2006-06-30

Brand X, A Cross-Layer Architecture for Quality of Transport (QoT)

Gregory Arthur De Hart
Brigham Young University - Provo

BRAND X, A CROSS-LAYER ARCHITECTURE FOR QUALITY OF TRANSPORT (QOT)

by

Gregory A. DeHart

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2006

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Gregory A. DeHart

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Charles D. Knutson, Chair
Date	Daniel Zappala
Date	Eric G. Mercer

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Gregory A. DeHart in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Charles D. Knutson
Chair, Graduate Committee

Accepted for the Department

Tony R. Martinez
Department Chair

Accepted for the College

Thomas W. Sederberg
Associate Dean,
College of Physical and Mathematical Sciences

ABSTRACT

BRAND X, A CROSS-LAYER ARCHITECTURE FOR QUALITY OF TRANSPORT (QOT)

Gregory A. DeHart

Department of Computer Science

Master of Science

Computing devices are commonly equipped with multiple transport technologies such as IrDA, Bluetooth and WiFi. Transport switching technologies, such as Quality of Transport (QoT), take advantage of this heterogeneity to keep network sessions active as users move in and out of range of various transports or as the networking environment changes. Autonomous transport switching technologies rely on information regarding current network status and the ambient wireless environment in order to make intelligent decisions. This thesis proposes Brand X, a cross-layer architecture designed for a QoT environment to provide timely and accurate environment information in order to facilitate autonomous transport switching. This thesis also presents a performance analysis of network protocol stack latency in a QoT environment considering the various cross-layer mechanisms utilized in Brand X and other architectures.

ACKNOWLEDGMENTS

I would like to thank my wife for her patience and understanding throughout the seemingly never ending research process. As my friends are fond of pointing out, I definitely overachieved when I married you.

To my advisor, Dr. Knutson, thank you once again for all of your insightful commands and guidance throughout the research process. Thank you for time spent editing and helping to make my writing more technical and punchy.

I would like to thank Dr. Zappala for his ability to clearly see the obstacles that I would face in this research and for pointing out a better path.

Lastly, for those long nights when everything seemed stacked against me I would like to thank my keyboard for being the only part of my computer that never let me down. Day after day it worked just as expected, unlike the rest of the infuriating machine.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There is an increasing trend for mobile devices to be equipped with multiple wireless transceivers, such as IrDA infrared, Bluetooth, Wi-Fi or cellular, which we refer to as *intra-device transport heterogeneity*. Current mobile communication architectures provide inadequate support for intra-device transport heterogeneity in which a single device is equipped with multiple wireless transports [1]. Rather, applications are generally bound to a specific application-layer or session-layer protocol and thus to a particular transport. As a consequence, applications are forced to communicate through a single transport when multiple transport options exist between devices. Architectures in which applications are bound to a particular session layer protocol and its corresponding lower layers are commonly referred to as *stovepipe* architectures. Such architectures render applications inaccessible whenever the transport they rely on is unavailable, even if another transport could establish an acceptable link to the desired endpoint.

This binding between applications and transports has motivated the development of transport switching technologies that provide the capability to maintain session connection integrity despite changes at the transport connectivity level. The Quality of Transport (QoT) project [2] at Brigham Young University has sought to resolve the problem of transport-bound applications by enhancing the capability of such devices to intelligently utilize multiple transports. The goal of QoT is to provide a synergistic solution to intra-device transport heterogeneity through intelligent autonomous transport switching and multi-transport multiplexing to increase connectivity and transport utilization without increasing complexity at the session, application, or transport layers.

One of the significant challenges in implementing intelligent and autonomous transport switching is that effective decision making is often dependent on the availability of environmental information not generally accessible outside the layer in which such information is directly discerned. One way of solving this problem is to use a cross-layer architecture. Cross-layer mechanisms can be employed to gather environment information from individual protocol layers, without requiring a complete redesign of the protocol stack or a compromise of layered architectural principles.

Cross-layer architecture is a relatively new area of research that does not benefit from a large body of established results, well-tested design methodologies, or well-constructed validation and verification procedures. Previous research in cross-layer architectures focused on the ability to improve network throughput or maintain connectivity in adverse conditions without addressing the local network protocol stack effects of those mechanisms. Our research indicates that the type of mechanism utilized in a cross-layer architecture has a potentially large impact on the performance of the local network protocol stack.

1.1 Thesis Statement

This thesis presents Brand X, a cross-layer feedback mechanism for supplying environmental information from multiple protocol stack layers to the autonomous transport switching algorithms of QoT. This research facilitates the enhancement of QoT by enabling dynamic, configurable, autonomous environment information gathering. This thesis also presents the results of an empirical performance analysis of network protocol stack latency in a QoT environment resulting from Brand X and other cross-layer architectures.

1.2 Thesis Layout

The remainder of this thesis is outlined as follows: Chapter 2 consists of a journal paper currently in preparation for submission. This paper describes the design and features of Brand X and a detailed description of the cross-layer performance analysis. This paper also includes supporting research in cross-layer taxonomies resulting from our work with

Brand X. Chapter 3 includes extended tables and data sets derived from the analysis of variance (ANOVA) performance evaluation of Brand X and other cross-layer architectures.

Appendix A is the Cross-Layer Test Harness documentation, which details our implementations of Brand X, Synchronous Push and Synchronous Pull cross-layer architectures. Documentation is also included for the supporting test harness software and the modifications made to the Linux TCP/IP and UDP protocol stacks.

Chapter 2

Brand X, A Cross-Layer Architecture for Wireless Intra-Device Heterogeneity

2.1 Introduction

Recent trends in wireless heterogeneity are yielding mobile computing devices equipped with multiple transport mechanisms (including IrDA, Bluetooth, Wi-Fi, and cellular). Current mobile device architectures provide inadequate support for *intra-device heterogeneity* in which a single device is equipped with multiple wireless transports[1] [1]. Rather, applications are generally bound to a specific application-layer or session-layer protocol and thus to a particular transport. Architectures in which applications are bound to a particular session layer protocol and its corresponding lower layers are commonly referred to as *stovepipe* architectures. Such architectures render applications inaccessible whenever the transport they rely on is unavailable, even if another transport could establish an acceptable link to the desired endpoint.

The Quality of Transport (QoT) project [2] at Brigham Young University has sought to resolve the problem of transport-bound applications by enhancing the capability of such devices to intelligently utilize multiple transports. The goal of QoT is to provide a synergistic solution to intra-device transport heterogeneity by providing intelligent, autonomous transport switching and multi-transport multiplexing to increase connectivity and transport

[1] By "transport" we refer broadly to traditional stovepipe communication architectures interfaced primarily via the transport layer of the protocol stack. Hence, when we use the term "transport," we refer to all layers from the transport layer to the physical layer inclusive. As an example, we would refer to IrDA, Bluetooth and Wi-Fi as separate "transports."

utilization without increasing complexity at the session, application, or transport layers. QoT alleviates the constraints of stovepipe architectures by directing data flow through the most desirable available transport in a manner transparent to the application and session layers while applications continue to send data through the session protocols for which they were designed. Figure 2.1 shows an example of QoT using IrDA as the best available transport for a Bluetooth application.

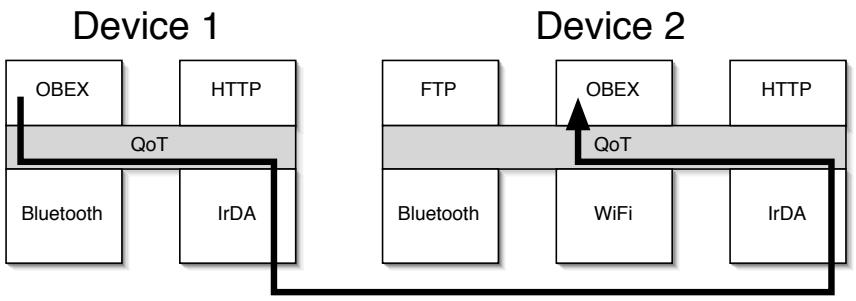

Figure 2.1: Quality of Transport (QoT) Data Exchange.

One of the significant challenges in implementing an architecture such as QoT is that effective transport selection is often dependent on the availability of environmental information not generally accessible outside the layer in which such information is directly discerned. Protocol stacks are designed utilizing a layered approach in order to limit coupling between layers and allow independent system development while retaining compatibility. Layered protocol stacks limit the accessibility of environment information outside the source protocol layer. This masks information that is necessary for intelligent decision making. Examples of environment information useful for intelligent network decision making but not available outside of the source protocol layer include signal strength, lost packets, interference and radio noise.

One way of solving this problem is to use a cross-layer architecture. Cross-layer mechanisms can be employed to gather environment information from individual protocol

layers, without requiring a complete redesign of the protocol stack or a compromise of layered architectural principles.

In this paper we present Brand X, a hybrid cross-layer architecture designed for the QoT multi-transport environment. Brand X employs a Data Store cross-layer mechanism, which enables loosely coupled asynchronous communication between protocol layers. The asynchronous data store facilitates information queries across multiple transports while providing simple and configurable cross-layer data retrieval.

We demonstrate that an implementation of Brand X in a Linux kernel can provide fully configurable information retrieval. We evaluate the performance of Brand X and compare its performance to several other cross-layer architectures. The performance analysis results show that Brand X out performs synchronous cross-layer mechanisms by as much as 47.6% on incurred protocol stack latency.

The remainder of this paper is structured as follows. Section 2.2 outlines previous work in the area of cross-layer design and analysis. Section 2.3 presents a cross-layer taxonomy we have developed to aid in the comparison of cross-layer architectures based on their functional attributes and performance results. Section 2.4 is a detailed description of Brand X, a cross-layer architecture for the QoT environment. Section 2.5 presents our cross-layer performance study and results. In Section 2.6 we present our conclusions and future work.

2.2 Related Work

Research in cross-layer architectures has historically taken one of two forms: 1) design and implementation of cross-layer architectures for specific environments; 2) analysis of performance improvements due to application of cross-layer mechanisms. In the following sections we present an overview of common cross-layer architectures. We then summarize research conducted on the performance enhancements of cross-layer architectures and the potential problems arising from improper use of cross-layer mechanisms. We conclude this section by presenting an overview of the Quality of Transport research.

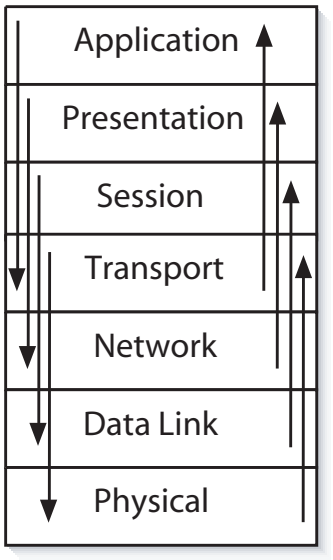

Figure 2.2: Piped Message Cross-Layer Architecture.

2.2.1 Cross-Layer Designs for Specific Environments

Architectures for cross-layer mechanisms can be generalized into three categories: Piped Message, Data Store and Inter-Layer Signaling. We briefly discuss each of these approaches in turn.

2.2.1.1 Piped Message

Piped Message architectures, (see Figure 2.2), require the modification of one or more protocol layers to gather environment information and transmit it to other layers via the protocol stack. Environment information is detected at appropriate protocol layers and assembled into cross-layer packets, which are injected into the protocol stack in the direction of the recipient layer. The recipient layer recognizes the cross-layer packet, reads the environment information contained therein and utilizes the information to influence its own decision making.

Several implementations of Piped Message have been proposed. Montenegro and

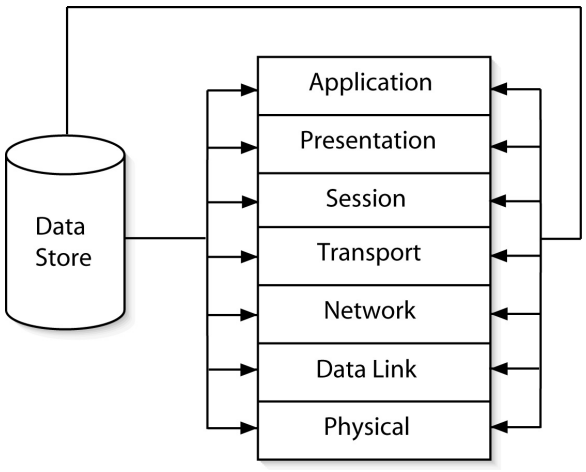

Figure 2.3: Data Store Cross-Layer Architecture.

Drach [3] use Internet Control Message Protocol (ICMP) [4] messages to notify an operating system of a lost connection. Their architecture is very limited in the scope of the information it is able to transmit. Furthermore, their dependence on the ICMP protocol does not allow for implementation on a wide range of devices (such as handhelds). Sudame and Badrinath extend the design of Montenegro and Drach to propagate network environment information throughout the stack [5]. Their extensions include the idea of notable events[2] and the capability of communicating a wide array of environment information. Wu, et al., present an implementation of Interlayer Signaling Pipes that relies on the Wireless Extension Headers (WEH) of IPv6 [6].

2.2.1.2 Data Store

Data Store architectures, (see Figure 2.3), share the common characteristic that one or more layers of a given protocol stack write information to a centralized location. When new network status information becomes available it is recorded in the data store where it can be accessed by interested protocol stack layers. It is the responsibility of each layer of a protocol stack to make available through the data store the results of a notable event.

[2] A notable event is a change in the environmental conditions or stack performance that causes the state of the system to cross one or more predetermined thresholds.

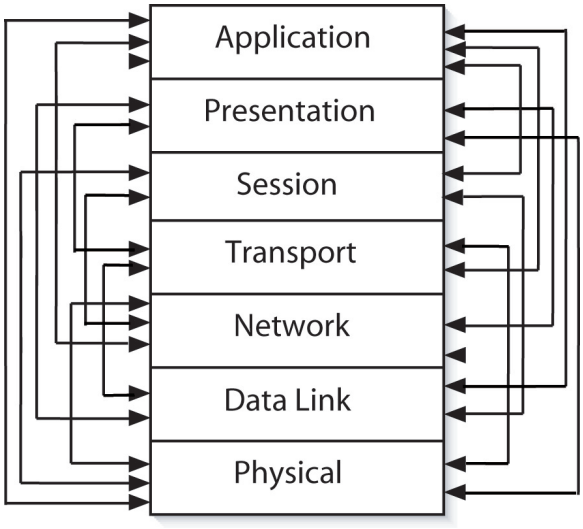

Figure 2.4: Inter-Layer Signaling Cross-Layer Architecture.

Through the use of such a mechanism Harter, et al., were able to reduce the number of writes to a centralized database for a context aware system by more than 90% [7].

Several Data Store architectures have been proposed. Chen, et al., discuss the use of a cross-layer data store in an ad hoc network environment [8]. The cross-layer mechanism provides additional communication between middleware and routing layers to allow faster look-up of media services and higher quality streaming of that data. Clark and Tennenhouse propose the use of a single field which could be accessed by all protocol layers or modules in order to improve performance [9].

2.2.1.3 Inter-Layer Signaling

Inter-Layer Signaling architectures, (see Figure 2.4), support bi-directional data communication between non-neighboring layers, reducing the propagation latency common in other cross-layer approaches. This structure was first proposed by Wang et al. in the design of Cross-Layer Signaling Shortcuts (CLASS) [10]. The theoretical propagation latency in CLASS is only about $\frac{1}{(n-1)}$ as large compared to the Piped Message architecture, where *n* is the number of layers traversed. Wu, et al., propose Interlayer Signaling Pipe,

a communication architecture that does not require the use of standardized protocols for internal signaling, thus facilitating a lightweight internal message format [6].

For a specific protocol stack implementation, cross-layer interactions are task-dependent and protocol-specific. Inter-Layer Signaling architectures focus on communication between layers of the protocol stack. It creates unique communication channels between stack layers that exchange information. Creating such communication channels requires cooperation from the designers of the various protocol layers and reduces the inherent flexibility of a protocol stack. As a result, the complexity of system design increases and maintainability of the system decreases.

2.2.2 Cross-Layer Analysis

Raisinghani and Iyer [11] analyze the benefits of a cross-layer feedback mechanism for mobile devices. They present a representative survey of the OSI protocol layers and present examples of cross-layer feedback for each layer, discussing the benefits and disadvantages incurred. Koucheryavt, et al., [12], present an overview of recent developments in cross-layer architecture in next generation systems and outline directions of further work in performance evaluations of all-IP next generation systems. Current traffic modeling and wireless channel modeling techniques are considered and their limitations for future IP based mobile systems are addressed.

