned off. If the client application wish to opt for

another EAP method, at this point it could invoke **suppInit()** for the desired EAP method.

3.3.2.3 Error Handling

Errors are reported synchronously upon the return of the APIs. An error is returned when the user has specified an invalid parameter, or an API is invoked at an unexpected state (e.g., **suppStop()** invoked before **suppInit()** is invoked). Protocol-level failures and timer expiries are conveyed through the event callback function. For example, state transition from *Authenticating* to *Held* indicates that an *EAP-Failure* packet is received. The statistics returned from **suppGetParm()**

3.4 Internal Architecture

This section details the internal architecture of the supplicant module. The supplicant module is organized into three main components: the interface layer, the *Wind NET* supplicant stack and the EAP method handler (see Figure 13). The interface layer implements the API and event callback function for the supplicant module, ensuring that the supplicant module conforms to the *PhonexChange* architecture. The *Wind NET* supplicant stack implements the 802.1X supplicant state machine and handles EAP packets with the *VxWorks* TCP/IP stack. Finally, the EAP method handler manages the EAP method specific *EAP-Request* packets from the authenticator. Since the supplicant module uses the *Wind NET* supplicant stack, the design of the supplicant module is influenced by the *Wind NET* supplicant.

3.4.1 The Interface Layer

The interface layer relays API calls and events synchronously between the client application and the *Wind NET* supplicant. It ensures that the interface of supplicant module conforms to the *PhonexChange* architecture. Specifically, all APIs are protected by a semaphore so that only one task can access the internal data at a time. The interface layer also simplifies and separates the call client's interaction with the *Wind NET* module.

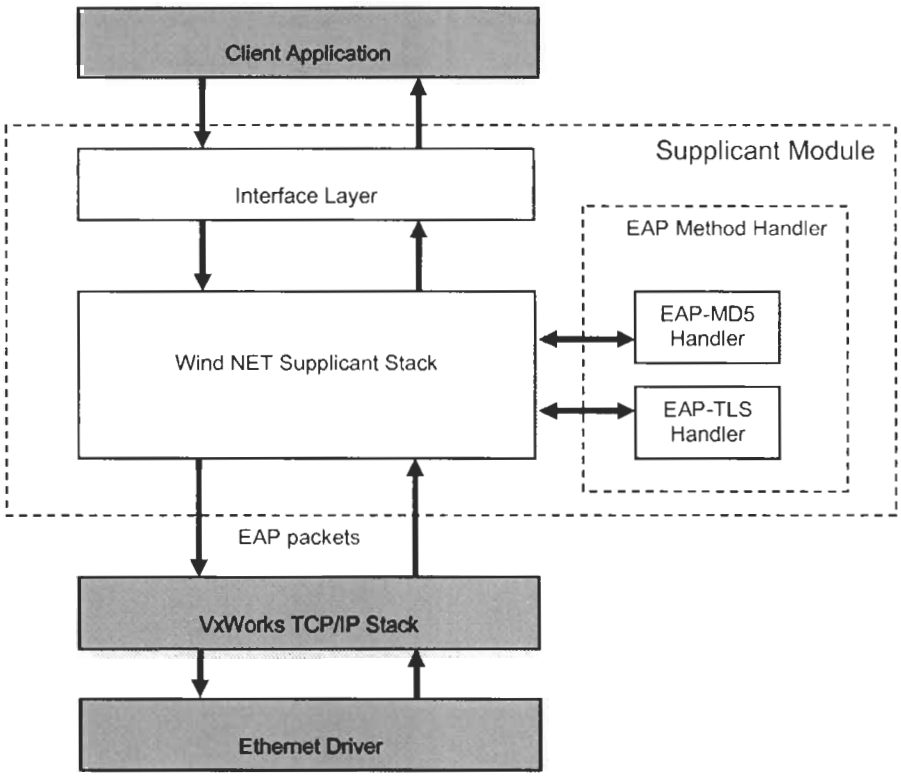

Figure 13: Components of the supplicant module

3.4.2 The Wind NET Supplicant Stack

The *Wind NET* supplicant stack contains a number of passive components and is designed to be event-driven. The components generally belong to the supplicant state machine parsing and processing inbound EAP packets, and creating outbound EAP packets, as shown in Figure 14.

At startup, a task is created to listen for incoming events (incoming EAP packets, user event, timer tick, etc), a system message queue is created for queuing events in first in, first out order, and a system timer is configured to generate a timer event to the message queue every second from an unknown system task. When an EAP packet arrives, the packet would be parsed and if valid, would be inserted to the message queue. The internal task, which is waiting for incoming events, wakes up and processes the packet. Depending on the packet type, the state machine may updated, in which case would generate a state change event to the application.

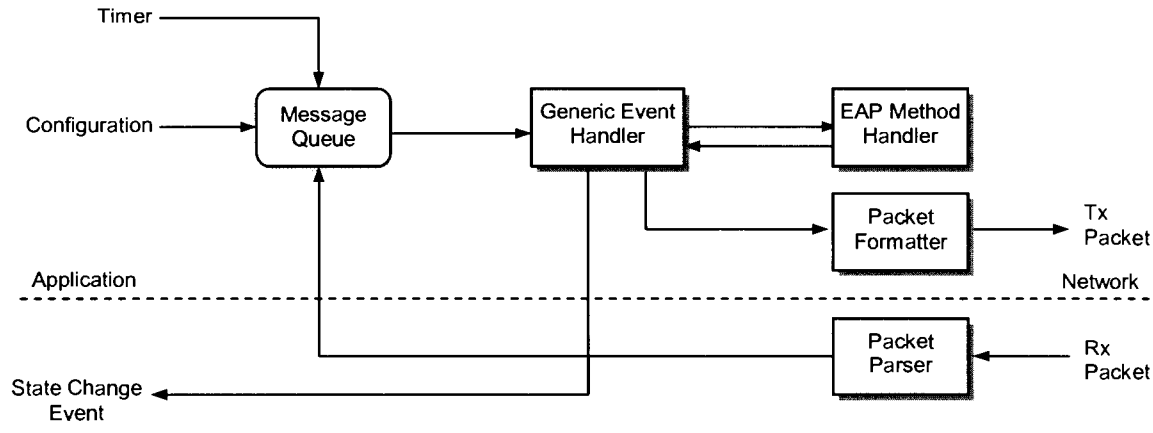

Figure 14: Internal architecture of the *Wind NET* supplicant stack

The *Wind NET* supplicant stack handles only generic EAP packets such as *EAPOL-Start*, *EAP-Request/Identity*, *EAP-Response/Identity*, *EAP-Success, EAP-Failure* and *EAPOL-Logoff*. EAP specific requests are handled by the EAP method handler. Figure 15 shows the call flow of the *Wind NET* supplicant.

3.4.2.1 The EAP Method Handler

While the *Wind NET* supplicant stack handles only generic EAP packets, the EAP method handler manages all (non *EAP-Request/Identity*) *EAP-Request* packets from the authenticator and generates appropriate *EAP-Responses*. For this project, the EAP-MD5 and EAP-TLS methods are required, and hence the EAP-MD5 and EAP-TLS handlers were implemented. At initialization when the **suppInit**() invoked with the EAP method configured, the supplicant stack setup the method handler to use so that when an *EAP-Request* frame is received, it would be passed to the method handler.