Raisinghani, et al., [13], outline two mechanisms for cross-layer feedback with TCP and model their performance benefits. These cross-layer feedback mechanisms rely on an extended implementation of TCP that allows modification to application priority through receiver window control. They show that by using their proposed mechanism they achieve an improvement of up to 150% over TCP Reno.

Fang and McDonald, [14], demonstrate that improper use of cross-layer technology can have a significant negative impact on energy efficiency, throughput, and delay. In their results, a twelve hop path with no transport contention is shown to achieve 90% throughput at 100 Kbps, dropping to 50% at 500 Kbps. When transport contention along the network path reduces system throughput to 40% at only 100 Kbps.

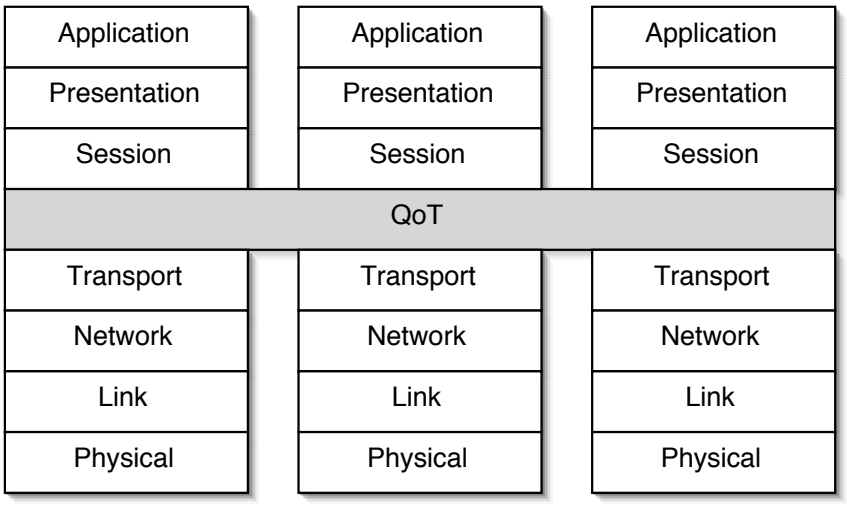

Figure 2.5: Quality of Transport (QoT) Architecture.

2.2.3 Quality of Transport (QoT)

The goal of Quality of Transport (QoT) is to facilitate dynamic, transparent and autonomous transport switching for multi-transport devices in order to provide the highest quality data transfer capability within heterogeneous wireless environments. QoT automatically manages the nature of the underlying network connection in order to maximize user experience and satisfaction. In one case maximizing a user's experience means maintaining a connection through various network environments, while in another case the user might want to optimize battery life at the expense of higher data throughput. Duffin, et al., [15], presented a qualitative method for establishing user defined constraints for QoT.

Barnes, et al., [16], introduce multi-transport discovery within the context of the QoT architecture. This paper presents transport probing as a method by which QoT may establish a communication link with each remote device. Transport querying then uses the link established during the transport probing phase to identify all of the transport capabilities of the remote device. QoT then maintains a table of device-to-address translations and transport availabilities to determine the appropriate transport to be utilized.

Knutson, et al., [17], present an overview of the QoT architecture (see Figure 2.5)

including transport discovery, service discovery, object exchange, transport switching, and intelligent transport selection. Their preliminary results suggest that intelligent transport switching can help to improve user experience and session layer performance in heterogeneous wireless environments.

2.3 Cross-Layer Taxonomy

Cross-layer architecture is a relatively new area of research that does not benefit from a large body of established results, well-tested design methodologies, or well-constructed validation and verification procedures. We have developed a conceptual taxonomy that provides insight into the ways in which cross-layer architectures are employed to solve problems, the functionality they provide, and the mechanisms on which they rely. Additionally, this taxonomy provides a foundational set of definitions by which cross-layer research can be compared and discussed.

We present the cross-layer taxonomy in two parts: attributes of cross-layer architectures and taxonomic relationships. The cross-layer attributes section contains categories that are used to classify and compare cross-layer functionality. In the taxonomic relationship section we present a hierarchical representation of the cross-layer attributes along with a description of the relationships that are observed between certain cross-layer attributes.

2.3.1 Attributes of Cross-Layer Architectures

In order to develop a useful cross-layer taxonomy, we must first consider the fundamental factors dealt with during design and operation of cross-layer architectures. These factors are common across all cross-layer architectures and provide a consistent reference for comparison. We found that cross-layer architectures are distinguished by their attributes and the functionality they employ to provide their results. These attributes and functional mechanisms make up the classification categories in the cross-layer taxonomy.

In the following sections we define terminology and categories that can be used for the classification, identification, and comparison of cross-layer architectures. We have

identified eight categories of classification that are common among cross-layer research that distinguish the mechanisms involved in cross-layer operation. These eight categories have been selected because they provide information about key aspects of cross-layer behavior.

2.3.1.1 Information Discovery

Information discovery describes the mechanism employed to collect environment information from the protocol layer in which such information is collected. Each cross-layer architecture employs some method for breaking the strict layered protocol structure and gathering the required environment information. For example, a cross-layer architecture could extend the existing TCP protocol implementation to store information about packet retransmissions as they occur. The mechanism utilized to gather information determines the invasiveness of the cross-layer architecture to the current protocol structure.

We use the following three data collection techniques, common to software monitoring systems, to classify information discovery methods used in cross-layer architectures [18]. These data collection techniques can be used to produce the same results, but they vary greatly in their environment impact and compatibility with existing software.

- *Implicit Spying* requires observing all communications between protocol layers. Implicit spying is commonly used in network traffic sniffers. The advantage of implicit spying is that there is no direct impact on the performance of the system being monitored. No changes in the protocol layers are required, but the ability to observe data as it passes between protocol layers must be guaranteed in order to obtain accurate results.

- *Explicit Instrumentation* requires direct modification of the protocol layers being monitored. Protocol layers are modified to include data calculation algorithms and reporting interfaces. Each item of environment information monitored requires modification in the protocol stack. Extensive modification of the protocol stack, resulting from extensive information gathering, increases system complexity and decreases maintainability [19].

- *Probing* requires making special 'feeler' calls to report the status of protocol layers. These calls, in the form of specially marked packets, are sent via the protocol stack and are made available to the cross-layer application. The protocol layer must be capable of capturing packets, including necessary information, and returning the packet via the protocol stack to the cross-layer information repository. Well-defined mechanisms do not exist for all transports so custom protocols and messages are required [19].

2.3.1.2 Data Elicitation Method

The *Data Elicitation Method* describes the protocol or application that initiates transfer of environment information from the originating protocol layer to the final destination. Depending on the cross-layer architecture utilized and the *freshness*[3] or timeliness of information required, the signal to transfer information can originate from the source or destination module. Each method has advantages and disadvantages and must be fully considered when developing a cross-layer architecture for a specific network environment.

There are three possible data elicitation schemes: 1) Data is pushed from the source layer to the destination layer; 2) Data is requested by the destination layer and subsequently returned by the source layer; 3) A hybrid approach is taken in which information is pushed from the source layer or requested by the destination layer depending on the current status of the system. Each of these three data elicitation methods are described below.

- *Synchronous Push* is a model in which protocol layers relay detected environment information to a pre-determined destination (commonly another protocol layer). This approach requires coordination between stack layers that exchange information. This method delegates more of the data delivery timing and overhead to the protocol layers and reduces the possibility of a bottleneck in the destination layer. Improperly designed query mechanisms can introduce tight coupling between protocol layers,

[3] By "freshness" we refer to the accuracy of stored environment information in reflecting the current state of the network environment. "Freshness" is not a simple calculation of the length of time information has been stored because as the volatility of the environment increases the length of time information can be considered accurate decreases.

and increase system complexity, causing new bottlenecks in source protocol layers. Protocol layers in a Synchronous Push architecture do not have the advantage of knowing the current network status and behave using a greedy approach calculating and sending information regardless of external conditions such as power consumption or network utilization [20].

- *Synchronous Pull* is a model in which each protocol layer requiring environment information sends a query to the appropriate observation layer and receives a response. Requests for information may be based on an immediate need, a regular time schedule, or may vary based on protocol activity. Synchronous pull retains the information request mechanisms for all protocol layers within the destination protocol layer. Without delegating data elicitation to the source protocol layers there is a greater chance for a network bottleneck at the destination protocol layer. Retaining control of information updates in the decision making protocol layer allows for intelligent scheduling of information updates based on the current network and system status. Fine tuning update scheduling decreases excessive updating with the drawback that sudden changes to the environment are not reported immediately [20].

- *Billboard* is a hybrid data elicitation method employing features of both Synchronous Push and Synchronous Pull to generate a loosely coupled asynchronous information transfer mechanism. In this approach, protocol layers discern and publish information to a local data store[4] either at some time interval, or as a result of some event or condition. The information on the Billboard can then be accessed by other layers asynchronously. In effect, observing layers publish information to the Billboard via a Synchronous Push, while consuming layers access information from the Billboard via a Synchronous Pull. This approach allows producers and consumers to effectively decouple, yielding a tunable asynchronous cross-layer system.

[4]Billboard refers to a data elicitation method and data store refers to an independent module used for data storage.

2.3.1.3 Activation Mechanism

Activation Mechanism describes the process that determines initiation of information transfer between protocol layers. Once environment information has been determined by the information discovery process, that information needs to be relayed to the destination location at some interval. For example, a cross-layer architecture closely monitoring packet corruption could signal an information update each time a packet is received containing corruption. The activation mechanism and frequency of information transfer can have a large effect on the network efficiency due to the resources consumed by the transfer mechanism. The degradation of network performance should be balanced with the need for current and accurate information.

There are several factors that influence the frequency at which data is updated by the cross-layer architecture. In a volatile network environment, information changes rapidly, requiring frequent information transfer in order for the system to make accurate decisions. Network utilization can have an effect on the appropriate frequency at which data should be updated. During periods of little or no network utilization, battery power can be conserved by updating information every few seconds. The type of network traffic can also help determine the proper information update frequency.

Two classifications of activation mechanisms, (timer driven and event driven), are common to cross-layer architectures and are described in the following sections:

- *Timer* driven mechanisms update environment information based on a regular time interval regardless of changes to network utilization or network environment. A potential side effect of timer driven mechanisms is a synchronization in update timers across the protocol layers that results in all protocol layers attempting to update at the same time. This synchronization occurs due to the limited granularity in current operating system timers. This causes a short interval of high cross-layer overhead which may introduce jitter into the network traffic latency.

- *Event* driven mechanisms establish intervals or thresholds used to determine if information should be updated. The intervals are based on information freshness requirements and provide a level of granularity in information updating. If the current environment information and the previously published information are within the same interval they are considered current and no update occurs. If the values fall into different intervals the previously published information is considered out of date and an update occurs.

2.3.1.4 Information Requirements

Information Requirements describe the environment information that each cross-layer architecture is designed to gather and utilize in order to achieve its intended goal. Environment information that can be useful in making cross-layer decisions is available in all layers of the protocol stack. The set of information that each cross-layer architecture requires is based on the intended cross-layer goal and the network environment.

Classification based on a cross-layer architecture's information requirements helps determine the intended goal of the architecture and the mechanisms that are utilized to achieve the goal. Grouping architectures based on information requirements can be used as a tool in comparing analogous cross-layer mechanisms and in designing new solutions to cross-layer problems.

2.3.1.5 Motivation

Motivation describes the specific problem or opportunity which the cross-layer architecture is designed to address. The following sections describe four categories of cross-layer architecture design motivations:

- *New Functionality* – By providing protocols and applications access to cross-layer environment information, new ideas, and directions in network communication are made possible. New techniques in MAC retransmission or transport selection based on the highest quality connection are two examples of recent research made possible

by the use of cross-layer information. Cross-layer design often occurs as a by-product of other network research and is created in an effort to overcome traditional network limitations.

- *Improved Performance* – Utilization of information from multiple protocol layers facilitates informed decisions regarding network performance. Cross-layer architectures provide a mechanism whereby access to a greater body of information is possible. Utilizing additional information available from other source protocol layers, improved networking algorithms can be created that may increase network performance.

- *Robust Data Exchange* – Utilization of cross-layer information can allow a device to provide a more robust wireless connection and improve packet error correction due to interference in the ambient wireless environment. Current robust data exchange algorithms rely on encoding schemes or signal detection at the physical layer. Cross-layer mechanisms can be used to determine packet errors or re-route traffic to avoid the problematic area.

- *Power Conservation* – Varying signal strength based on cross-layer information allows fine-tuned adjustments in the power to be made, thereby allowing a power constrained wireless mobile device to improve performance while decreasing power consumption. Mobile devices equipped with multiple transport mechanisms are becoming more common and as with any such device, battery life is an important issue. Sichitiu showed an increase in battery life from 3.2 months to 24.2 months in researching cross-layer scheduling for power efficiency in wireless sensor networks [21].

2.3.1.6 Network Environment

Network environment describes the network topology and physical transport that the cross-layer architecture utilizes. This includes application of cross-layer mechanisms on new and existing physical transports as well as current research on improved network protocols.

Categorizing the network environment for which a cross-layer architecture is designed helps to group architectures that have similar environment requirements and may help identify other possible cross-layer solutions. This aids in collaboration between cross-layer solutions and helps extend research to a broader subject domain.

2.3.1.7 Protocol Compatibility

Protocol compatibility refers to the specific protocol or protocols for which a cross-layer architecture has been designed to be compatible. We define compatibility as the ability of a protocol to utilize cross-layer functionality or the use of a protocol by the cross-layer architecture in order to provide an improvement in network communication. For instance, a cross-layer solution can be designed to optimize TCP retransmission requests whereby any application protocol utilizing TCP may experience an improvement in performance.

There are many different protocol structures for which cross-layer architectures are designed and implemented. With each presenting a unique set of compatible protocols. Cross-layer architectures can be designed to work with information from the entire protocol stack, ensuring compatibility with all layers of the protocol stack. Additionally, a cross-layer architecture can be designed to work with a single transport stack providing benefit only to applications that utilize the enhanced transport. Cross-layer architectures can also be designed to be independent of any specific protocol restrictions and remain compatible with all types of network traffic for the target network environment.

2.3.1.8 System Definition

System Definition describes the system boundaries of an architecture including affected protocol layers, utilized system resources, and external application support. Cross-layer architectures include a variety of resources depending on the availability of environment information and the cross-layer mechanisms utilized in data transfer. Typical system resources that are utilized include protocol stack layers, operating system resources, and battery power indicators.

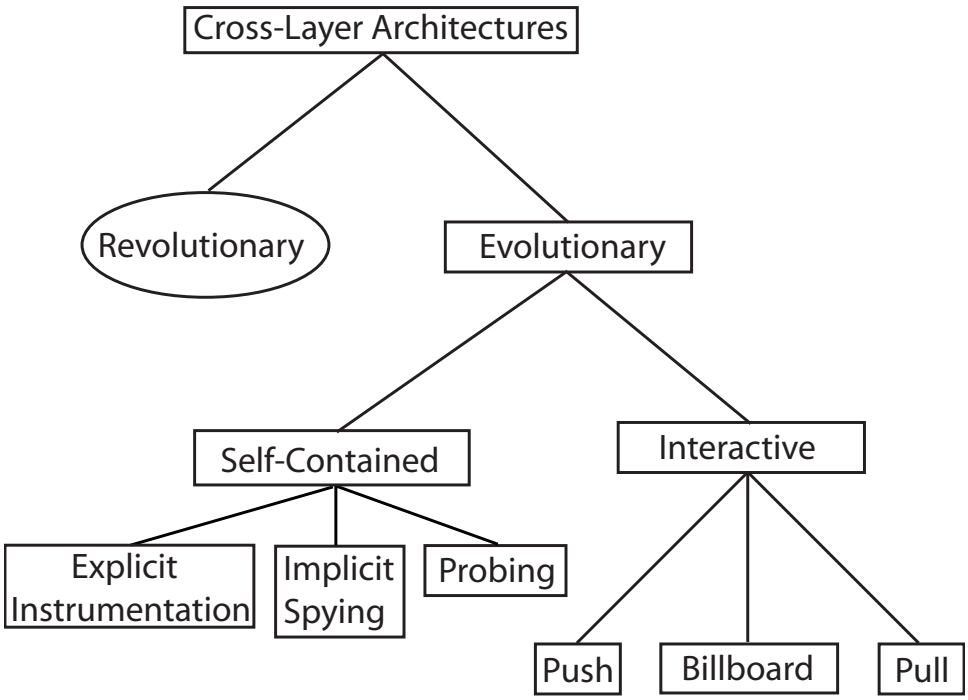

Figure 2.6: Hierarchical Representation of Taxonomic Relationships.

The following are important questions to answer when classifying an architectural system: 1) What protocol layers and system resources are utilized by the cross-layer architecture? 2) What modifications are required to the system and protocol layers? 3) Where are the decision making algorithms for the cross-layer solution located?

2.3.2 Taxonomic Relationships

Two important relationships exist between the functional cross-layer taxonomy characteristics including: 1) Evolutionary versus revolutionary design; 2) Self-contained versus multi-layer (see Figure 2.6).

2.3.2.1 Evolution versus Revolution

Researchers have generally followed one of two approaches in cross-layer research: *evolutionary* or *revolutionary*. Evolutionary is a traditional approach to cross-layer design

in which existing layered protocols are extended and backward compatibility is retained. Revolutionary is an approach in which the layered protocol stacks are removed in favor of high-performance custom protocol architectures [22].

Evolutionary design for cross-layer systems focuses on extending existing layered protocol structures in an effort to maintain backward compatibility while realizing performance and functional improvement. Adhering to strict protocol layering provides backward compatibility but also limits design flexibility and performance.

Revolutionary design is not bound by existing protocol implementations or layered design approaches. Rather, revolutionary design prioritizes performance above compatibility and removes strict layering in favor of performance. Because of the high cost of hardware and software upgrades when introducing protocol changes, abandoning strict layering and neglecting backward compatibility have limited revolutionary designs to research environments [23].

2.3.2.2 Self-contained versus Multi-layer

Cross-layer architectures are traditionally designed to transfer information from one protocol layer to another with out compromising the advantages of a layered stack structure. The mechanisms involved in transferring information between protocol layers is a topic of continuing research. Each mechanism has advantages and disadvantages that must be considered when designing a cross-layer architecture for a particular environment.

Self-contained cross-layer designs are bound by a reliance on the limited set of data available as packets are processed within the protocol layer. Toumpis and Goldsmith [24] propose two self-contained cross-layer architectures that vary energy efficiency and packing density in the Medium Access Control (MAC) sub-layer of the Data Link layer. Toumpis and Goldsmith's architectures make decisions based on cross-layer information gathered from network packet headers in order to conserve energy and increase channel utilization without requiring access to other protocol layers.

In contrast, multi-layer architectures involve direct communication between protocol layers. Knowledge of the protocol stack layers allows the cross-layer designer to take advantage of information available within all protocol layers. Leveraging information from throughout the protocol stack, advanced network algorithms can be created to improve network performance. Multi-layer architectures are able to gain the benefits of a non-layered protocol while maintaining the advantages of a layered protocol stack [25].

2.3.3 Brand X Taxonomy Classification:

The following is a taxonomic classification of Brand X in a QoT environment.

- Information Discovery: Explicit instrumentation is utilized in the Brand X architecture for information discovery. Protocol layers are modified to calculate and report various items of environment information.
- Data Elicitation Method: The Brand X architecture utilizes a Billboard data elicitation model. Environment information is pushed from the source protocol layers to the data store and QoT then pulls from the Billboard when information is required.
- Activation Mechanism: Brand X utilizes both Timer and Event driven activation mechanisms. Protocol layers report environment information using a Timer mechanism in order to reduce cross-layer overhead in network traffic processing. QoT pulls information from Brand X utilizing an Event mechanism. Brand X can configure source protocols and QoT for alternative activation mechanisms depending on the ambient wireless environment and network traffic flow.
- Information Requirements: Brand X creates a registration system for producers whereby any type of information can be registered and exchanged. No limitations on information type are enforced by Brand X.
- Motivation: The goal of the Brand X is to facilitate robust data exchange in a heterogeneous wireless environment by providing accurate environment information to the QoT autonomous transport switching algorithms.

- Network Environment: Brand X operates in a heterogeneous wireless environment encompassing multiple protocols and network transports.
- Protocol Compatibility: Brand X is designed to operate independently of any specific set of network or application protocols.
- The Brand X system includes: Brand X Core, Brand X Interface, Brand X Brain, network protocol layers extensions and QoT.

2.4 Brand X, Cross-Layer Architecture for QoT

Brand X is a cross-layer architecture specifically designed for a QoT multi-transport environment. Brand X employs a Billboard data elicitation mechanism that provides loosely coupled asynchronous data communication between producer and consumer protocols. The following sections describe the features and implementation of Brand X.

2.4.1 Brand X Features

Some of the most salient features of Brand X include: Cross-Layer Configuration, Protocol Integration, and Information Update Frequency. We briefly discuss each of these in turn.

2.4.1.1 Cross-Layer Configuration

Brand X utilizes both push and pull data elicitation mechanisms in order to interact with producer and consumer protocol layers. Communication mechanisms between protocol layers and Brand X can be configured to utilize either a push or pull mechanism. The ability to configure data elicitation mechanisms allows Brand X to efficiently respond to highly volatile environments and minimize cross-layer overhead in stable environments.

Brand X can also vary information update frequencies in response to changes in the ambient wireless environment. Adapting at the protocol level to changes in the wireless

environment minimizes system wide cross-layer overhead and provides timely information in a highly varied network environment.

2.4.1.2 Protocol Integration

Brand X utilizes a registration and notification mechanism to create an asynchronous communication mechanism between protocol layers. Protocol layers that produce environment information register with Brand X as a provider for a certain type of information, while those layers requiring cross-layer information register with Brand X as a consumer of information.

During registration, Brand X stores a callback function for the registered producer or consumer protocol. Brand X uses the callback function of the producer protocol layer to request information updates as needed, and to pass configuration information to the producer protocol. The consumer protocol's callback function is utilized by the Brand X notification mechanism to inform the consumers of updated environment information.

Brand X maintains a time stamp that is associated with each item of information. The time stamp is used by Brand X for monitoring information freshness. Once information is considered out-of-date Brand X utilizes the producer callback function to request updated environment information from the source protocol layer.

2.4.1.3 Information Update Frequency

Determining optimal information update frequencies for environment information is a complex problem involving various factors, including, ambient wireless environment, network utilization, traffic type and battery consumption. QoT and other consumer protocols determine the update frequency for information they require. Consumers have two options for information updating: 1) The consumer protocol utilizes a pull mechanism and queries Brand X for information according to the determined update frequency. 2) The consumer protocol utilizes a push mechanism and passes the update configuration to Brand X, causing Brand X to signal the consumer protocol with information updates as specified

by the update frequency.

2.4.2 Brand X Implementation

We implemented a proof-of-concept version of Brand X using the SUSE 9.2 Linux operating system with the 2.6.8-24 kernel. Utilizing an open-source Linux operating system provides access to kernel and network source code.

The Brand X implementation includes functional extensions to the native Linux network protocol layers that integrate cross-layer functionality into the existing network protocol stack. These functional extensions include data calculation functionality, a query response interface, a timing mechanism for information updates, information threshold simulation for event driven updating, and a configuration interface. The following sections detail the architecture of Brand X and highlight key functional extensions and cross-layer components.

2.4.2.1 Brand X Core

The Brand X Core provides storage for network protocol registration, environment configuration data and environment information transferred between protocol layers. Information update settings are transferred from QoT and stored in the Brand X Core for use by the Brand X Brain. This module also provides a caching mechanism for information passed between protocol layers in order to reduce the number of protocol layer interrupts necessary to service information updates.

Brand X Core utilizes a data store mechanism in which protocol layers can post or retrieve information, and configuration data. The advantage of the data store is that multiple information queries can be serviced simultaneously without interrupting the source protocol layer. This mechanism decouples the source and destination layers thus reducing overhead that results from cross-layer communication.

2.4.2.2 Brand X Brain

Brand X Brain provides autonomous information update services. Guided by information update requirements provided by QoT, the Brand X Brain utilizes environment information stored in the Brand X Core to determine future updates. Brand X Brain also configures the connection mechanisms used for communication between Brand X and the source protocol layers. Brand X Brain is responsible for managing Brand X functionality in order to maximize information exchange while minimizing cross-layer overhead.

2.4.2.3 Brand X Interface

The Brand X Interface controls data communication between Brand X and other protocol layers. This interface is used to register protocol layers, accept information updates, handle information requests, push configuration settings to protocol layers and distribute environment information as directed by the Brand X Brain. One of the primary responsibilities of the Brand X Interface is to maintain information integrity by enforcing an order to the reads and writes to the stored data.

2.4.2.4 Protocol Layer Modules

Brand X enabled protocol layers are extended with algorithms for environment information calculation, interface mechanisms, and activation mechanisms. These changes are minimally intrusive and do not alter existing functionality, thus allowing a layered protocol stack structure to be maintained. Protocol layers are additionally extended with interfaces for handling configuration and information requests. Environment information calculation algorithms are unique for each protocol layer and depend on the implementation of the protocol along with the type of information required. Protocol layers are extended with timer and event activation mechanisms that are used for initiating information updates. By maintaining a small code footprint in the protocol layers, this approach minimizes cross-layer induced overhead in protocol layer functionality.

2.4.2.5 QoT Interface Module

The QoT/Brand X interface provides a more sophisticated configuration interface in addition to the information request interface that exists between Brand X and other protocol layers. These additional configuration mechanisms are required between QoT and Brand X in order to transfer configuration settings from QoT Brain to Brand X.

No environment information determination mechanisms are required in QoT since all environment information is passed from the network protocols via Brand X to QoT. Any information calculated in QoT would be a subset of information gathered from the network protocol stack. QoT implements timer and event activation mechanisms as a part of QoT Core. The activation mechanisms initiate information updates with Brand X as information is required.

2.4.2.6 Communication Mechanism

The `Inter_Module Communication (IMC)` interface is utilized in the implementation of Brand X to facilitate communication between the various kernel modules and the protocol stack layers. The `IMC` interface allows modules to register functions or data that can be retrieved and used by other modules. Through this process, callback functions are registered between Brand X, QoT, and the other network protocol layers. By utilizing callback functions, we are able to eliminate the overhead and complexity that occurs with other process communication, such as, sockets and shared memory.

Interaction between user-space modules and kernel-space modules is achieved through the use of the `/proc` file system. The `/proc` file system allows modules running in user-space to interface with kernel-space modules through a file or stream read/write interface. This interface requires a user-space module to read or write to a file in the `/proc` directory. The file in the `/proc` directory is a stub interface provided by the Linux kernel that in turn calls a registered function in the kernel module. The one-sided initiation inherent with this mechanism makes it useful for configuration but limits kernel initiated data transfer. Other mechanisms should be employed for any time sensitive or two-way

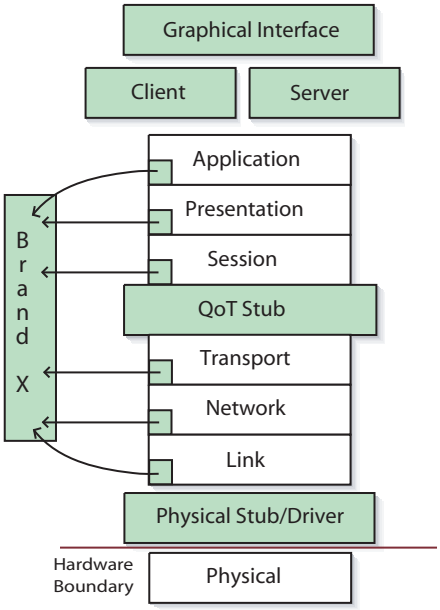

Figure 2.7: Cross-Layer Performance Analysis Test Harness.

communication between modules in kernel-space and user-space.

2.4.3 Test Harness Design

A test harness was constructed to control the QoT environment and measure system performance (see Figure 2.7). The test harness consists of a graphical user interface, an application driver, a QoT stub, a physical layer stub, protocol layer extensions, and a Brand X module.

The graphical user interface (GUI) is provided for test setup and debugging. The GUI allows direct access to protocol level configuration (see Figure 2.8). The GUI also provides a scripted interface which can accept and run test environment configurations (see Figure 2.9). In addition to the GUI, a programmatic interface is provided to facilitate more elaborate test harness interaction and to allow series of test to be run programmatically. The programmatic interface offers all of the functionality present in the GUI as well as additional debugging and scripting mechanisms.

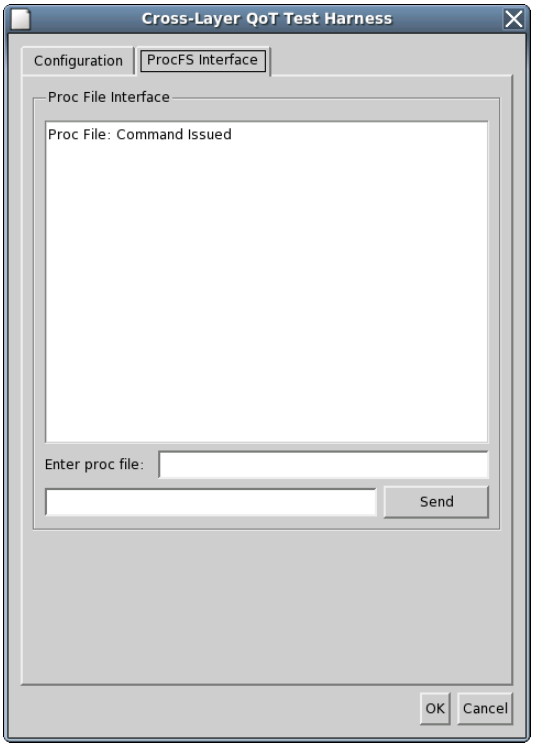

Figure 2.8: Test Harness Graphical User Interface.

An application driver sits on top of the protocol stack and is used to generate workload traffic and to provide protocol configuration. The application driver consists of two separate client and server modules.

- The client module generates network traffic according to the specifications of the experiment. As each packet is constructed the current system time is included in the packet payload. The traffic is then sent via the BSD socket interface into the network protocol stack.
- The server module listens on a specified port for configuration and data packets. Once a packet is received, a time stamp of the current system time is taken and stored along with the time stamp and packet information included in the payload. The test

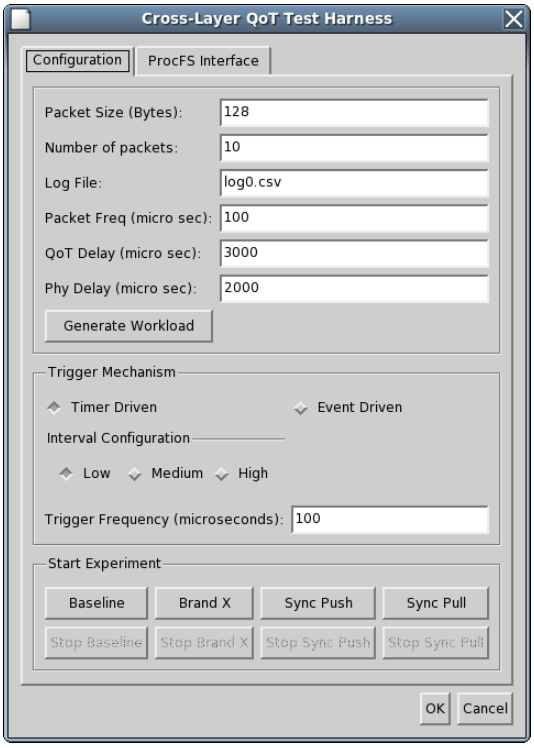

Figure 2.9: Test Harness Graphical User Interface.

packet protocol consists of one configuration packet, used to synchronize the settings between the client and the server modules, followed by test data packets and a final configuration packet marking the end of the test transfer. The number of workload packets sent can vary depending on the experiment. Once the final configuration packet is received the stored result-data is written to a log file.

A stubbed QoT protocol layer is inserted between the Session layer and the Transport layer of the protocol stack. The QoT stub receives all network traffic as it passes between the upper protocol layers and the transport protocol layer. The test harness QoT layer implements the limited functionality necessary to interact with a cross-layer architecture but does not include any transport switching or decision making functionality. Interfaces

to perform Synchronous Push, Synchronous Pull, Billboard, and configuration communication were implemented along with a configurable delay that can be applied to network traffic passing through QoT in order to approximate execution time for a full QoT system.