The actual payload contents within the *EAP-Requests* and *EAP-Responses* are specific to the authentication method used, and the number of *EAP-Request / Response* pairs could go from one to as many as the authentication method needs. EAP-TLS, for example, normally involves five transactions. In order to properly manage the EAP transaction sequences, a mini state machine was implemented for each handler to handle the specific *EAP-Request / EAP-Response* sequences. The mini state machine operates in conjunction with the supplicant state machine to complete the authentication process. An

EAP specific data structure was also created to store its current state and credential information. For all EAP methods, the authenticator completes the negotiation process by indicating the authentication status with either an *EAP-Success* or an *EAP-Failure*.

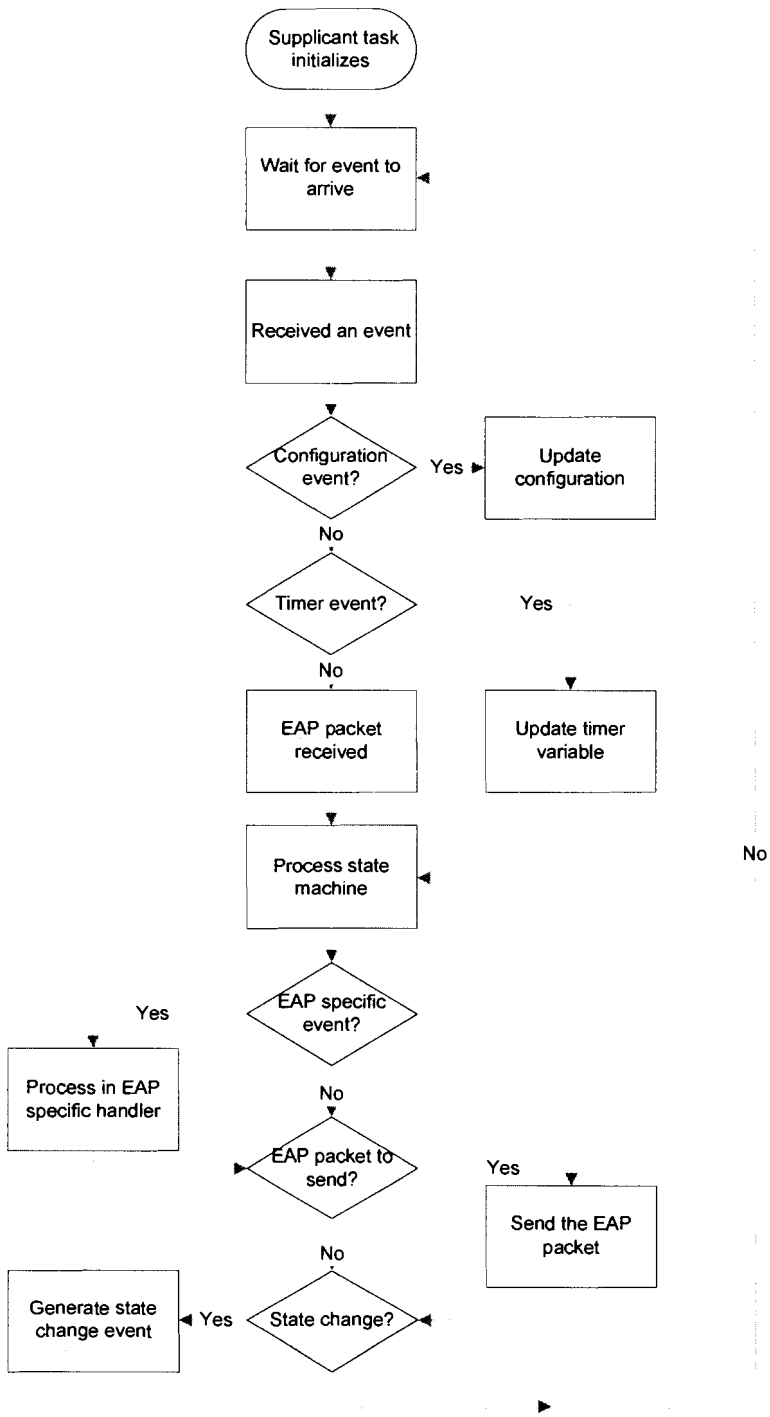

Figure 15: Supplicant task flow chart

3.4.2.2 The EAP-MD5 Handler

As discussed earlier, MD5 is a challenge-response authentication method. The authentication server sends an *EAP-Request* along with a challenge token and the EAP-MD5 handler composes a response based on the challenge token, the user identity and the user password, digested with MD5.

Figure 16 illustrates the state machine of the EAP-MD5 handler. At initialization, the handler allocates a MD5 data block to store the current state of the handler. The state machine then enters the *WAIT_FOR_CHALLENGE* state. When an *EAP-Request* frame with the challenge token is received, the handler generates the response, and is transitioned to the *WAIT_FOR_SUCCESS* state. Finally, the state machine transitions to the *AUTHENTICATED* state when an *EAP-Success* frame is received. Note that *EAP-Request / Identity* reset the state machine, and the *EAP-Failure* frame is processed by the main supplicant state machine.

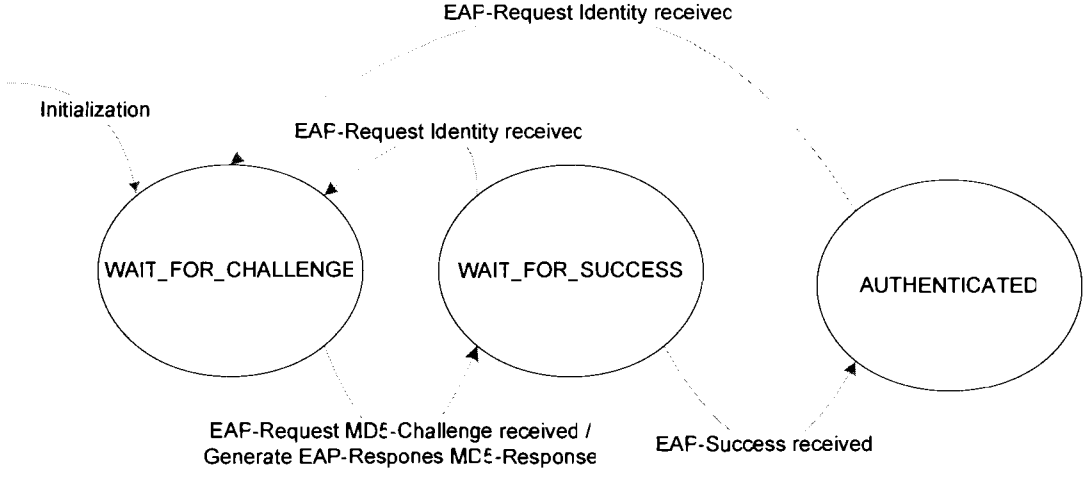

Figure 16: State machine of the EAP-MD5 handler

3.4.3 The EAP-TLS Handler

The EAP-TLS handler implements a TLS client (as oppose to the TLS server at the authentication server). Essentially, the handler establishes a TLS connection with the authentication server over EAP via the authenticator. Establishing the TLS connection

ensures that both the client and server are mutually authenticated as the peer certificate can be verified against the trusted list of certificates locally.

The EAP-TLS handler uses the *OpenSSL* library as the TLS stack. As discussed earlier, while the *OpenSSL* library contains industrial strength crypto algorithms and a complete SSL/TLS stack, its memory (code, static and dynamic data) footprint of *~800kB* on *VxWorks* can discourage some customers with devices with small memory footprints. To ease the portability of the supplicant software to use another TLS implementation, a TLS client wrapper was written to abstract the *OpenSSL* library from the EAP-TLS handler. The wrapper contains only standard *ANSI C* data types, making it easier to integrate with other TLS solutions. The wrapper is also generic that its use is not limited to the supplicant software. The TLS client wrapper API is summarized in Table 5, and its detailed description and usage are included in Appendix C.

Table 5: The TLS client wrapper API

Interface	Description
tlsInit	Initializes the TLS stack and creates a context that can be referenced until deleted with **tlsDeinit**(). The context stores connection specific preferences and certificate-related information.
tlsConfig	Configures the TLS stack with the created context. Information such as certificates, private key, and authentication preference can be provisioned.
tlsCreate	Creates a TLS session based on the context. The TLS session represents the TLS connection.
tlsProcess	Processes an incoming TLS frame and generate a response packet (if any).
tlsDelete	Deletes the created TLS session.
tlsDeinit	Delete the created TLS context and de-initializes the TLS stack.

Figure 17 describes the state machine of the EAP-TLS handler. At startup, the EAP-TLS handler initializes the TLS stack and allocates a TLS data block to store the current state of the state machine and credential information. The state is then set to

WAIT_FOR_TLS_START. *TLS-Start* is a special flag in the first octet of the *EAP-Request* payload from the authenticator, which is normally sent after the *EAP-Request /Response Identity* transaction. When an *EAP-Request* with the *TLS-Start* flag set is received, the TLS client initiates a TLS connect request and transitions to the *HANDSHAKE_PROCESS* state. The client then begins the TLS handshake process with the server as already introduced earlier in Figure 6. Note that the entire handshake process is transported through the *EAP-Request* and *EAP-Response* packets respectively for the authenticator and the supplicant. In other words, the TLS client must be the peer sending an *EAP-Response* to complete the transaction. Therefore, as the handshake process completes, if the TLS client has no more data to send, it simply sends an *EAP-Response* with the *TLS-Ack* flag set. Finally, the state machine transitions to the *AUTHENTICATED* state when an *EAP-Success* is received.

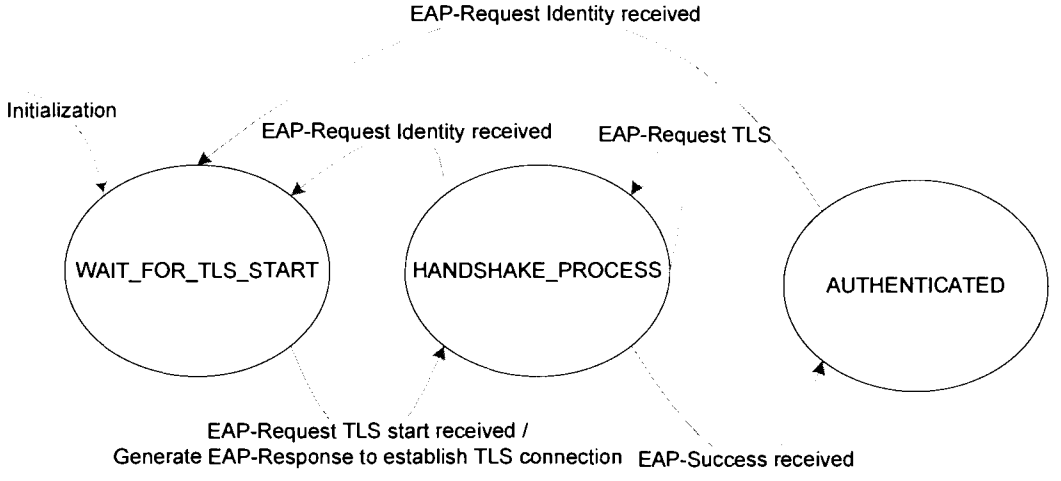

Figure 17: EAP-TLS state machine

3.5 Summary

Based on the various design considerations, the supplicant module was designed and implemented for the *PhonexChange* software on the IP phone reference platform with the role of an 802.1X compliant supplicant. The implementation was then tested with different test cases to ensure proper functionality. The test setup, test scenarios and results are described and discussed in the next chapter.

Chapter 4: Testing

The goal of testing for this project is to verify that the implemented supplicant is functional on the reference IP phone platform. To verify the supplicant's functionalities, a simple 802.1X framework was setup. The setup consists of an authentication server, an authenticator, a supplicant (the reference IP phone) and an observer. Test cases were then manually applied to the supplicant and the results were observed and analyzed.

The test cases can be classified into three main categories: unit tests, performance tests and interoperability tests. Unit tests were conducted to ensure that the implementation follows supplicant state machine operations. Performance tests were performed to ensure that supplicant is operational with the system load and memory availability. The implementation was also compared with other supplicants to confirm that its performance is competitive. Finally, a limited interoperability test with another authentication server was conducted

In this chapter, the test setup and tools would first be introduced. The procedures and results for each test categories would then be described and presented.

4.1 Test Setup and Tools

A simple 802.1X framework was setup for testing and is shown in Figure 18. The setup contains an authentication server (a PC with the RADIUS server installed), an authenticator (a managed switch), a supplicant (the reference IP phone) and an observer (PC equipped with a network analyzer). Commercial equipments were chosen to be used instead of simulators sending and receiving scripts because commercial equipments resemble a real enterprise network. Commercial equipments also ensure that the implementation is functional and interoperable, at least with the equipments tested. With scripts, for example, the server password check could be wrongly written that it never fails. Furthermore, commercial equipments are relatively easy to setup. However, the disadvantage is that only features available for the equipments can be used which could

hinder some of the test scenarios or setup, for example, when customer reports an interoperability problem using another authenticator with specific timing.

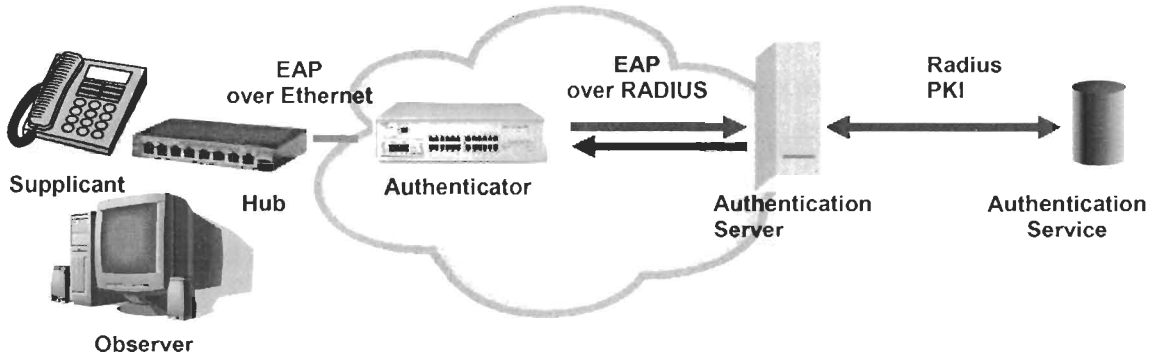

Figure 18: Supplicant test setup

4.1.1 The Authentication Server

The authentication server is a PC with the *Microsoft Windows Server 2003* OS installed and the RADIUS server service configured. The EAP-MD5 or EAP-TLS authentication method can be chosen to be used with the credentials configured. A CA service was also installed to issue a certificate / key set to the phone prior to testing. The authenticator server owns the root certificate, which can be configured to be part of the phone's trusted certificate list for peer authentication.