A physical layer stub is inserted in the protocol stack between the hardware and software interfaces. This protocol layer stub provides a delay mechanism used to provide a consistent physical transmission time. The delay is configurable and occurs uniformly on all traffic passed through the physical stub. By applying a consistent time delay, any variation resulting from uncontrolled changes to the ambient wireless environment can be eliminated.

During the performance analysis, the loopback interface is utilized to eliminate variability due to fluctuations in the ambient wireless environment. By controlling traffic delay at the physical stub and the properties of the loopback interface, the test harness can simulate various wireless transports.

Physical transports between devices can be tested using a remote Application Driver module. The Application Driver module is executed on the remote machine and intercepts traffic from the source client. By running across a live network environment it is possible to study protocol stack latency and physical transmission latency over various transmission mediums.

2.4.3.1 Brand X – Asynchronous Billboard

The Brand X module provides storage for data and configuration during information transfer between the network protocol layers and QoT. Information is passed to Brand X from the protocol layers and cached until it is requested by QoT. The Brand X module is able to provide asynchronous communication since it resides in the Linux kernel space but is not included in the layered network protocol stack. Any computations performed by Brand X do not directly effect the flow of network traffic.

Protocol layers pass information to Brand X utilizing a push communication mechanism and either a timer or event activation mechanism. Information is stored by Brand X

until a request is made by QoT, at which time the latest environment information is returned to QoT. The advantage of implementing Brand X separate of the network protocol stack is the ability to service information update interrupts out of the network data control path.

2.4.3.2 Synchronous Push

In the Synchronous Push architecture source protocol layers publish information directly to QoT via an interrupt mechanism. QoT is passed an interrupt and must block to service the information update. As more protocol layers are included in the system, QoT is forced to handle an increasing number of update interrupts. The increased interrupts for QoT can consume resources necessary for timely network traffic handling.

The Synchronous Push test harness utilizes the Application Driver module, the QoT module, the Physical layer module and the protocol layer extensions. The Brand X module is not utilized during Synchronous Push experiments.

2.4.3.3 Synchronous Pull

In Synchronous Pull architectures, QoT interacts directly with network protocol layers during the environment information gathering process by relating requests for information via a registered callback system. When information is required by QoT, an information update is signalled and sent to the source protocol layer. When the source protocol receives an update request, the response must be calculated and returned to QoT. This mechanism blocks network protocols longer than Synchronous Push architectures but can scale to handle much larger sets of network protocols.

The Synchronous Pull test harness utilizes the Application Driver module, the QoT module, the Physical layer module and the protocol layer extensions. The Brand X module is not utilized during Synchronous Pull experiments.

2.5 Performance Analysis of Cross-Layer Architectures in a QoT Environment

We conducted a performance analysis of network protocol stack latency on various cross-layer architectures. The following sections describe the performance analysis in greater detail. In the following sections, we present cross-layer performance analysis goals, performance analysis factors, evaluation techniques, evaluation metrics, test harness, results and analysis.

2.5.1 Cross-Layer Performance Analysis Goals

The goal of this performance analysis is to quantify the effect on network protocol stack latency of four cross-layer factors across multiple factor levels, focusing on increased latency in the network protocol stack due to the cross-layer interference. While prior research has focused on the effects of cross-layer architectures on end-to-end network performance, the area of network stack performance has received significantly less attention.

2.5.2 Performance Analysis Factors

The four factors included in the experimental performance analysis are: Architecture, Activation Mechanism, Workload, and Volatility. These factors represent four performance intensive areas that are common among cross-layer architectures. They are described in greater detail in the following sections.

2.5.2.1 Architecture

Three cross-layer architectures are compared with a control environment to isolate the effect of cross-layer mechanisms. The Control architecture is an unmodified Linux network protocol stack in which there are no cross-layer mechanisms. The three cross-layer architectures represent three fundamental data elicitation mechanisms, Brand X, Synchronous Push, and Synchronous Pull, described in Section 2.3.1.2. The three cross-layer

architectures were constructed from the same code base as the control architecture and include modifications for cross-layer functionality. We briefly discuss each of these three cross-layer architectures in turn.

- Analysis of the Billboard architecture facilitates an improved understanding of the effects of a loosely coupled asynchronous cross-layer architecture on protocol stack latency.
- Analysis of the Synchronous Pull and Synchronous Push architectures facilitates an improved understanding of the effects of traffic blocking on protocol stack latency.

2.5.2.2 Activation Mechanism

The two activation mechanisms compared in our performance analysis are Timer and Event. The activation mechanism determines the method by which information updates are initiated. Timer activation mechanisms trigger information updates according to a predefined time schedule. Event activation mechanisms monitor network traffic for changes in the environment; once changes have reached a predefined threshold an information update is signalled. By analyzing Timer and Event activation mechanisms we are able to determine the effect of update interruptions and traffic monitoring on network stack latency.

2.5.2.3 Workload

In order to accurately measure network stack latency, the system must be tested while processing appropriate workloads. The workload generated by the test harness for the experiments is comprised of UDP data packets. The packet size and packet frequency parameters are varied along with the system workload.

Uncontrolled network traffic is eliminated by closing all network interfaces except the loopback interface. Additionally network traffic on the loopback interface is monitored and any active network processes are closed.

Early results indicated that varying workload packet size caused unpredictable changes in the measured network protocol stack latency. The changes are a result of packet fragmentation and assembly in the network protocol stack. As a result, workload packet size is limited to 500 bytes, a level that causes no packet fragmentation and resulted in stable protocol stack latency in the initial performance measurements.

The following four traffic workloads represent real system usage scenarios:

- Heavy Traffic – Packets are sent at 1 microsecond intervals. This load represents a steady stream of data, pushing the boundaries of the capacity of the system, characterized by 90% utilization of the system bandwidth.
- High Medium Traffic – Packets are sent at 10 microsecond intervals. This load represents high network usage, such as real-time streaming media and is characterized by 70% utilization of the system.
- Low Medium Traffic – Packets are sent at 100 microsecond intervals. This load represents normal active usage of the system without stressing the system capacity, characterized by 30% utilization of the system.
- Light Traffic – Packets are sent at 1000 microsecond intervals. This load represents low utilization of the system, characterized by less than 10% system utilization.

2.5.2.4 Volatility

Volatility represents the condition and stability of the wireless link. Volatility is used to represent various link stabilities in the test system. Volatility levels produce workload on the system that can affect the performance of the cross-layer architecture.

The test harness implementation controls volatility levels through the information query frequency parameter. By varying the query frequency we can simulate various ambient wireless environments in a consistent and repeatable way.

The Timer activation mechanism has four query frequency levels: 1 millisecond, 10 milliseconds, 100 milliseconds and 1000 milliseconds. Query frequency levels represent environment conditions in which protocol layer status changes sufficiently within the respective time interval to require an information update.

The Event activation mechanism has query frequency thresholds: 1 unit, 10 units, 100 units and 1000 units levels to determine if an information update is required. Thresholds are ranges in which environment information can fluctuate before an information update is triggered.

- A highly volatile environment consists of perpetual changes in the ambient wireless environment and link conditions, such as moving between a high quality state with few bit errors and a low quality state with a high level of packet retransmission and bit errors.
  - Timer – 1 millisecond Query Interval
  - Event – 1 unit Information Threshold
- A heightened environment consists of frequent changes in the wireless environment causing packet corruption and errors, such as traveling between wireless access points.
  - Timer – 10 millisecond Query Interval
  - Event – 10 unit Information Threshold
- A moderately volatile environment consists of levels of packet corruption and errors that are typical of an average use case with minimal interference.
  - Timer – 100 millisecond Query Interval
  - Event – 100 unit Information Threshold
- A stable environment has consists of minimal changes in the ambient wireless environment and link conditions. Stability may suggest consistently good or consistently

bad conditions, but in either case minimal cross-layer updating is necessary to keep environmental information up to date.

- Timer – 1000 millisecond Query Interval
- Event – 1000 unit Information Threshold

2.5.3 Evaluation Techniques

To quantify the effect of cross-layer interference on network protocol stack latency each architecture is evaluated in context of the various network volatilities, workloads, and activation mechanisms. The results from the experiments are input into an experimental model to calculate the component effects. Once the component effects are known, the experimental model is refined by removing components that are not statistically significant.

The following sections describe the experimental design, the experimental model and the ANOVA terminology used in this paper.

2.5.3.1 Experimental Design

The experimental design uses ANOVA calculations to estimate the contribution of factors, interactions, and measurement errors in the performance of the network protocol stack. In order to determine component variance we performed a full factorial design utilizing every possible combination of levels and factors. The number of experiments, *n*, in our cross-layer study is:

$$n = (4 \text{ Architectures}) \times (2 \text{ Activation Mechanisms}) \times (4 \text{ Volatility Levels}) \times (4 \text{ Workloads}) \\ = 128 \text{ experiments}$$

The full factorial design facilitates calculation of the effects of factors, interactions, and measurement error. The experiments are repeated three times to distinguish variance caused by factors and interactions from measurement error.

Measuring a Control architecture in a theoretically implausible but conceptually useful condition in which cross-layer mechanisms contribute no latency established a baseline by which cross-layer architectures can be compared.

2.5.3.2 Experimental Model

The model for the cross-layer performance analysis is:

$$y_{ijklm} = \mu + \alpha_i + \beta_j + \zeta_k + \delta_l + \gamma_{ABij} + \gamma_{ACik} + \gamma_{ADil} + \gamma_{BCjk} + \gamma_{BDjl} + \gamma_{CDkl} + \gamma_{ABCijk} + \gamma_{ABDijl} + \gamma_{ACDikl} + \gamma_{BCDjkl} + \gamma_{ABCDijkl} + \varepsilon_{ijklm}$$
$$i = 1, ..., a; \quad j = 1, ..., b; \quad k = 1, ..., c; \quad l = 1, ..., d; \quad m = 1, ..., r;$$

Where:

y_{ijklm} = response (observation) in the m^{th} replication of experiment with factors *A, B, C* and *D* at levels *i, j, k* and *l*, respectively.

μ = mean response.

α_i = effect of factor *A* at level *i*.

β_i = effect of factor *B* at level *j*.

ζ_i = effect of factor *C* at level *k*.

δ_i = effect of factor *D* at level *l*.

γ_{XYxy} = interaction between two factors *X* and *Y* at levels *x* and *y*.

γ_{XYZxyz} = interaction between three factors *X*, *Y* and *Z* at levels *x, y* and *z*, respectively.

$\gamma_{ABCDijkl}$ = interaction between *A, B, C* and *D* at levels *i, j, k* and *l*, respectively.

ε_{ijklm} = errors at levels *i, j, k, l* over repetitions *m*.

2.5.3.3 ANOVA Terms

An overview of the ANOVA equations is provided in Table 2.1. The following are terms used in an ANOVA performance analysis:

Table 2.1: ANOVA Table for Cross-Layer Experiment Analysis

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F-value	F-table
y	$SSY = \sum(y_{ijklr}^2)$		$abcdr$			
$\bar{y}..$	$SS0 = abcdr\mu^2$		1			
$y - \bar{y}..$	$SST = SSY - SS0$	100	$abcdr - 1$			
A	$SSA = bcdr\sum\alpha_i^2$	$100(\frac{SSA}{SST})$	$ar - 1$	$MSA = \frac{SSA}{a-1}$	$\frac{MSA}{MSE}$	$F_{[1-\alpha;a-1,a(r-1)]}$
B	$SSB = acdr\sum\beta_j^2$	$100(\frac{SSB}{SST})$	$br - 1$	$MSB = \frac{SSB}{b-1}$	$\frac{MSB}{MSE}$	$F_{[1-\alpha;b-1,b(r-1)]}$
C	$SSC = abdr\sum\zeta_k^2$	$100(\frac{SSC}{SST})$	$cr - 1$	$MSC = \frac{SSC}{c-1}$	$\frac{MSC}{MSE}$	$F_{[1-\alpha;c-1,c(r-1)]}$
D	$SSD = abcr\sum\delta_l^2$	$100(\frac{SSD}{SST})$	$dr - 1$	$MSD = \frac{SSD}{d-1}$	$\frac{MSD}{MSE}$	$F_{[1-\alpha;d-1,d(r-1)]}$
e	$SSE = \sum e_{ijklm}^2$	$100(\frac{SSE}{SST})$	$abcd(r-1)$	$MSE = \frac{SSE}{abcd(r-1)}$		

- *Sum of Squares* is a quantification of the results associated with the given component. A component consists of a factor or the interaction between multiple factors.

- *Percentage of Variation* is a comparison of the individual component value and the summation of the total variation that provides the percentage of over all variation that is due to the effects of the stated component.

- *Degrees of Freedom* are the number of independent terms in the sum of squares calculation.

- *Mean Square* is calculated by taking the sum of squares over the degrees of freedom and is used in calculating an F-value for a component.

- The *F-value* is a ratio of the variance of a component and the variance of error. F-value is used in determining the statistical significance of a component.

- *F-table* is a computed ratio of the F distribution using the ratio of component degrees of freedom over error degrees of freedom. The computed ratio, F-value, is compared with F-table obtained from the table of F quantiles and the sums of squares are considered significantly different if the computed F-value is more than F-table. The comparison between calculated F-value and expected F-table is called the F-test.

Table 2.2: Factors and Levels for Cross-Layer Performance Analysis

Symbol	Factor	Level 1	Level 2	Level 3	Level 4
A	Architecture	Control	Brand X	Synchronous Push	Synchronous Pull
B	Activation Mechanism	Timer	Event		
C	Workload	1 μs	10 μs	100 μs	1000 μs
D	Volatility	1 μs/1 unit	10 μs/10 unit	100 μs/1000 unit	1000 μs/1000 unit

2.5.4 Evaluation Metrics

Latency is defined as the time it takes a packet to travel from an application, down the protocol stack, across a wireless link, and back up the protocol stack of the receiver. The hypothesis is that cross-layer architectures increase overall network performance at the expense of increased latency during packet processing within the protocol stack. Based on measured latency and experiment traffic patterns the theoretical throughput of the system can be calculated.

2.5.5 Experiment Setup and Configuration

The performance analysis consists of a matrix of experiments generated by the combination of all possible factors and levels, (see Table 2.2). The experiments in the matrix were repeated to ensure accuracy in the results and allow determination of statistical significance for the effects.

The matrix consists of:

- Four Architectures. Control, Brand X, Synchronous Push, and Synchronous Pull (see Section 2.5.2.1).
- Two Activation Mechanisms. Timer and Event (see Section 2.5.2.2).
- Four Workload Levels. The workload levels are varied through the network packet

frequency. The four packet intervals are 1 microsecond, 10 microseconds, 100 microseconds and 1000 microseconds (see Section 2.5.2.3).

- Four Volatility Levels. The level of volatility is dependant on the activation mechanism (see Section 2.5.2.4). Timer mechanism volatility levels are 1 microsecond, 10 microseconds, 100 microseconds and 1000 microseconds. The event mechanism volatility levels are referenced in terms of changes of 1 unit, 10 units, 100 units and 1000 units.

There are 128 experiments contained in the matrix of factors and factor levels. Each experiment represents a unique cross-layer setup and the matrix of experiments covers all possible combinations of values. In order to differentiate measurement errors from variation due to statistically significant factors, the experiments were repeated three times. Repetition of the experiment matrix provides us with the necessary data to calculate main effects, interaction effects and effects due to experimental error. Repetition of the experiment matrix provides us with the summary of 768,000 independent measurements.

2.5.5.1 Steady State Calculation

Each experiment has an initial startup cycle in which system latency fluctuates before a steady packet transmission time is achieved (see Figure 2.10). We are interested in measuring system performance during stable workload conditions due to the fact that the QoT environment remains active while the network protocol stack is alive. Initial startup occurs during the system startup cycle and does not impact the user experience.

In order to accurately measure the system each experiment is repeated 2,100 times. The initial 100 measurements represent the startup cycle and the results are discarded. The remaining 2,000 measurements are recorded and averaged in order to return system latency at steady state.