4.1.2 The Authenticator

The authenticator is a *D-Link* Layer-2 managed switch (DES-3226). The switch is an enterprise grade 24-port managed switch that contains Level 2 (Ethernet / 802.1 level) features such as 802.1X (access control), 802.1pq (QoS), etc. Bridging the authentication server and the supplicant, the switch was configured to require 802.1X authentication for all its downstream ports, including the port that is connected the IP phone. The RADIUS server is connected to the upstream port and does not require authentication. The current state of the supplicant can be audited through the web server on the switch.

4.1.3 The Supplicant

The supplicant is a reference IP phone that includes the 802.1X supplicant software. Recall the IP phone contains a 2-port Ethernet switch that is intended to connect to the LAN for one port, and optionally to a PC (or a network of PCs) for the other. The IP phone is connected to the managed switch from its LAN port via a hub. The purpose of the hub is to bypass the managed switch and to allow the observer PC to communicate with the IP phone even without being authenticated.

4.1.4 The Observer

The observer is a PC with the *Microsoft Windows XP* OS with the network analyzer *Ethereal* installed. The observer can be setup in two different ways depending on the situation. For the first case, the PC is connected to IP phone and the managed switch via an 8-port hub[6]. When a hub is used, unlike a switch which sends traffic only to the destined port(s), traffic is broadcasted to all other ports including the observer PC. The hub allows the observer PC to capture the network traffic between the IP phone and the managed switch, particularly the 802.1X network traffic. A very similar setup was used during development as it allows the phone to download the binary image for the application using TFTP via the hub without being authenticated. The phone can also be provisioned this way without being authenticated.

For the second case, the PC is connected directly to the PC port of the switch on the IP phone (supplicant) so that the 802.1X activities through the PC port is recorded. This setup is useful, for example, when verifying that EAP packets from the supplicant on the phone are sent only to the LAN port.

---

[6] A hub contains a number of interconnected ports. When an Ethernet packet arrives at one port, it is copied to the other ports so that all segments in the LAN receive the packet. Hub is an old Ethernet switching technology that can be inefficient because of all the collisions and is now being replaced by switches. On the other hand, because of the nature of hub that packets are copy to all ports, it is useful for network packet capturing.

4.1.5 Ethereal

Ethereal is a network packet analyzer that captures network traffic. This open source software runs on *Windows*, *Linux* and other flavors of operating systems. Depending on the settings, *Ethereal* can capture packets in promiscuous mode where all packets that the network interface card receives are captured. With *Ethereal*, all the packets between the authenticator and the supplicant can be captured and time stamped. Figure 19 shows a filtered network capture showing the EAP-MD5 authentication handshake between the supplicant and the authenticator. *Ethereal* serves as the primary tool for verifying the supplicant software, as well as a useful debugging tool to aid development.

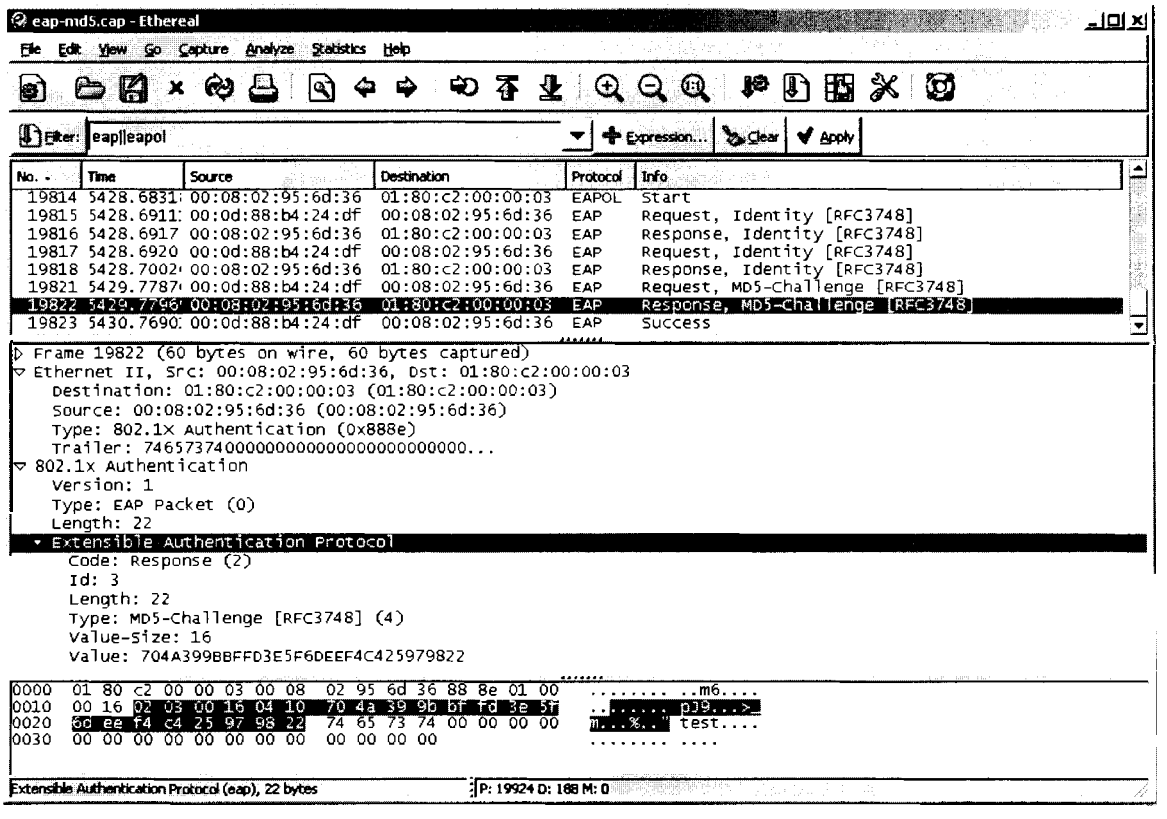

Figure 19: Ethereal showing a network capture for EAP-MD5

4.1.6 HyperTerminal

In additional to using *Ethereal*, the serial port output from the IP phone is displayed using *HyperTerminal* on the observer PC. In particular, all the state change event of the

supplicant module is logged for analysis. Also *VxWorks* shell commands can be executed from *HyperTerminal* for performance profiling.

4.2 Unit Tests

The unit tests were primarily used to verify that the supplicant implementation conforms to the 802.1X specification and the EAP operations. The unit tests also serve as a baseline for the supplicant feature as the IP phone software continues to evolve in the future. For each test, specific steps were performed with its network trace captured by *Ethereal*. The trace was then analyzed manually to verify that the supplicant implementation behaved expectedly. For example, at start up, it is expected that the supplicant would send out the *EAPOL-Start* packet for *maxStart* (3 by default) number of times that are *startPeriod* (5 seconds by default) apart. Table 6 contains the list of test cases used for testing. Appendix D describes the procedure and expected observation of each test case in detail. The supplicant implementation passed all the unit tests.

Table 6: Unit test cases

Test Case	Description
1	Authentication using EAP-MD5 (correct credentials).
2	Authentication using EAP-MD5 (incorrect credentials).
3	Authentication using EAP-TLS (correct credentials).
4	Authentication using EAP-TLS (incorrect credentials).
5	Configure the supplicant and authentication server to use different EAP methods.
6	Authentication with a wrong identity.
7	Supplicant without an authenticator.