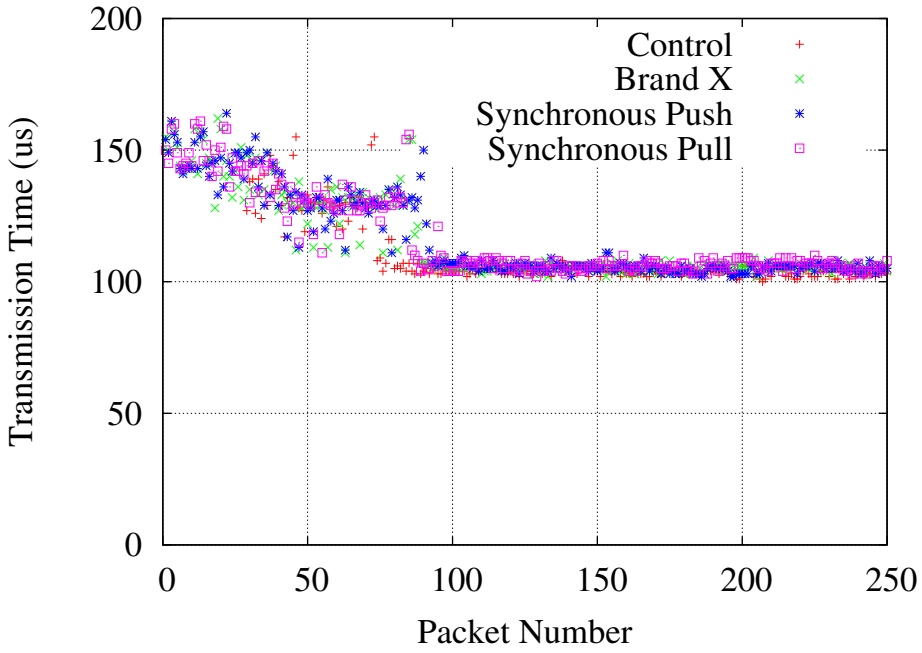

Figure 2.10: Packet Transmission Startup Cycle

2.5.6 Results and Analysis

The following sections provide an overview of results of our experiments and detail the statistical techniques used in the analysis.

2.5.6.1 Computation of Effects

The table mean (μ), must first be computed in order to calculate the main effects of the factors for all experiments. The table mean is used throughout the ANOVA calculations when comparing a set of the experimental results to the predicted results. The equation for calculating the table mean (μ) is:

$$\mu = \bar{y}.... = \frac{1}{abcdr} \sum_{i=1}^{r} \sum_{j=1}^{a} \sum_{k=1}^{b} \sum_{l=1}^{c} \sum_{m=1}^{d} y_{ijklm} = 92.04$$

Once the table mean is calculated, the parameters for the factors at each level can be calculated. Parameters are calculated for each factor at each respective value by averaging

Table 2.3: Main Effects in the Cross-Layer Performance Analysis

Factor	Level 1	Level 2	Level 3	Level 4
Architecture	-16.15	-.045	3.71	12.89
Activation Mechanism	-11.17	11.17		
Workload	-0.17	0.26	-0.06	-0.03
Volatility	5.82	-2.05	-2.12	-1.65

along various axes of the experiment matrix and subtracting the table mean. The following are the parameter values of Factor A, Architecture:

$$\text{Control} = \alpha_1 = \bar{y}_{1...} - \bar{y}_{....} = 75.89 - 92.04 = -16.15$$

$$\text{Brand X} = \alpha_2 = \bar{y}_{2...} - \bar{y}_{....} = 91.59 - 92.04 = -0.45$$

$$\text{Synchronous Push} = \alpha_3 = \bar{y}_{3...} - \bar{y}_{....} = 95.75 - 92.04 = 3.71$$

$$\text{Synchronous Pull} = \alpha_4 = \bar{y}_{4...} - \bar{y}_{....} = 104.93 - 92.04 = 12.89$$

Parameters are valuable for identifying trends in the experimental results but they are not a direct indication of relative performance between trend levels. The parameters for the factors at each respective level are listed in Table 2.3. There are several trends in the parameter results that are of interest.

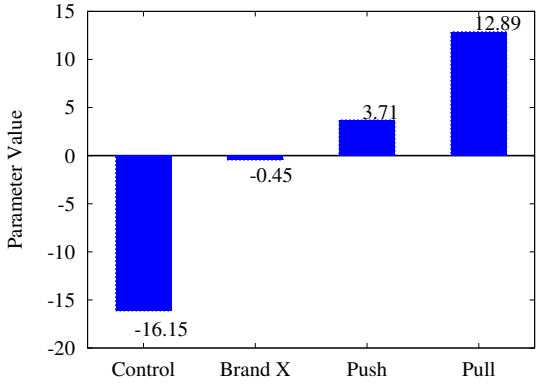

Figure 2.11: Architecture Parameters

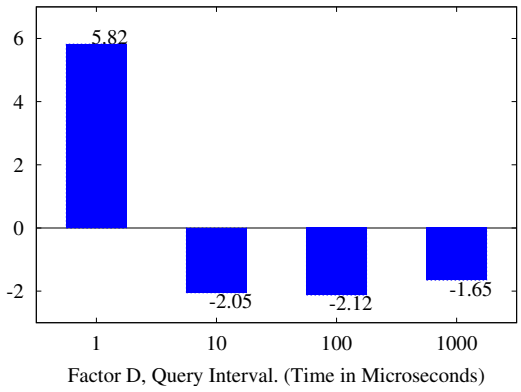

Figure 2.12: Volatility Parameter Levels

- Factor *A*, Architecture, shows a significant variance between the four factor levels: parameter values range between -16.15 and 12.89 (see Figure 2.11). Level 1, the Control architecture, has the lowest parameter value, signifying that the Control architecture has the least latency of the four architectures. Level 2, Brand X, has the second lowest parameter value at -0.45, signifying Brand X incurs less latency than the synchronous cross-layer architectures. Level 3, Synchronous Push, has the next lowest value, 3.71, which indicates that the shorter blocking times of Synchronous Push architectures incur less latency than the longer blocking times of Synchronous Pull architectures.

- The parameters of factor *D*, Query Interval, show an interesting trend (see Figure 2.12). The factor levels of *D* represent query interval frequency and as the factor levels decrease query frequency decreases. The parameter values at levels two, three and four vary only slightly from one another suggesting that at the lower query frequencies, network latency varies only slightly. From factor level two to factor level one there is a significant increase. The high parameter value of factor *D* level one suggests a large increase in latency between level two and level one. This indicates that there is a threshold for query intervals between level one and level two.

Parameters are used to calculate the main effect of each factor by the respective equation in Table 2.1. For instance, the effect of factor *A* is:

$$SSA = bcdr \sum \alpha_i^2 = 4*2*4*3*((-16.15)^2*(-0.45)^2*(3.71)^2*(12.89)^2) = 42337.52$$

By calculating the main effects we can determine what percentage of the variation results from each factors.

2.5.6.2 Computation of Interactions

In order to calculate interactions between factors we must average across multiple axes of the experimental matrix and account for individual factor effects as well as the table mean. Parameter values are calculated for each axis of the experiment matrix. The equation for calculating the parameters of the first order interaction between architecture (*A*) and activation mechanism (*B*) at level one is:

$$\gamma_{11...} = \bar{y}_{11...} - \bar{y}_{1....} - \bar{y}_{.1...} + \bar{y}_{.....} = 75.7956 - 91.8719 - 75.8900 + 92.0419 = 0.0756$$

The resulting matrix of values is used to calculate the interactions with the following equation:

$$SSAB = cdr \sum_{ij} \gamma_{ij}^2 = 25058.19$$

Parameter calculation becomes increasingly complex in second-order and third-order interactions as the number of individual effects and sub-interactions increases.

2.5.6.3 Computation of Errors

The test harness runs in a Linux operating system environment in which all possible factors affecting the experiment results cannot be controlled. In order to limit uncontrolled factors, all nonessential programs and processes are shutdown when experiments are conducted. Even with these precautions, the operating system is very complex and not all

factors can be controlled. Therefor, factors outside the defined test system can still contribute to the experiment results, causing artificially high or low measurements. In order to ascertain the influence of outside factors, experiments are run in random order and repeated multiple times.

The estimated response, $\hat{y}_{ijkl}$, in the (i, j, k, l) experiment is given by a summation of the table mean and all of the relative effects:

$$\hat{y}_{ijkl} = \mu + \alpha_i + \beta_j + \zeta_k + \delta_l + \sum interactions$$
$$\hat{y}_{1111} = 92.04 + -16.15 + -0.17 + -11.17 + 5.82 + 5.13 = 75.50$$

The difference between the observed response, y_{ijklm}, and the predicted response, $\hat{y}_{ijkl}$, is residual or error:

$$e_{ijklm} = y_{ijklm} - \hat{y}_{ijkl.}$$
$$e_{11111} = 75.50 - 74.72 = 0.78$$

The sum of the squared errors (SSE) is used to calculate the total variance due to error and used to compute statistical significance for the effects. SSE is given by:

$$SSE = \sum_{i=1}^{abcd} \sum_{j=1}^{r} e_{ij}^2 = 1257.73$$
$$SSE = (-0.7838)^2 + (1.6823)^2 + \cdots + (0.4736)^2 = 1257.73$$

The sum of the squared errors is used to calculate total variance because the sum of the errors for the entire experiment always equals zero.

2.5.6.4 Allocation of Variation

Once the effects of the factors, interactions and errors have been calculated, the total variation (SST) of the experiment can be calculated. SST is used in the allocation and analysis of variance among the factors and interactions in the model. Using SST we can calculate the percentage of variation due to an individual factor and if the effect is statistically significant.

In our model, (see Section 2.5.3.1), the total variation of *y* can be allocated to four factors, eleven interactions and experiment errors. In order to do so, we square both sides of the model and add across all observations:

$$\sum_{ijkl} y_{ijkl}^2 = abcd\mu^2 + bcd\sum_{jkl}\alpha_{jkl}^2 + acdr\sum\beta_j^2 + abdr\sum\zeta_k^2 + abcr\sum\delta_l^2 + cd\sum_{kl}AB^2 + bd\sum_{jl}AC^2 + bc\sum_{jk}AD^2 + ad\sum_{il}BC^2 + ac\sum_{ik}BD^2 + ab\sum_{ij}CD^2 + d\sum_{l}ABC^2 + c\sum_{k}ABD^2 + b\sum_{j}ACD^2 + a\sum_{i}BCD^2 + \sum ABCD^2 + \sum e_{ijklm}^2$$

The sum of squares of the terms can be substituted in to clarify the equation:

$$SSY = SS0 + SSA + SSB + SSC + SSD + SSAB + SSAC + SSAD + SSBC + SSBD + SSCD + SSABC + SSABD + SSACD + SSBCD + SSABCD + SSE$$

The total variation is calculated by the sum of squares of the factors and interactions in the equation:

$$SST = SSY - SS0 =$$
$$SSA + SSB + SSC + SSD + SSAB + SSAC + SSAD + SSBC + SSBD + SSCD + SSABC + SSABD + SSACD + SSBCD + SSABCD + SSE = 131662.84$$

$$SST = 3384798.12 - 3253135.29 =$$
$$42337.52 + 47871.78 + 9.60 + 4346.15 + 25058.19 + 33.94 + 4404.45 + 7.42 + 2401.24 + 40.92 + 11.35 + 3659.65 + 132.00 + 23.31 + 67.59 + 1257.73 = 131662.84$$

Variation associated with measurement error and uncontrolled factors is:

$$e = 100(\tfrac{SSE}{SST}) = 100(\tfrac{1257.73}{131662.84}) = .0096 = 0.96\%$$

A low error value is a sign of a well designed experiment in which all significant factors have been included. The commonly accepted level for unexplained variation (error) in an ANOVA evaluation is 0.05 or 5% [26]. This is a reassurance that the factors we selected for our performance evaluation are the primary factors that influence cross-layer protocol stack latency.

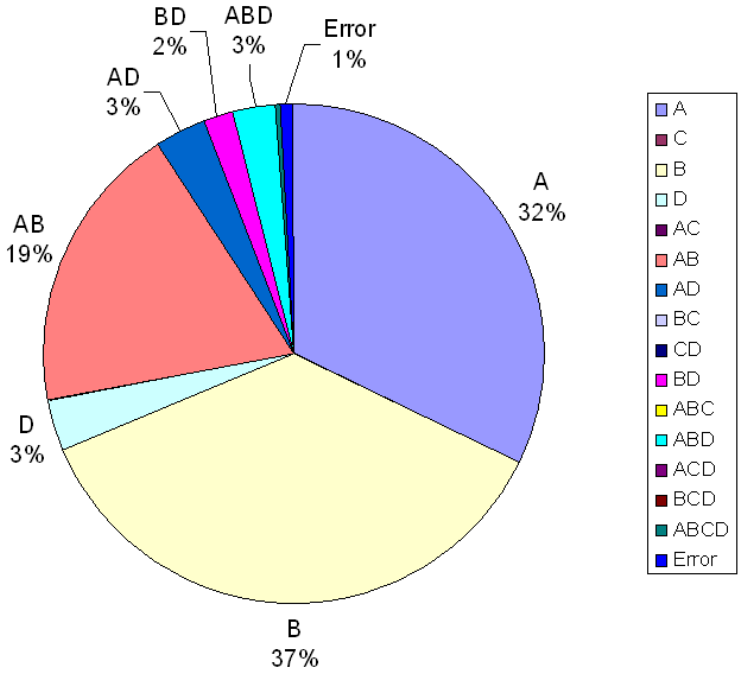

Figure 2.13: Component Percentages of Total Variation

In order to calculate the percentage of variation due to a specific factor the sum of the square of the factor is divided by the total sum of the squares and multiplied by 100. The three components that contribute a significant percentage of variation, 5% or greater, account for 87.55% of the total variation (see Figure 2.13). These components are:

$$B = 100(\frac{SSB}{SST}) = 100(\frac{47871.75}{131662.84}) = 36.36\%$$
$$A = 100(\frac{SSA}{SST}) = 100(\frac{42337.52}{131662.84}) = 32.16\%$$
$$AB = 100(\frac{SSAB}{SST}) = 100(\frac{25058.19}{131662.84}) = 19.03\%$$

Factor *B* represents the two activation mechanism implemented in the study: Timer and Event. This factor accounts for the largest percentage of variation in the experiment results: 36.36% (see Figure 2.13). The large percentage of variation suggests that the fundamental differences between the Timer and Event mechanisms are the leading cause for cross-layer latency in the network protocol stack. Event mechanisms had an average increase in latency of 19% in Brand X, 31% in Synchronous Push and 44% in Synchronous

Pull.

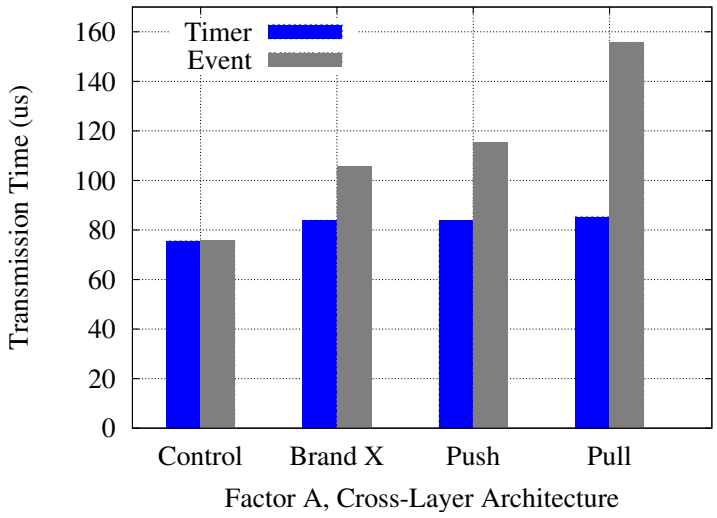

Figure 2.14: Factor *B*, a comparison of Activation Mechanism Transfer Times

The large increase in latency in Event activation mechanisms is a result of additional network traffic processing that occurs with each network packet processed in the protocol stack (see Figure 2.14). Event activation mechanisms monitor each network traffic packet as it passes through the protocol stack. In contrast, the Timer activation mechanism only monitors network traffic when an information update is requested. If network performance is the highest priority then a Timer activation model should be utilized. If timely system response in a highly volatile environment is a high priority and an increase in protocol stack latency is acceptable then the event activation mechanism should be utilized.