Although it is sufficient to execute the test cases and verify the results manually, the verification process can be tedious and error prone. Automated test with the aid of a scripted authenticator may be implemented in the future to enhance the process.

4.3 Performance Tests

Performance tests are divided into two portions. First, the performance of the supplicant implementation was profiled to ensure that it performs within the memory and CPU usage performance requirements. The memory and CPU usage would also be compared against the same system without the supplicant software. The time performance of supplicant is then measured and is compared with two other supplicant implementations.

4.3.1 Memory Performance

The memory usage was measured using *VxWorks* console command *memShow*. An example console output of *memShow* is shown in Figure 20. This command shows the available system heap memory in bytes for dynamic memory allocation (e.g., **malloc()** in *C*) and the heap memory that is already allocated. Since the SDRAM is shared, the amount of system heap memory actually varies based on the size of the code and static data memory consumption; the larger the code or static data size, the less system heap memory is available. In other words, when referencing the heap memory numbers with the same software without the supplicant, the memory foot print of the supplicant software can be determined.

```
    -> memShow
 status     bytes     blocks   avg block  max block
 ------ ---------- --------- ---------- ----------
current
   free    1679520         6     279920    1593568
  alloc    1410704       912       1546          -
cumulative
  alloc    2441760      1105       2209          -
```

Figure 20: The *memShow* command on *VxWorks* showing the memory consumption with the supplicant software

By executing the *memShow* command within the application, the memory consumption was measured every 200 ms with and without the supplicant software[7]. The peak memory consumptions with the various setup were recorded, which is shown in Table 7.

[7] A sampling rate is chosen because the authentication process can vary from 2 seconds up to 15 seconds to complete. 200ms appears to be a reasonable sampling rate.

Three images were built for the comparison. For the first and second configurations, an image was built with full 802.1X support including the *OpenSSL* library. For the third and fourth configurations, an image was built without EAP-TLS support (and hence the *OpenSSL* library was not included). For the last configuration, the 802.1X software was not included in the image. Note that each image has the same total heap memory regardless of the configuration because they consume the same amount static and code space. Using the figures in Table 7, it is possible to determine the memory footprint of the supplicant software. For example, the static data and code size of the supplicant software can be found by subtracting the heap size of the full supplicant software (*3,084,512 bytes*) with the heap size without it (*3,876,688 bytes*). Table 8 summarizes the results.

Table 7: Peak allocated memory in bytes with different 802.1X configurations

	Free Memory (bytes)	Allocated Memory (bytes)	Total Heap (bytes)
EAP-TLS	1,467,712	1,616,800	
EAP-MD5 (with *OpenSSL*)	1,646,416	1,438,096	3,084,512
Disabled (with *OpenSSL*)	1,658,752	1,425,760	
EAP-MD5 (no *OpenSSL*)	2,389,456	1,438,096	
Disabled (no *OpenSSL*)	2,401,792	1,425,760	3,827,552
Disabled (no 802.1X)	2,450,928	1,425,760	3,876,688

Table 8: Memory footprint of the supplicant software

	Authentication Method	Static Data, Code (bytes)	Dynamic (bytes)	Total (bytes)
Full supplicant	EAP-TLS	792,176	191,040	983,216
Full supplicant	EAP-MD5	792,176	12,336	804,512
Full supplicant	Disabled	792,176	0	792,176
Supplicant (no *OpenSSL*)	EAP-MD5	49,136	12,336	61,472
Supplicant (no *OpenSSL*)	Disabled	49,136	0	49,136

Note that the full supplicant software is as much as *934kB* larger than the supplicant without EAP-TLS support. That is solely because of the *OpenSSL* library. As discussed earlier, the EAP-TLS implementation accesses the *OpenSSL* library through a wrapper. If the large memory footprint becomes an issue in the future, the *OpenSSL* library can easily be replaced with a smaller TLS solution.

4.3.2 CPU Performance

The CPU % usage is measured using the Idle CPU Profiler (ICP) tool developed in-house. The profiler measures the number of CPU ticks that the CPU stays idle during the time of measurement. Essentially, the profiler enters a training stage at startup keeping track of the number of ticks it runs. This counter is treated as the reference count for the rest of the process. The profiler then creates a task using the lowest priority of the OS. The task contains a simple loop with a counter that increments as it runs. Since the task is the lowest priority of the system, it is executed only when the CPU is idle. Based on the reference and idle counters, the % of CPU usage can be calculated. For example, an idle counter of 300 and a reference count of 100 indicate that the CPU is idle for 30% within the duration. The increase in task context switch has certainly added extra overhead to the CPU. Experiment shows that the extra overhead takes up approximately 0.5% of the CPU cycle.

Using the ICP, the CPU % was measured every 200 ms with and without the supplicant software. The peak CPU consumption is roughly 45.87 % and 46.25 % with EAP-MD5

and EAP-TLS respectively comparing with 45.77 % with the supplicant disabled, which are well within the requirement. The result also shows that the CPU consumption of the supplicant software is negligible.

4.3.3 Time Performance

In this test, the timing performance of the supplicant software is measured. In particular, the time it takes to successfully complete the 802.1X authentication process with the two EAP methods were measured based on the network capture. In order to make the time performance measurement meaningful, the supplicant software was compared against the same authentication process with two other implementations – one supplicant runs on the observer using a third party supplicant developed by *Meetinghouse*, and the other supplicant runs a *Linux* device using the open source *x-supplicant*.

Table 9 lists the measured time performance with the EAP-MD5 and EAP-TLS methods. The results with the EAP-MD5 method are very close while the results with the EAP-TLS method vary possibly due to different number of TLS handshake operations. Furthermore, upon analyzing the network capture, it was found that much of the delay comes from the RADIUS server. It typically takes less than 10ms for the supplicant to respond whereas it takes as much as 1.5s for the RADIUS server to respond.

Table 9: Time performance of the EAP-MD5 and EAP-TLS methods with different supplicant software

	EAP-MD5	EAP-TLS
IP phone supplicant	2.09s	10.04s
Windows XP / Meetinghouse	2.28s	7.31s
Linux / x-supplicant	2.12s	8.97s

4.4 Interoperability Tests

The purpose of interoperability tests is to exercise the supplicant software with different brands of 802.1X switches and RADIUS servers (authentication server) to ensure that the

supplicant is interoperable. At this moment, only another RADIUS server, the *FreeRadius* that runs on the *Linux* OS, was setup for testing. It is expected that more devices would be added to the profile in the future.

Chapter 5: Conclusion

The 802.1X standard combined with the EAP defines a framework to provide access control for devices. Client devices known as supplicants (in 802.1 X terminologies) connect to an 802.1X aware network require authentication from the RADIUS server, via an 802.1X aware switch.

In this project, the supplicant software was implemented for the *BCM91101* reference IP phone. The interfaces and internal architecture design of this software were based a list of design requirements to make it both modular and suitable for the *PhonexChange* software. Finally, the supplicant software was functionally tested and performance tested to make sure it is up to standard and meets the requirement specified.

As mentioned in the previous chapter, the test results show that the supplicant software was successfully implemented. The supplicant matches the requirements with a peak memory consumption of *983kB* with the supplicant with the EAP-TLS method and *61kB* with only the EAP-MD5 method. The CPU usage of the supplicant software is slightly less than 1%, which is negligible and does not interfere with the rest of the *PhonexChange* software.

In future releases, the test methodologies can be further enhanced such as automated testing in a simulated 802.1X environment, as well as introducing different reference authenticator and authentication server devices and/or software to improve the supplicant interoperability.

Although the supplicant software was successfully implemented, much work remains to be done. There has been interest to implement an authenticator for the two-port switch on the *BCM1101*. Such configuration must be carefully evaluated for its impact with the supplicant software. Furthermore, the emerging PEAP method backed by *Microsoft*,

Cisco, and *RSA Security* appear to be gaining popularity. It is unquestionable this emerging method should be considered in future releases of the supplicant software.

References

[1] "802.1X: Port-Based Network Access Control", *IEEE*, October 2000.

[2] Blunk, Vollbrecht , "RFC 2284: PPP Extensible Authentication Protocol (EAP)", *IETF*, March 1998.

[3] Rivest, "RFC 1321: The MD5 Message-Digest Algorithm", *IETF*, April 1992.

[4] Dierks, Allen, "RFC 2246: The TLS Protocol Version 1.0", *IETF*, January 1999.

[5] Zorn, Cobb, "RFC 2433: *Microsoft* PPP CHAP Extensions", *IETF*, October 1998.

[6] Frier, Karlton, and Kocher, "The SSL 3.0 Protocol", *Netscape Communications Corp.*, November 1996.

[7] Ip, "Performance Analysis of *VxWorks* and *RTLinux*", *Languages of Embedded Systems*, December 2001.

[8] "802.1D: Media access control (MAC) Bridges", *IEEE*, June 2004.

[9] "Protected Extensible Authentication Protocol (PEAP)", Symbol http://www.symbol.com/products/wireless/peap.html, accessed: March 21, 2005.

[10] Viega, Messier, and Chandra, "Network Security with *OpenSSL*", *O'Reilly*, June 2002.

Appendix A: 802.1X Supplicant State Machine

The supplicant state machine as specified in the 802.1X specification is shown in Figure 21. The operation of the supplicant with the different events and states is specified in Table 2.

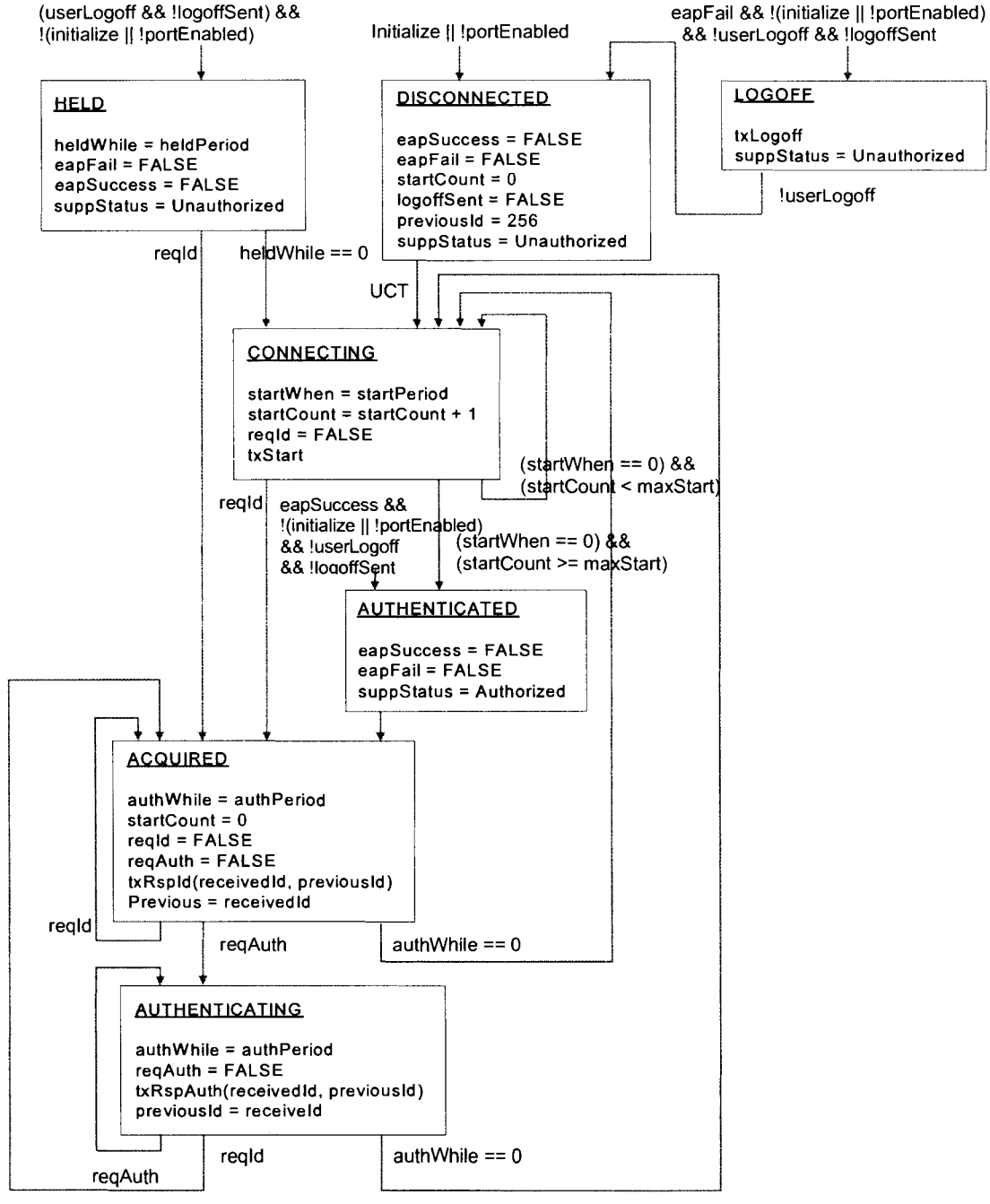

Figure 21: Supplicant state machine

Appendix B: Detailed Supplicant API Description

This appendix defines the set of command and query functions available to the client application.

supplnit

This function is used to perform global initialization of the supplicant module, including setting the EAP method to use and an event callback function which will receive asynchronous event notifications from the supplicant module. The function returns the status synchronously through the return code.

```
/*************************************************************************
*
*  suppInit -- Initialize the supplicant module
*
*  PARAMETERS:
*     eapMethod  - EAP method to use
*     suppevt     - Callback function for event notification
*
*  RETURNS:
*     SUPPSTATUS
*
*************************************************************************/
SUPPSTATUS suppInit(SUPPEAPMETHOD eapMethod, SUPPEVTCB *suppevt);
```

SUPPSTATUS Enum

```
typedef enum SUPPSTATUS
{
   SUPPSTATUS_SUCCESS,
   SUPPSTATUS_BADPARM,        /* Bad input parameter */
   SUPPSTATUS_INTERR,         /* Internal error */
   SUPPSTATUS_NOTINIT         /* Supplicant not initialized */
} SUPPSTATUS;
```

EAP Method Enum

```
typedef enum SUPPEAPMETHOD
{
   SUPPEAPMETHOD_MD5, /* EAP-MD5 method */
   SUPPEAPMETHOD_TLS  /* EAP-TLS method */
} SUPPEAPMETHOD;
```

suppSetParm

This function configures 802.1X parameters. Specifically, it provisions the EAP specific credentials such as EAP-TLS credentials, as well as letting the client application to clear the collected 802.1X statistics. Please see Table 10 below for the acceptable parameter types for **suppSetParm()**.

```
/*************************************************************************
*
*  suppSetParm -- Configure 802.1X parameters
*
*  PARAMETERS:
*     type  - Parameter type to set
*     val   - Pointer to the value to set
*
*  RETURNS:
*     suppSTATUS
*
*************************************************************************/
SUPPSTATUS suppSetParm(SUPPPARM type, void *val);
```

Table 10: Acceptable parameter types for suppSetParm()

Parameter Id	Description
SUPPPARM_MD5CREDITS	Set EAP-MD5 credentials (must be set before **suppStart**() is called)
SUPPPARM_TLSCREDITS	Set EAP-TLS credentials (must be set before **suppStart**() is called)
SUPPPARM_CLEARSTATS	Clears the collected 802.1X statistics. The 802.1X statistics is defined in IEEE 802.1X-2001 Section 9.5.2.1.3.