Factor *A* represents the four architectures involved in the performance analysis, Control, Brand X, Synchronous Push, Synchronous Pull. This factor has the second highest percentage of variation, 32.16% (see Table 2.4). In Section 2.5.6.1, we determined the relative order of factor *A*'s levels in terms of performance. The experimental results show that the Control architecture has the lowest associated latency at 75.50 μs. Brand X has an 8.3% increase in latency over the Control architecture with a Timer activation mechanism

Table 2.4: ANOVA Table for Cross-Layer Performance Analysis

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F-Computed	F-Table
y	3384798.12		384			
$\bar{y}$	3253135.29	1				
$y - \bar{y}....$	131662.84	100	383			
Main Effects	94565.05	71.82%	10	9456.50	1924.78	1.8678
A	42337.52	32.16%	3	14112.51	2872.47	2.6399
B	47871.78	36.36%	1	47871.78	9743.85	3.8780
C	9.60	0.01%	3	3.20	0.65	2.6399
D	4346.15	3.30%	3	887.39	180.62	2.6399
First-Order Interactions	31946.15	24.26%	36	887.39	180.62	1.4633
AB	25058.19	19.03%	3	8352.83	1700.14	2.6399
AC	33.94	0.03%	9	3.77	0.77	1.9166
AD	4404.45	3.35%	9	489.38	99.61	1.9166
BC	7.42	0.01%	3	2.47	0.50	2.6399
BD	2401.24	1.82%	3	800.41	162.92	2.6399
CD	40.92	0.03%	9	4.55	0.93	1.9166
Second-Order Interactions	3826.32	2.91%	54	70.86	14.42	1.3878
ABC	11.35	0.01%	9	1.26	0.26	1.9166
ABD	3659.65	2.78%	9	406.63	82.77	1.9166
ACD	132.00	0.10%	27	4.89	0.99	1.5294
BCD	23.31	0.02%	9	2.59	0.53	1.9166
Third-Order Interaction						
$ABCD$	67.59	0.05%	27	2.50	0.51	1.5294
Error						
e	1257.73	0.96%	256	4.91		

and a 28.0% increase with an Event activation mechanism. Synchronous Push has a 10.8% increase in latency over the Control architecture with a Timer mechanism and a 42.4% increase with an Event mechanism. Synchronous Pull has a 12.9% increase in latency over the Control architecture utilizing a Timer mechanism and a 53.5% increase utilizing an Event mechanism. It is clear from the experimental results that Brand X has the lowest network protocol stack latency of the cross-layer architectures.

The third major component is the interaction between factor *A* and factor *B*, which causes 19.03% of the total experiment variation. The *AB* interaction can be seen in the results table by comparing the variance in packet transmission times between the Timer and Event mechanism in the various architectures (see Figure 2.14). The rate of change in the packet transmission times is not held constant across the various architectures. The large percentage of variation due to this interaction indicates there is a correlation between the architecture and the activation mechanism utilized. Brand X architecture utilizing a Timer activation mechanism has the highest cross-layer performance of the tested. The experimental results show that if an event system is required that the Brand X Event activation system has the least incurred protocol latency due to cross-layer interference.

2.5.6.5 Analysis of Variance

In order to determine if a factor has a significant impact on the response the F-test is used to compare the variance resulting from a factor with the variance caused by errors. The F-test determines if variance due to a factor can be considered statistically significant. The F-test is an appropriate determination of statistical significance in our model because SSE and the sum of squares of the factors are assumed to have a chi-square distribution with the errors that are normally distributed.

Sum of squares having a chi-square distribution of the ratio $\frac{SSA/DFF}{SSE/DFE}$, where DFF and DFE are the degrees of freedom for SSA and SSE respectively, have an F-distribution with DFF numerator and DFE denominator degrees of freedom. The quantity $\frac{SSA}{DFA}$ is called the Mean Square of *A* (MSA). Similarly, $\frac{SSE}{DFE}$ is called the Means Square of Errors (MSE).

If the computed ratio $\frac{MSA}{MSE}$ is greater than the quantile, $F_{[1-\alpha;DFF,DFE]}$, which can be obtained from a table of quantiles of F-variates, then the variance based on the factor *A* is considered statistically significant. For instance, the factor *A* has a calculated F-ratio of 2872.47, which is larger than the F-table value, 2.6399.

Factor A, F-ratio		Table F-value
$\frac{SSA/DFF}{SSE/DFE}$		$F_{[1-\alpha;DFF,DFE]}$
$\frac{42337.52/3}{1257.73/256}$		$F_{[0.95;3,256]}$
2872.47	$>$	2.6399

Table 2.5: Significant Factors in Cross-Layer Network Latency

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F-Computed	F-Table
A	42337.52	32.16%	3	14112.51	2872.47	2.6399
B	47871.78	36.36%	1	47871.78	9743.85	3.8780
D	4346.15	3.30%	3	887.39	180.62	2.6399
AB	25058.19	19.03%	3	8352.83	1700.14	2.6399
AD	4404.45	3.35%	9	489.38	99.61	1.9166
BD	2401.24	1.82%	3	800.41	162.92	2.6399
ABD	3659.65	2.78%	9	406.63	82.77	1.9166

Calculating the F-ratio for the factors of the performance analysis we found that there are seven factors and interactions that are statistically significantly (see Table 2.5). We have already discussed the three components that contribute the largest percentage of variation (see Section 2.5.6.4). The four statistically significant components that contribute less than 5% of the total variation are:

- Factor *D*, information query interval, contributes 3.30% of the total variation. Our hypothesis was that the network stack latency would increase as the query level increased. As the query level approaches the limit of the function (constant querying)

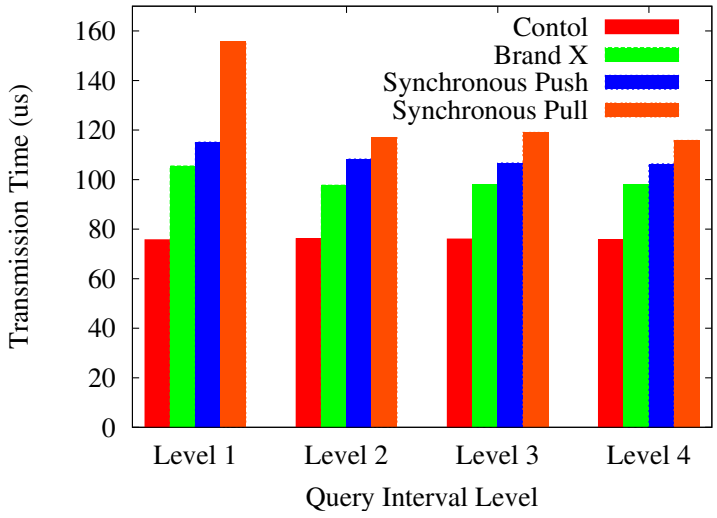

Figure 2.15: Event Transmission times with Workload at 1 μs

the entire bandwidth of the system is utilized by information updates and there is an infinite delay for network traffic (see Figure 2.15). We were not able to fully test this hypothesis due to constraints in the Linux kernel timer mechanism.

The activation mechanisms are controlled by the kernel system timer. The kernel system timer is the fastest timing mechanism available in the standard Linux 2.6.8-24 kernel. Function pointers are registered in a system timer queue and are executed after a set delay. The system timer is limited to the frequency of the system clock, which is specified in the kernel variable `HZ`. The `HZ` variable specifies the granularity of the system timer. In the Linux 2.6.8-24 kernel the `HZ` value is 1,000, meaning the system timer launches at one millisecond intervals. The fastest information query frequency available using the system timer is once per millisecond. Network traffic is processed on the order of 10-100 microseconds, therefor a millisecond delay on information updates is insufficient to fully test this factor.

Alternatives to the system timer have been considered including modified high-frequency system timers, high-speed internal clocks, and multi-threaded query mechanisms that don't rely on the system timer. These mechanism are not natively supported in the Linux 2.6.8-24 kernel and require substantial modification to the underlying kernel and network protocol stack. Such modifications have been implemented but they are still in the experimental phase and are only supported on custom hardware platforms.

- Interaction *AD* contributes 3.35% of the total variation. The interaction between factor *A*, architecture, and factor *D* (query interval) results from a variation in the transmission time as the query frequency increases over the various architectures. The largest variation occurs in Synchronous Pull architectures utilizing an Event activation mechanism. The least variation occurs in the Control architecture.

- Interaction *BD* contributes 1.82% of the total variation. Although small, this interaction is still considered statistically significant. Interaction *BD* results from different rates of increase in transmission times between Timer and Event activation mechanisms as the query frequency is increased.

- Interaction *ABD* contributes 2.78% of the total variation. This interaction is the result of different rates of change in the *BD* interaction among the various architectures.

The remaining interactions are not statistically significant and are considered negligible.

Removing the non-significant components from our original experiment model (see Section 2.5.3.2) yields a simplified model that represents cross-layer network performance.

$$y_{ijklm} = \mu + \alpha_i + \beta_j + \delta_l + \gamma_{ABij} + \gamma_{ADil} + \gamma_{BDjl} + \gamma_{ABDijl} + \varepsilon_{ijklm}$$
$$i = 1, ..., a; \quad j = 1, ..., b; \quad k = 1, ..., c; \quad l = 1, ..., d; \quad m = 1, ..., r;$$

2.6 Conclusions and Future Work

It was our hypothesis that an asynchronous cross-layer architecture could be utilized in an efficient manor to provide accurate and timely environment information in a heterogeneous wireless environment. Brand X is the resulting cross-layer architecture that fulfils the information gathering requirements of QoT and provides an extensible and configurable autonomous data gathering mechanism. Implementation of the Brand X architecture provides a functional proof that the requirement for cross-layer information gathering in a QoT context can be achieved in a heterogeneous wireless environment. In order to further validate our hypothesis that an asynchronous cross-layer architecture can provide information in a timely and efficient manner we examined the performance of Brand X versus a control architecture and synchronous cross-layer architectures.

2.6.1 Performance Analysis

Our findings indicate that Brand X incurs between 7.96% and 47.6% less network stack latency than the other cross-layer architectures measured. These findings confirm the fact that an asynchronous cross-layer architecture can achieve less latency and higher performance than similar synchronous architectures.

Brand X incurs as little as 8.13% and at most 39.38% more network stack latency than the control environment. While Brand X out performs the other cross-layer architectures in the study there remains room for improvement. Further optimization of the Brand X communication mechanisms and internal code could lead to addition improvements in network performance.

This research quantifies the effects of cross-layer architecture, activation mechanism, workload, and volatility on network protocol stack latency. This study has shown that the type of cross-layer architecture selected can have

2.6.2 Timer versus Event Activation Mechanisms

Examining the results from the Event activation mechanism experiments can be misleading because of the large increase in network stack latency. The increase in network protocol stack latency should be compared to the latency of the complete data transmission to determine the effect of increased protocol stack latency on theoretical system throughput. For instance, in a Bluetooth data transfer each transmission slot is 625μs long [27]. Utilizing Brand X the additional cross-layer latency for a Bluetooth environment is 30μs, or 4.8% of the slot window. The 4.8% of the slot window utilized by Brand X occurs during the protocol processing time frame and does not directly impact the data transmission time, therefor theoretical throughput is not decreased. In 5 slot Bluetooth data transmission, Brand X utilizes only 0.20% to 0.96% of the available slot.

In a worst case scenario, Synchronous Pull utilizes an Event mechanism and increases protocol stack latency by 75μs. This will increase the protocol processing time and reduce the data burst time by 55μs or 8.8%. The loss of 8.8% of the theoretical throughput of the Bluetooth connection is a significant decrease in performance, but the cross-layer functionality facilitates communication in an environment in which no stable connection could be maintained.

2.6.3 Future Work

The cross-layer test harness provides a valuable tool for implementing and comparing various cross-layer mechanisms in a Linux environment. The following are just a few of the many areas that could be further examined.

The cross-layer test harness utilizes the Linux kernel system timer functionality in order to schedule regular events, such as timer based information queries. The Linux kernel timer is limited in timing granularity; the Linux 2.6.8-24 kernel timer is limited to one millisecond intervals. The addition of a higher frequency timing mechanism would provide the ability to fully test cross-layer architectures.

The results from the performance analysis focus on network stack latency in a single

network environment. The test harness could be extended to include support for various wireless transports. This would allow examination of cross-layer architectures in their target environments.

The results presented in this thesis dealt only with constant network traffic at set data rates. It is well understood that traffic and usage patterns play an important role in evaluating network performance. Extension of the Application Driver to include dynamic and repeatable traffic generation would allow researches to study the effects of traffic patterns on cross-layer architectures in a heterogeneous wireless environment.

Chapter 3

ANOVA Tables

Table 3.1: ANOVA Table for Cross-Layer Performance Analysis

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F-Computed	F-Table
y	3384798.12		384			
$\bar{y}$	3253135.29	1				
$y - \bar{y}....$	131662.84	100	383			
Main Effects	94565.05	71.82%	10	9456.50	1924.78	1.8678
A	42337.52	32.16%	3	14112.51	2872.47	2.6399
B	47871.78	36.36%	1	47871.78	9743.85	3.8780
C	9.60	0.01%	3	3.20	0.65	2.6399
D	4346.15	3.30%	3	887.39	180.62	2.6399
First-Order Interactions	31946.15	24.26%	36	887.39	180.62	1.4633
AB	25058.19	19.03%	3	8352.83	1700.14	2.6399
AC	33.94	0.03%	9	3.77	0.77	1.9166
AD	4404.45	3.35%	9	489.38	99.61	1.9166
BC	7.42	0.01%	3	2.47	0.50	2.6399
BD	2401.24	1.82%	3	800.41	162.92	2.6399
CD	40.92	0.03%	9	4.55	0.93	1.9166
Second-Order Interactions	3826.32	2.91%	54	70.86	14.42	1.3878
ABC	11.35	0.01%	9	1.26	0.26	1.9166
ABD	3659.65	2.78%	9	406.63	82.77	1.9166
ACD	132.00	0.10%	27	4.89	0.99	1.5294
BCD	23.31	0.02%	9	2.59	0.53	1.9166
Third-Order Interaction						
$ABCD$	67.59	0.05%	27	2.50	0.51	1.5294
Error						
e	1257.73	0.96%	256	4.91		

Table 3.2: Cross-Layer Performance Analysis Compiled Data

Architecture		Control		Brand X		Push		Pull		Statistics		Column
Packet Freq.	Query Interval	Timer	Event	Timer	Event	Timer	Event	Timer	Event	Sum	Avg	Effect
1 μs	1 μs	75.5071	75.6795	83.8844	105.4885	83.6821	115.1627	85.2557	155.7365	780.40	97.55	5.51
	10 μs	75.5944	76.1597	81.9164	97.7554	82.4994	108.3091	80.9567	116.9414	720.13	90.02	-2.03
	100 μs	75.7461	76.0200	81.9044	97.9320	81.0966	106.7114	81.5025	119.0579	719.97	90.00	-2.05
	1000 μs	75.9358	75.7223	83.3514	97.9079	82.7985	106.3658	81.5082	115.8113	719.40	89.93	-2.12
10 μs	1 μs	75.6972	76.2092	84.8776	106.2620	83.9563	114.7521	84.2583	158.2979	784.31	98.04	6.00
	10 μs	75.5664	75.7124	83.0046	99.5832	80.6190	109.3316	82.1611	113.3032	719.28	89.91	-2.13
	100 μs	75.9819	75.4701	81.2353	97.8536	81.3514	106.4282	81.4564	118.0172	717.79	89.72	-2.32
	1000 μs	76.0808	75.8520	83.7521	98.1540	81.3121	106.5755	88.0114	122.4871	732.22	91.53	-0.51
100 μs	1 μs	76.0670	75.6028	85.1873	106.1215	84.5024	115.8399	83.6679	157.2325	784.22	98.03	5.99
	10 μs	75.5516	75.8259	81.9327	99.6002	81.9697	107.9593	80.8311	116.0317	719.70	89.96	-2.08
	100 μs	75.9837	76.2781	81.7282	97.5425	81.0884	106.9444	80.6425	117.7380	717.95	89.74	-2.30
	1000 μs	75.9199	76.0929	81.5885	100.1752	81.3081	106.4259	81.3289	118.7752	721.61	90.20	-1.84
10000 μs	1 μs	76.0594	75.4785	85.3365	107.6619	84.8681	114.4315	85.0877	153.6853	782.61	97.83	5.78
	10 μs	75.7486	76.0787	82.2388	97.5843	81.7803	108.7600	81.2474	117.3000	720.74	90.09	-1.95
	100 μs	76.1673	76.9355	82.1224	97.6051	81.7934	106.5655	81.1824	119.3041	721.68	90.21	-1.83
	1000 μs	75.6782	76.0773	81.8040	97.8699	81.9648	106.9332	81.2558	117.7578	719.34	89.92	-2.12
Statistics	Sum	1213.29	1215.19	1325.86	1605.10	1316.59	1747.50	1320.35	2037.48	11781.36		
	Avg	75.83	75.95	82.87	100.32	82.29	109.22	82.52	127.34		92.04	
	Row Effect	-16.21	-16.09	-9.18	8.28	-9.75	17.18	-9.52	35.30			