EAP-MD5 Credential Structure

```
/*
** username        - user identity
** password        - md5 password
*/
typedef struct SUPPMD5CREDITS
{
   char username[SUPP_MAX_STR];
   char password[SUPP_MAX_STR];
} SUPPMD5CREDITS;
```

EAP-TLS Credential Structure

```
/*
** username        - user identity
** privatekey      - private key in PEM format (0-terminated)
** localcert       - local certificate associated with the private key in PEM
**                   format (0-terminated)
** certchain       - list of certificate chain associated with the local cert
**                   in PEM format (0-terminated)
** certchainNum    - number of certificates in the list
** trustedcerts    - list of trusted certificates (peer authentication) in
**                   PEM format (0-terminated)
** trustedcertsNum- number of trusted certificates in the list
*/
typedef struct SUPPTLSCREDITS
{
   char privatekey[SUPP_MAX_KEYLEN];
   char localcert[SUPP_MAX_CERTLEN];
```

```
   int certchainNum;
   char certchain[SUPP_MAX_CERTCHAIN][SUPP_MAX_CERTLEN];
   int trustedcertsNum;
   char trustedcerts[SUPP_MAX_TRUSTEDCERTS][SUPP_MAX_CERTLEN];
   char username[SUPP_MAX_STR];
} SUPPTLSCREDITS;
```

suppGetParm

This function is used to retrieve the stored EAP specific credentials, the 802.1X statistics, and the 802.1X status. Please refer to Table 11 for the list of acceptable parameter types for **suppGetParm()**.

```
/*****************************************************************************
*  suppGetParm - Retrieve 802.1X parameters
*
*  PARAMETERS:
*     type  - Parameter type to get
*     val   - Pointer to the value to get
*
*  RETURNS:
*     SUPPSTATUS
*
*****************************************************************************/

SUPPSTATUS suppGetParm(SUPPPARM type, void *val);
```

Table 11: Acceptable parameter types for suppGetParm().

Parameter Id	Description
SUPPPARM_MD5CREDITS	Get the provisioned EAP-MD5 credentials
SUPPPARM_TLSCREDITS	Get the provisioned EAP-TLS credentials
SUPPPARM_STATUS	Get 802.1X status. The 802.1X statistics is defined in IEEE 802.1X-2001 Section 9.5.1.1.3.
SUPPPARM_STATS	Get collected 802.1X statistics. The 802.1X statistics is defined in IEEE 802.1X-2001 Section 9.5.2.1.3.

802.1X Supplicant Status Structure

```
/* Supplicant status, 802.1X, 9.5.1 */
typedef struct SUPPSPSTATUS
{
   /* Supplicant PAE state, 802.1X, 8.5.10 */
   SUPPSTATE state;              /* Supplicant PAE state */

   /* Supplicant Constants, 802.1X, 8.5.10.1.2 */
   int authPeriod;               /* authWhile timer init value, in secs */
   int heldPeriod;               /* heldWhile timer init value, in secs */
```

```
   int startPeriod;              /* startWhen timer init value, in secs */
   int maxStart;                 /* Max number of EAPOL-Start messages */
} SUPPSPSTATUS;
```

802.1X Supplicant Statistics Structure

```
/* Supplicant statistics, 802.1X, 9.5.2 */
typedef struct SUPPSTATS
{
   unsigned int packetRx;        /* EAPOL frames of any type received */
   unsigned int packetTx;        /* EAPOL frames of any type sent */
   unsigned int startTx;         /* EAPOL Start frames sent */
   unsigned int logoffTx;        /* EAPOL Logoff frames sent */
   unsigned int respIdTx;        /* EAPOL Resp/Id frames sent */
   unsigned int respTx;          /* EAPOL Response (non-Id) frames sent */
   unsigned int reqIdRx;         /* EAPOL Request/Id frames received */
   unsigned int reqRx;           /* EAPOL Request (non-Id) frames received */
   unsigned int invalidRx;       /* Invalid EAPOL frames received */
   unsigned int lengthErrorRx;   /* EAP length error frames received */
   unsigned char lastVersionRx;  /* Last EAPOL frame version */
   unsigned char lastSrcMac[6];  /* Last EAPOL frame source MAC address */
} SUPPSTATS;
```

suppStart

This function starts the 802.1X supplicant service and binds the service with the TCP/IP stack for receiving EAP-type packets. Once the supplicant service is started, it would immediate send an *EAPOL-Start* to the authenticator (MAC destination 01:80:C2:00:00:03).

```
/*********************************************************************
*
*  suppStart -- Starts the 802.1X supplicant service
*
*  PARAMETERS:
*     None
*
*  RETURNS:
*     SUPPSTATUS
*
*********************************************************************/
SUPPSTATUS suppStart(void);
```

suppStop

This function stops the 802.1X supplicant service and unbinds the service from the TCP/IP stack from receiving EAP-type packets.

```
/*********************************************************************
*
*  suppStop - Stops the 802.1X supplicant service
*
*  PARAMETERS:
*     None
*
*  RETURNS:
*     SUPPSTATUS
*
*********************************************************************/
SUPPSTATUS suppStop(void);
```

SUPPEVTCB Event Callback

```
/*****************************************************************************
*
*   SUPPEVTCB - Notify client application state change of service
*
*   PARAMETERS:
*      prevState - previous state
*      newState  - new state
*
*   RETURNS:
*      SUPPSTATUS
*
*****************************************************************************/
typedef void (*SUPPEVTCB)(SUPPSTATE prevState, SUPPSTATE newState);
```

SUPPSTATES Enum

```
/* Current state of the supplicant PAE state machine, 802.1X, 8.5.10 */
typedef enum SUPPSTATE
{
   SUPPSTATE_DISCONNECTED,
   SUPPSTATE_LOGOFF,
   SUPPSTATE_CONNECTING,
   SUPPSTATE_AUTHENTICATING,
   SUPPSTATE_AUTHENTICATED,
   SUPPSTATE_ACQUIRED,
   SUPPSTATE_HELD,
   SUPPSTATE_MAX
} SUPPSTATE;
```

Appendix C: The TLS Client Wrapper

Because of the relatively large size of the *OpenSSL* memory footprint (roughly *~800kB* on *VxWorks* depending on the components included), some customers may choose to opt for other TLS implementations with a smaller footprint and feature set. The TLS client wrapper is created with this in mind. This section describes the API of the TLS wrapper in detail. Note that the TLS wrapper is designed to be generic so that it is not restricted to be used for 802.1X.

tlsInit

This function initializes the TLS stack and creates a TLS context. On success, an unnamed pointer to the created TLS context would be returned synchronously. This context would be used throughout the course of the TLS connection. Note the TLS wrapper owns the created context that needs to be freed through **tlsDeinit**() later when the TLS operation ends (e.g., received *EAP-Success*).

```
/*********************************************************************
*
*  tlsInit -- Initialize the TLS stack and create a TLS context
*
*  PARAMETERS:
*     context  - Pointer to the unnamed TLS context to be created
*
*  RETURNS:
*     int - 0 - success, otherwise failure
*
*********************************************************************/
int tlsInit(void **context);
```

tlsConfig

This function configures the created TLS context. Essentially it provisions the private key / certificates to be used for the TLS session to be created with **tlsCreate()**.

```
/*********************************************************************
*
*  tlsConfig -- Initialize the TLS stack and create a TLS context
*
*  PARAMETERS:
*     context  - Pointer to the created TLS context
*     config   - TLS configuration block
*
*  RETURNS:
*     int - 0 - success, otherwise failure
*
*********************************************************************/
```

```
int tlsConfig(void *context, TLSCFG *config);
```

Note this function must be called before the TLS session is created with **tlsCreate()**. The content of the *TLSCFG* configuration block is copied and can be deleted once the function returns.

TLS Configuration Block - TLSCFG

```
typedef struct
{
   char *privateKey;     /* Private key in PEM format (0-terminated) */
   char *localCert;      /* Local certificate associated with
                            the private key in PEM format (0-termianted) */
   char **certChain;     /* List of certificate chain in PEM format for the
                            local certificate. The list is 0-terminated. */
   char **trustedCerts; /* List of trusted certificates in PEM format to
                            validate the server certificates */
   int  verifyPeer;      /* Indicates if the peer certificate would be
                            verified or not (1: verified, 0: bypass) */
} TLSCFG;
```

tlsCreate

This function creates a TLS session using the configured context information.

```
/*********************************************************************
*
*  tlsCreate -- Create a TLS session using the configured context
*               information
*
*  PARAMETERS:
*     context  - Pointer to the created TLS context
*     session  - Pointer to the unnamed TLS session to be created
*
*  RETURNS:
*     int - 0 - success, otherwise failure
*
*********************************************************************/
int tlsCreate(void *context, void **session);
```

Note **tlsConfig()** must be called prior to calling **tlsCreate()**.

tlsProcess

This function processes an incoming TLS packet and returns the client response. If the incoming TLS packet is NULL, the function would assume that the user would like the client to initiate a TLS connection with the provisioned context information.

```
/*********************************************************************
*
*  tlsProcess -- Process an incoming TLS packet and return the client
*                response
*
```

```
*  PARAMETERS:
*     session  - Pointer to the TLS session
*     inData   - Input TLS server packet
*     inSize   - Size of the input packet
*     outData  - Pointer to the output TLS client packet
*     outSize  - Size of the output packet
*
*  RETURNS:
*     int - 0 - success, otherwise failure
*
*************************************************************************/
int tlsProcess(void *session, unsigned char *inData, unsigned int inSize,
   unsigned char **outData, unsigned int *outSize);
```

Note that returning an outsize is 0 means that there is no TLS client response. The session handle has the ownership of *outData*, which will be deleted when the TLS session is deleted through **tlsDelete()**.

tlsDelete

This function deletes the created TLS client session.

```
/*************************************************************************
*
*  tlsDelete -- Delete the created TLS session
*
*  PARAMETERS:
*     session  - Pointer to the TLS session
*
*  RETURNS:
*     int - 0 - success, otherwise failure
*
*************************************************************************/
int tlsDelete(void *session);
```

tlsDeinit

This function frees the TLS context and cleans up the TLS stack.

```
/*************************************************************************
*
*  tlsDeinit -- Free the TLS context and cleanup the TLS stack
*
*  PARAMETERS:
*     context  - Pointer to the TLS context
*
*  RETURNS:
*     int - 0 - success, otherwise failure
*
*************************************************************************/
int tlsDeinit(void *context);
```

Appendix D: Unit Test Cases

This appendix contains the list of unit test case procedures and expected observation in detail. The unit tests were executed and verified manually although automating the unit tests has been planned for the project.

Case 1: Successful authentication using EAP-MD5

Procedure:

Configure the RADIUS server and the supplicant to use the EAP-MD5 method along with the correct credentials. Unplug and plug the Ethernet connection between the hub and the switch (authenticator) to trigger the beginning of the 802.1X process.

Observations:

- Authenticator should generate an *EAP-Request/Identity* packet. Supplicant state transitions to *Acquired*.
- Supplicant should then generate an *EAP-Response/Identity* packet with the Identity set to the provisioned value.
- Authenticator in response to the *EAP-Response/Identity*, initiates a challenge *EAP-Request* with MD5 as the EAP type. Supplicant state transitions to *Authenticating*.
- Supplicant should generate an *EAP-Response* to the MD5 challenge.
- Authenticator generates *EAP-Success*. Supplicant state transitions to *Authenticated*.

Case 2: Successful authentication using EAP-TLS

Procedure / Description:

Repeat Case 1 except configure the RADIUS server and the supplicant using the EAP-TLS method.

Case 3: EAP-MD5 authentication with the wrong credentials

Procedure:

Configure the RADIUS server and the supplicant to use the EAP-MD5 method along with the wrong credentials (e.g. wrong password). Unplug and plug the Ethernet connection between the hub and the switch (authenticator) to trigger the beginning of the 802.1X process.

Observations:

- Authenticator should generate an *EAP-Request/Identity* packet. Supplicant state transitions to *Acquired*.
- Supplicant should then generate an *EAP-Response/Identity* packet with the Identity set to the provisioned value.
- Authenticator in response to the *EAP-Response/Identity*, initiates a challenge *EAP-Request* with MD5 as the EAP type. Supplicant state transitions to *Authenticating*.
- Supplicant should generate an *EAP-Response* to the MD5 challenge.
- Authenticator generates *EAP-Failure*. Supplicant state transitions to *Held*

Case 4: EAP-TLS authentication with the wrong credentials

Procedure / Description:

Repeat Case 3 except configure the RADIUS server and the supplicant using the EAP-TLS method.

Case 5: Use a different EAP method for the supplicant and authentication server

Procedure:

Set the EAP method of the supplicant to EAP-TLS, and the EAP method of the authentication server to EAP-MD5.

Observation:

- Authenticator should generate an *EAP-Request/Identity* packet. Supplicant state transitions to *Acquired*.
- Supplicant should then generate an *EAP-Response/Identity* packet with the Identity set to the provisioned value.
- Authenticator in response to the *EAP-Response/Identity*, initiates a challenge *EAP-Request*

with MD5 as the EAP type. Supplicant state transitions to *Authenticating*.
- Supplicant should generate an *EAP-NAK*. Supplicant state transitions to *Acquired*.
- The Authenticator may reinitiate an *EAP-Request* another authentication method.

Case 6: Authentication with the wrong identity

Procedure:
Setup the supplicant with a different credential than what the RADIUS server expects.

Observation:
- Authenticator should generate an *EAP-Request/Identity* packet. Supplicant state transitions to *Acquired*.
- Supplicant should then generate an *EAP-Response/Identity* packet with the Identity set to the provisioned value.
- Authenticator should not respond to the *EAP-Response/Identity*.

Case 7: Supplicant without an authenticator

Procedure:
Unplug the authenticator from the hub, and then start the supplicant.

Observations:
EAPOL-Start packets should be sent *maxStart* times with *startPeriod* apart. Supplicant state transition from *Disconnected* to *Connecting*, then from *Connecting* to *Authenticated*