Table 3.3: Main Effects in the Cross-Layer Performance Analysis

Factor	Level 1	Level 2	Level 3	Level 4
A	-16.15	-.045	3.71	12.89
B	-11.17	11.17		
C	-0.17	0.26	-0.06	-0.03
D	5.82	-2.05	-2.12	-1.65

Table 3.4: Data Repetition Number 1

Architecture	Query	Control		Brand X		Push		Pull		Statistics		Column
Packet Freq.	Interval	Timer	Event	Timer	Event	Timer	Event	Timer	Event	Sum	Avg	Effect
1 μs	1 μs	74.7233	75.0099	81.6107	104.5334	80.7206	113.9468	85.8221	159.4413	775.8085	96.9760	51.1315
	10 μs	75.2046	75.3813	81.3277	97.4823	81.4013	107.7722	79.9658	117.1714	715.7069	89.4633	43.6188
	100 μs	75.2382	75.2461	81.3108	97.1972	79.7885	105.4334	80.8179	116.8606	711.8932	88.9866	43.1421
	1000 μs	75.0157	75.4103	85.8537	96.7995	84.7964	105.8642	81.5270	115.0420	720.3093	90.0386	44.1941
10 μs	1 μs	75.1141	75.4529	84.9547	105.6712	81.4539	113.7732	84.0331	155.8916	739.1483	92.3935	46.5489
	10 μs	74.9426	75.0589	82.9100	99.7543	80.0126	112.3019	80.9226	112.2146	712.0562	89.0070	43.1624
	100 μs	75.3556	75.1714	81.1467	97.1025	80.1678	105.7091	79.9805	117.4224	718.1178	89.7647	43.9201
	1000 μs	75.4460	75.2667	82.6975	97.8811	80.9673	105.0925	101.7980	119.9989	776.3450	97.0431	51.1985
100 μs	1 μs	75.6433	75.2293	84.7369	104.7701	83.4971	116.3619	81.2872	150.7033	723.0836	90.3854	44.5409
	10 μs	75.1451	74.8264	81.8563	99.2140	80.5018	107.0173	81.1130	116.6417	715.0983	89.3872	43.5427
	100 μs	74.9505	75.2167	82.4039	97.0794	81.3392	106.6596	79.3450	118.1036	716.3161	89.5395	43.6949
	1000 μs	74.8427	75.0226	82.3035	104.1167	80.9826	105.8711	81.1136	118.8306	772.2293	96.5286	50.6841
1000 μs	1 μs	75.1956	75.3172	85.0557	108.7569	84.5502	114.1688	85.4523	152.1983	716.6827	89.5853	43.7407
	10 μs	75.6017	75.4644	81.3945	97.4402	80.8900	108.2788	79.7364	120.4681	723.4392	90.4299	44.5853
	100 μs	75.0631	75.8853	82.5170	97.4034	81.9915	106.3982	80.7953	123.3850	719.2745	89.9093	44.0647
	1000 μs	75.1399	75.3671	82.0052	97.1499	82.2435	106.4318	80.9989	117.3461	780.6954	97.5869	51.7423
Statistics	Sum	1202.62	1053.93	1324.08	1497.81	817.13	1081.98	1324.70	2031.72	11736.20		
	Avg	75.1639	75.2811	82.7553	99.8546	81.5815	108.8175	82.7943	126.98253		45.8445	
	Row Effect	29.3193	29.4365	36.9108	54.0100	35.7370	62.9730	36.94980	81.1379			

Table 3.5: Data Repetition Number 2

Architecture		Control		Brand X		Push		Pull		Statistics		Column
Packet Freq.	Query Interval	Timer	Event	Timer	Event	Timer	Event	Timer	Event	Sum	Avg	Effect
1 μs	1 μs	77.1894	77.2525	86.5655	106.4445	86.9521	116.7543	86.4350	154.3167	726.66	90.83	44.99
	10 μs	76.7580	77.2756	83.4445	99.1236	84.2635	108.0615	81.7870	117.8180	736.81	92.10	46.26
	100 μs	77.3630	76.9853	83.5066	100.5481	83.5860	109.0710	83.6223	122.1252	728.53	91.07	45.22
	1000 μs	76.6602	76.6844	83.0878	98.8022	83.0710	107.8716	83.2209	117.2583	791.91	98.99	53.14
10 μs	1 μs	77.6397	77.3419	86.2678	108.5066	85.7754	115.9742	85.8143	161.5676	739.57	92.45	46.60
	10 μs	77.1594	77.6512	84.2409	101.5902	82.0289	109.5129	85.1873	115.8432	733.48	91.68	45.84
	100 μs	77.5176	76.6802	82.3740	99.2977	83.6938	108.3730	83.2362	122.3062	733.21	91.65	45.81
	1000 μs	77.1257	77.2909	87.9143	99.8327	82.1973	109.8885	82.4182	122.8990	798.89	99.86	54.02
100 μs	1 μs	76.9221	76.6828	87.2199	108.6118	86.5413	116.9553	85.8890	164.8154	736.98	92.12	46.28
	10 μs	76.5934	77.0368	83.5060	101.0884	84.5266	110.8774	81.8038	115.2572	726.43	90.80	44.96
	100 μs	76.7438	78.3151	82.7838	99.4913	81.7849	108.4492	82.4871	116.3782	730.69	91.34	45.49
	1000 μs	77.4398	78.0805	82.6433	100.0026	82.2199	108.3172	82.7470	125.5302	803.64	100.45	54.61
1000 μs	1 μs	77.4261	76.8906	86.5255	108.5855	86.4613	115.5466	85.6376	157.4114	728.66	91.08	45.24
	10 μs	77.0668	77.2593	84.5597	98.8648	82.4855	110.4098	83.5171	117.4250	728.42	91.05	45.21
	100 μs	76.8974	77.0110	83.5818	99.0694	83.1057	107.8901	82.2173	118.6497	731.59	91.45	45.60
	1000 μs	77.0105	77.4571	83.0700	99.6486	83.1257	107.9600	82.6944	117.6960	794.48	99.31	53.47
Statistics	Sum	1156.32	1158.64	1264.73	1523.06	1254.87	1655.16	1252.28	1912.98	11178.04		
	Avg	77.09	77.24	84.32	101.54	83.66	110.34	83.49	127.53		43.66	
	Row Effect	31.24	31.40	38.47	55.69	37.81	64.50	37.64	81.69			

Table 3.6: Data Repetition Number 3

Architecture		Control		Brand X		Push		Pull		Statistics		Column
Packet Freq.	Query Interval	Timer	Event	Timer	Event	Timer	Event	Timer	Event	Sum	Avg	Effect
1 μs	1 μs	74.6086	74.7759	83.4771	105.4876	83.3735	114.7870	83.5097	153.4513	711.24	88.90	43.06
	10 μs	74.8206	75.8222	80.9769	96.6602	81.8332	109.0936	81.1173	115.8348	711.21	88.90	43.06
	100 μs	74.6370	75.8285	80.8958	96.0505	79.9153	105.6297	80.0673	118.1878	716.16	89.52	43.68
	1000 μs	76.1315	75.0721	81.1126	98.1220	80.5281	105.3614	79.7764	115.1336	773.47	96.68	50.84
10 μs	1 μs	74.3377	75.8327	83.4103	104.6081	84.6397	114.5087	82.9274	157.4345	717.96	89.75	43.90
	10 μs	74.5971	74.4271	81.8627	97.4051	79.8154	106.1799	80.3735	111.8517	707.85	88.48	42.64
	100 μs	75.0726	74.5587	80.1852	97.1604	80.1925	105.2025	81.1526	114.3230	706.51	88.31	42.47
	1000 μs	75.6707	74.9984	80.6444	96.7480	80.7717	104.7454	79.8180	124.5634	777.70	97.21	51.37
100 μs	1 μs	75.6355	74.8964	83.6049	104.9826	83.4687	114.2025	83.8275	156.1789	704.78	88.10	42.25
	10 μs	74.9164	75.6144	80.4356	98.4982	80.8806	105.9832	79.5765	116.1962	712.31	89.04	43.19
	100 μs	76.2567	75.3025	79.9968	96.0568	80.1410	105.7244	80.0952	118.7322	712.10	89.01	43.17
	1000 μs	75.4771	75.1757	79.8185	96.4061	80.7217	105.0894	80.1262	111.9648	776.80	97.10	51.26
1000 μs	1 μs	75.5565	74.2278	84.4282	105.6433	83.5928	113.5792	84.1731	151.4461	712.68	89.08	43.24
	10 μs	74.5771	75.5124	80.7622	96.4477	81.9653	107.5913	80.4887	114.0068	713.17	89.15	43.30
	100 μs	76.5413	77.9100	80.2683	96.3425	80.2830	105.4082	80.5345	115.8774	711.35	88.92	43.07
	1000 μs	74.8843	75.4077	80.3367	96.8112	80.5250	106.4077	80.0742	118.2315	713.17	89.15	43.30
Statistics	Sum	1129.11	1130.59	1218.74	1477.94	1219.27	1614.71	1214.13	1859.96	10864.45		
	Avg	75.27	75.37	81.25	98.53	81.28	107.65	80.94	124.00		42.44	
	Row Effect	29.43	29.53	35.40	52.68	35.44	61.80	35.10	78.15			

Table 3.7: Interactions Between Factors A and B

AB Interactions	B L1	B L2
A L1	11.1057	-11.1057
A L2	2.4394	-2.4394
A L3	-2.3004	2.3004
A L4	-11.2447	11.2447

Table 3.8: Interactions Between Factors A and C

AC Interactions	C L1	C L2	C L3	C L4
A L1	0.0756	-0.3272	0.0832	0.1684
A L2	-0.1550	-0.0107	0.2000	-0.0342
A L3	0.2455	-0.4704	0.0600	0.1649
A L4	-0.1660	0.8084	-0.3432	-0.2992

Table 3.9: Interactions Between Factors A and D

AD Interactions	D L1	D L2	D L3	D L4
A L1	-5.9211	1.9361	2.3064	1.6786
A L2	-1.8087	0.9058	0.2714	0.6315
A L3	-1.9220	1.4473	0.3683	0.1065
A L4	9.6518	-4.2892	-2.9461	-2.4166

Table 3.10: Interactions Between Factors B and C

BC Interactions	C L1	C L2	C L3	C L4
B L1	0.1147	0.0727	-0.2374	0.0500
B L2	-0.1147	-0.0727	0.2374	-0.0500

Table 3.11: Interactions Between Factors B and D

BD Interactions	D L1	D L2	D L3	D L4
B L1	-4.3267	1.3961	1.3085	1.6221
B L2	4.3267	-1.3961	-1.3085	-1.6221

Table 3.12: Interactions Between Factors C and D

CD Interactions	D L1	D L2	D L3	D L4
C L1	-0.1410	0.1911	0.2480	-0.2981
C L2	-0.0802	-0.3437	-0.4525	0.8765
C L3	0.2251	0.0253	-0.1171	-0.1334
C L4	-0.0039	0.1273	0.3216	-0.4450

Table 3.13: Interactions Between Factors A, B and C

ABC	Interactions	C L1	C L2	C L3	C L4
A L1	B L1	-0.1548	-0.0027	0.2623	-0.1049
	B L4	0.1548	0.0027	-0.2623	0.1049
A L2	B L1	0.1079	0.0305	-0.1620	0.0236
	B L4	-0.1079	-0.0305	0.1620	-0.0236
A L3	B L1	0.0420	-0.3379	0.1655	0.1304
	B L4	-0.0420	0.3379	-0.1655	-0.1304
A L4	B L1	0.0049	0.3102	-0.2659	-0.0491

Table 3.14: Interactions Between Factors A, B and D

ABD	Interactions	D L1	D L2	D L3	D L4
A L1	B L1	4.4315	-1.5009	-1.3519	-1.5787
	B L4	-4.4315	1.5009	1.3519	1.5787
A L2	B L1	2.2717	-0.8489	-0.5753	-0.8475
	B L4	-2.2717	0.8489	0.5753	0.8475
A L3	B L1	2.3953	-1.3667	-0.5077	-0.5209
	B L4	-2.3953	1.3667	0.5077	0.5209
A L4	B L1	-9.0985	3.7165	2.4349	2.9471
	B L4	9.0985	-3.7165	-2.4349	-2.9471

Table 3.15: Interactions Between Factors A, C and D

ACD	Interactions	D L1	D L2	D L3	D L4
A L1	C L1	0.0411	0.0007	-0.3434	0.3016
	C L2	0.3145	0.2722	0.1745	-0.7612
	C L3	-0.2030	-0.1415	0.1499	0.1946
	C L4	-0.1526	-0.1313	0.0190	0.2650
A L2	C L1	-0.4500	-0.4822	0.2548	0.6774
	C L2	-0.2002	0.9380	0.0088	-0.7465
	C L3	-0.3152	0.1472	-0.1299	0.2979
	C L4	0.9653	-0.6030	-0.1336	-0.2287
A L3	C L1	-0.1615	-0.0159	-0.4169	0.5942
	C L2	-0.0031	0.3774	0.5569	-0.9312
	C L3	0.2946	-0.2164	0.1340	-0.2122
	C L4	-0.1300	-0.1451	-0.2740	0.5491
A L4	C L1	0.5703	0.4973	0.5055	-1.5732
	C L2	-0.1113	-1.5876	-0.7401	2.4390
	C L3	0.2236	0.2107	-0.1540	-0.2803
	C L4	-0.6827	0.8795	0.3886	-0.5854

Table 3.16: Interactions Between Factors B, C and D

BCD	Interactions	D L1	D L2	D L3	D L4
C L1	B L1	-0.0898	-0.1203	-0.1918	0.4019
	B L4	0.0898	0.1203	0.1918	-0.4019
C L2	B L1	-0.4220	0.1242	0.0662	0.2316
	B L4	0.4220	-0.1242	-0.0662	-0.2316
C L3	B L1	0.0579	0.1152	0.2117	-0.3848
	B L4	-0.0579	-0.1152	-0.2117	0.3848
C L4	B L1	0.4539	-0.1191	-0.0862	-0.2486
	B L4	-0.4539	0.1191	0.0862	0.2486

Appendix A

Cross-Layer Test Harness Reference Manual 1.1

Cross-Layer Test Harness Class Hierarchical Index

Cross-Layer Test Harness File Index

Cross-Layer Test Harness Documentation

A.1 appClient Class Reference

Experimental workload generation class. This class controls the flow of network traffic during performance analysis experiments, including packet size, packet frequency and transport protocol utilization.

`#include <appClient.h>`

Public Member Functions

- **appClient** ()

  Constructor Sets the default values of the client.

- ∼**appClient** ()

  Deconstructor Cleanup and free remaining memory.

- void **setTCPUDP** (const char ∗name)

  Public Function Set TCP or UDP traffic.

- void **setLogFile** (const char ∗name)

  Public Function Setter for the log file name.

- void **setAvgFile** (const char ∗name)

  Public Function Setter for the avg log file name.

- void **setPacketSize** (int newSize)

  Public Function Setter for the packet size.

- int **getPacketSize** ()

  Public Function Getter for the packet size.

- void **setPacketInterval** (int newInterval)

  Public Function Setter for the packet interval.

- int **getPacketInterval** ()

  Public Function Getter for the packet interval.

- void **setBurstSize** (int newBurst)

  Public Function Setter for the burst size.

- int **getBurstSize** ()

  Public Function Getter for the burst size.

- void **generateWorkload** ()

  Public Function Generates workload consistent with the predefined settings.

- void **socketConfiguration** ()

  Public Function Opens and listens on a socket for configuration and initialization calls. This socket is utilized by the GUI configuration utility.

Private Member Functions

- void **log** (char *buf, int type)

  Private Function Log function. Logs data to a file or stdout.

Private Attributes

- int **packetSize**

  TCP packet size in bytes.

- int **packetInterval**

  Interval at which packets are sent in microseconds.

- int **burstSize**

  Number of packets sent during the course of the test.

- int **nextSize**

  Temporary integer used in syncing the client and server packet size.

- int **recieve**
- char * **logFile**

  Log file name, this is sent to the server during initialization.

- char * **avgFile**

Data Structures

- struct **experiment**

  Private experiment structure.

A.1.1 Detailed Description

Traffic generation class.

appClient(p. 72) is a TCP traffic generation class. This call drives the workload generation as well as encompassing several workload configuration functions. The matching workload recipient class is **appServer**(p. 84).

Definition at line 41 of file appClient.h.

A.1.2 Constructor & Destructor Documentation

A.1.2.1 appClient::appClient ()

Constructor Sets the default values of the client.

Definition at line 14 of file appClient.cpp.

References avgFile, burstSize, logFile, nextSize, packetInterval, packetSize, and recieve.

A.1.2.2 appClient::∼appClient ()

Deconstructor Cleanup and free remaining memory.

Definition at line 36 of file appClient.cpp.

References avgFile, and logFile.

A.1.3 Member Function Documentation

A.1.3.1 void appClient::generateWorkload ()

Public Function Generates workload consistent with the predefined settings.

Definition at line 113 of file appClient.cpp.

References avgFile, burstSize, SocketException::description(), INIT_SIZE, logFile, MAX_SIZE, nextSize, packetInterval, packetSize, and ClientSocket::send_buffer_udp().

Referenced by main(), and socketConfiguration().

A.1.3.2 int appClient::getBurstSize ()

Public Function Getter for the burst size.

Returns:

int, umber packets sent when generating workload

Definition at line 88 of file appClient.cpp.

References burstSize.

A.1.3.3 int appClient::getPacketInterval ()

Public Function Getter for the packet interval.

Returns:

int, number of microseconds between workload packets.

Definition at line 71 of file appClient.cpp.

References packetInterval.

A.1.3.4 int appClient::getPacketSize ()

Public Function Getter for the packet size.

Returns:

int, number of bytes of in workload packet

Definition at line 54 of file appClient.cpp.

References packetSize.

A.1.3.5 void appClient::log (char * *buf*, int *type*) [private]

Private Function Log function. Logs data to a file or stdout.

Parameters:

buf character pointer to data to be logged

type integer flag, 0 -> stdout, 1 -> logfile

Definition at line 380 of file appClient.cpp.

References FILE, and STDOUT.

Referenced by socketConfiguration().

A.1.3.6 void appClient::setAvgFile (const char * *name*)

Public Function Setter for the avg log file name.

Parameters:

name character pointer, Name for the next log file

Definition at line 105 of file appClient.cpp.

References avgFile.

Referenced by main().

A.1.3.7 void appClient::setBurstSize (int *newBurst*)

Public Function Setter for the burst size.

Parameters:

newBurst int, number packets sent when generating workload

Definition at line 80 of file appClient.cpp.

References burstSize.

Referenced by main(), and socketConfiguration().

A.1.3.8 void appClient::setLogFile (const char * *name*)

Public Function Setter for the log file name.

Parameters:

name character pointer, Name for the next log file

Definition at line 96 of file appClient.cpp.

References logFile.

Referenced by main().

A.1.3.9 void appClient::setPacketInterval (int *newInterval*)

Public Function Setter for the packet interval.

Parameters:

newInterval int, time in microseconds between workload packets

Definition at line 63 of file appClient.cpp.

References packetInterval.

Referenced by main(), and socketConfiguration().

A.1.3.10 void appClient::setPacketSize (int *newSize*)

Public Function Setter for the packet size.

Parameters:

newSize int, number of bytes of in workload packet

Definition at line 45 of file appClient.cpp.

References nextSize.

Referenced by main(), and socketConfiguration().

A.1.3.11 void appClient::setTCPUDP (const char * *name*)

Public Function Set TCP or UDP traffic.

A.1.3.12 void appClient::socketConfiguration ()

Public Function Opens and listens on a socket for configuration and initialization calls. This socket is utilized by the GUI configuration utility.

The function is called by the local main method. Configuration packets are recieved as well as commands to generate workload.

On recieving a command to generate workload the **generateWorkload()**(p. 75) function is called and no more packets are recieved until the call has been completed.

Definition at line 222 of file appClient.cpp.

References ServerSocket::accept(), avgFile, generateWorkload(), INIT_SIZE, log(), logFile, MAX_SIZE, packetSize, ServerSocket::rec_buffer(), recieve, setBurstSize(), setPacketInterval(), setPacketSize(), and STDOUT.

A.1.4 Field Documentation

A.1.4.1 char* appClient::avgFile [private]

Log file name, this is sent to the server during initialization, This file contains the avg transmission times over the course of an expiriment

Definition at line 60 of file appClient.h.

Referenced by appClient(), generateWorkload(), setAvgFile(), socketConfiguration(), and ~appClient().

A.1.4.2 int appClient::burstSize [private]

Number of packets sent during the course of the test.

Definition at line 50 of file appClient.h.

Referenced by appClient(), generateWorkload(), getBurstSize(), and setBurstSize().

A.1.4.3 char* appClient::logFile [private]

Log file name, this is sent to the server during initialization.

Definition at line 57 of file appClient.h.

Referenced by appClient(), generateWorkload(), setLogFile(), socketConfiguration(), and ~appClient().

A.1.4.4 int appClient::nextSize [private]

Temporary integer used in syncing the client and server packet size.

Definition at line 52 of file appClient.h.

Referenced by appClient(), generateWorkload(), and setPacketSize().

A.1.4.5 int appClient::packetInterval [private]

Interval at which packets are sent in microseconds.

Definition at line 48 of file appClient.h.

Referenced by appClient(), generateWorkload(), getPacketInterval(), and setPacketInterval().

A.1.4.6 int appClient::packetSize [private]

TCP packet size in bytes.

Definition at line 46 of file appClient.h.

Referenced by appClient(), generateWorkload(), getPacketSize(), and socketConfiguration().

A.1.4.7 int appClient::recieve [private]

Recieve configuration flag, set to 0 to stop listening for configuration data and exit the program

Definition at line 55 of file appClient.h.

Referenced by appClient(), and socketConfiguration().

The documentation for this class was generated from the following files:

- **appClient.h**
- **appClient.cpp**

A.2 appClient::experiment Struct Reference

Private experiment structure.

Data Fields

- int **packetFrequency**
- int **queryInterval**
- int **packetSize**
- int **activationMechanism**
- int **architecture**
- int **state**

A.2.1 Detailed Description

Private experiment structure.

Stores the configuration settings for a specific experimental configuration. This struct is used to store the matrix on experiments in order to randamize the experiment order for statistical reasons.

Definition at line 78 of file appClient.h.

A.2.2 Field Documentation

A.2.2.1 int appClient::experiment::activationMechanism

activationMechanism stores the Timer or Event activation mechanism to be utilized

Definition at line 82 of file appClient.h.

A.2.2.2 int appClient::experiment::architecture

architecture states which of the architectures is to be used. Control, Brand X, Push, or Pull

Definition at line 83 of file appClient.h.

A.2.2.3 int appClient::experiment::packetFrequency

packetFrequency stores the number of microseconds between packets sent by the **appClient**(p. 72)

Definition at line 79 of file appClient.h.

A.2.2.4 int appClient::experiment::packetSize

packetSize states the size of the packet payload for the workload definition. This is set in number of bytes.

Definition at line 81 of file appClient.h.

A.2.2.5 int appClient::experiment::queryInterval

queryInterval stores the number of microseconds between information queries at the network protocol layers. Kernal timer granularity limites this to a millisecond delay not the microseconds the variable is set in.

Definition at line 80 of file appClient.h.

A.2.2.6 int appClient::experiment::state

states stores the status of the experiment, Not run, running, completed, errors

Definition at line 84 of file appClient.h.

The documentation for this struct was generated from the following file:

- **appClient.h**

A.3 appServer Class Reference

Traffic reception class.

`#include <appServer.h>`

Public Member Functions

- **appServer** ()

  Constructor.

- ∼**appServer** ()

  Deconstructor Cleanup and free remaining memory.

- void **recievePackets** ()

  Public Function.

- void **stopReceiving** ()

  Public Function.

Private Member Functions

- void **log** (char ∗buf, int type)

  Private Function Log function. Logs data to a file or stdout.

Private Attributes

- int **packetSize**

  TCP packet size in bytes.

- int **packetInterval**

  Interval at which packets are sent in microseconds.

- int **burstSize**

  Number of packets sent during the course of the test.

- int **nextSize**

  Temporary integer used in syncing the client and server packet size.

- int **recvPackets**

Receiving flag that governs when to stop listening at the socket.

- char * **logFile**

  Log file name, this is sent to the server during initialization.

- char * **avgFileName**

  Log file name, this is sent to the server during initialization.

Data Structures

- struct **packetInfo**

  Private Data Structure.

A.3.1 Detailed Description

Traffic reception class.

appServer(p. 84) is a TCP traffic generation class. This call receives workload packets from the **appClient**(p. 72) and documents the traffic statistics. The matching workload generation class is **appClient**(p. 72).

Definition at line 33 of file appServer.h.

A.3.2 Constructor & Destructor Documentation

A.3.2.1 appServer::appServer ()

Constructor.

Definition at line 7 of file appServer.cpp.

References avgFileName, and logFile.

A.3.2.2 appServer::~appServer ()

Deconstructor Cleanup and free remaining memory.

Definition at line 22 of file appServer.cpp.

References logFile.

A.3.3 Member Function Documentation

A.3.3.1 void appServer::log (char * *buf*, int *type*) [private]

Private Function Log function. Logs data to a file or stdout.

Parameters:

buf character pointer to data to be logged

type integer flag, 0 -> stdout, 1 -> logfile

Definition at line 34 of file appServer.cpp.

References FILE, and STDOUT.

Referenced by recievePackets().

A.3.3.2 void appServer::recievePackets ()

Public Function.

Initializes the socket and waits to recieve data from the client app. Once data is received the time payload is stripped and written to a log file with the time of packet reception.

Definition at line 62 of file appServer.cpp.

References ServerSocket::accept(), avgFileName, ServerSocket::BindUDP(), INIT_SIZE, log(), logFile, MAX_SIZE, appServer::packetInfo::number, appServer::packetInfo::packetSize, packetSize, ServerSocket::rec_buffer_udp(), appServer::packetInfo::Sec, STDOUT, and appServer::packetInfo::USec.

Referenced by main().

A.3.3.3 void appServer::stopReceiving ()

Public Function.

This function stops the server from recieving packets and closes the port. This function is called from the parent app

Definition at line 50 of file appServer.cpp.

References recvPackets.

A.3.4 Field Documentation

A.3.4.1 char* appServer::avgFileName [private]

Log file name, this is sent to the server during initialization.

Definition at line 64 of file appServer.h.

Referenced by appServer(), and recievePackets().

A.3.4.2 int appServer::burstSize [private]

Number of packets sent during the course of the test.

Definition at line 56 of file appServer.h.

A.3.4.3 char* appServer::logFile [private]

Log file name, this is sent to the server during initialization.

Definition at line 62 of file appServer.h.

Referenced by appServer(), recievePackets(), and ~appServer().

A.3.4.4 int appServer::nextSize [private]

Temporary integer used in syncing the client and server packet size.

Definition at line 58 of file appServer.h.

A.3.4.5 int appServer::packetInterval [private]

Interval at which packets are sent in microseconds.

Definition at line 54 of file appServer.h.

A.3.4.6 int appServer::packetSize [private]

TCP packet size in bytes.

Definition at line 52 of file appServer.h.

Referenced by recievePackets().

A.3.4.7 int appServer::recvPackets [private]

Receiving flag that governs when to stop listening at the socket.

Definition at line 60 of file appServer.h.

Referenced by stopReceiving().

The documentation for this class was generated from the following files:

- **appServer.h**

- **appServer.cpp**

A.4 appServer::packetInfo Struct Reference

Private Data Structure.

Data Fields

- int **number**
- double **Sec**
- double **USec**
- int **packetSize**

A.4.1 Detailed Description

Private Data Structure.

Data structure to store the packet information for the received server packets.

Definition at line 44 of file appServer.h.

A.4.2 Field Documentation

A.4.2.1 int appServer::packetInfo::number

Structure int value to store the number of the packet

Definition at line 45 of file appServer.h.

Referenced by appServer::recievePackets().

A.4.2.2 int appServer::packetInfo::packetSize

Structure int value to store the size of the packet

Definition at line 48 of file appServer.h.

Referenced by appServer::recievePackets().

A.4.2.3 double appServer::packetInfo::Sec

Structure double value to store the second the packet was received

Definition at line 46 of file appServer.h.

Referenced by appServer::recievePackets().

A.4.2.4 double appServer::packetInfo::USec

Structure double value to store the microsecond the packet was received

Definition at line 47 of file appServer.h.

Referenced by appServer::recievePackets().

The documentation for this struct was generated from the following file:

- **appServer.h**

A.5 ClientSocket Class Reference

Public Class.

`#include <ClientSocket.h>`

Inheritance diagram for ClientSocket::

Public Member Functions

- **ClientSocket** (std::string host, int port)

  Public Constructor.

- ∼**ClientSocket** ()

  Public DeConstructor.

- const **ClientSocket** & **operator**<< (const std::string &) const
- const **ClientSocket** & **operator**>> (std::string &) const
- const **ClientSocket** & **send_buffer** (const char ∗buf, int length) const

  Public Function.

- const **ClientSocket** & **send_buffer_udp** (char ∗buf, int length) const

  Public Function.

A.5.1 Detailed Description

Public Class.

Client **Socket**(p. 130) class. This class contains the interface for the client end of the socket connection. This is utilized during TCP and UDP connections to create a connection, send and receive data, and accept incomming connections.

Definition at line 19 of file ClientSocket.h.

A.5.2 Constructor & Destructor Documentation

A.5.2.1 ClientSocket::ClientSocket (std::string *host*, int *port*)

Public Constructor.

This is a constructor for the **ClientSocket**(p. 91) class. This creates a TCP socket for the specified host and connects to the host at the specified port.

Parameters:

host string Host to which the TCP and UDP connections will be established

port int Port at which the TCP connection will be bound and listen.

Definition at line 7 of file ClientSocket.cpp.

References Socket::connect(), and Socket::create().

A.5.2.2 ClientSocket::~ClientSocket () [inline]

Public DeConstructor.

This is the deconstructor for the CLientSocket class. This cleans up the socket connections and frees memory.

Definition at line 44 of file ClientSocket.h.

A.5.3 Member Function Documentation

A.5.3.1 const ClientSocket & ClientSocket::operator<< (const std::string &) const

Definition at line 22 of file ClientSocket.cpp.

References Socket::send().

A.5.3.2 const ClientSocket & ClientSocket::operator>> (std::string &) const

Definition at line 34 of file ClientSocket.cpp.

References Socket::recv().

A.5.3.3 const ClientSocket & ClientSocket::send_buffer (const char * *buf*, int *length*) const

Public Function.

Send a character array buffer via the TCP socket connection to the destination machine. This requires a TCP socket connection to be already established otherwise an error is returned. This will send a guarenteed length in bytes to the host machine. If the input buffer exceeds the length then the data will be truncated. If the input buffer is less than the length than the message will be padded.

Parameters:

buf character pointer to the send buffer.

length int length of data to send to the host machine.

Reimplemented from **Socket** (p. 136).

Definition at line 44 of file ClientSocket.cpp.

References Socket::send_buffer().

A.5.3.4 const ClientSocket & ClientSocket::send_buffer_udp (char * *buf*, int *length*) const

Public Function.

Send a character array buffer via the UDP socket connection to the destination machine. This requires a UDP socket connection to be already established otherwise an error is returned. This will send a guarenteed length in bytes to the host machine. If the input buffer exceeds the length then the data will be truncated. If the input buffer is less than the length than the message will be padded.

Parameters:

buf character pointer to the send buffer.

length int length of data to send to the host machine.

Reimplemented from **Socket** (p. 136).

Definition at line 55 of file ClientSocket.cpp.

References Socket::send_buffer_udp().

Referenced by appClient::generateWorkload().

The documentation for this class was generated from the following files:

- **ClientSocket.h**
- **ClientSocket.cpp**

A.6 configureTab Class Reference

Public Class.

`#include <tabdialog.h>`

Public Slots

- void **startBrandX** ()

  Public QT Slot Function.

- void **startPush** ()

  Public QT Slot Function.

- void **startPull** ()

  Public QT Slot Function.

- void **stopBrandX** ()

  Public QT Slot Function.

- void **stopPush** ()

  Public QT Slot Function.

- void **stopPull** ()

  Public QT Slot Function.

- void **startWorkload** ()

  Public Slot Function.

- void **startBaseline** ()

  Public QT Slot Function.

- void **stopBaseline** ()

  Public QT Slot Function.

- void **setPacketSize** ()

  Public QT Slot Function.

- void **setQotDelay** ()

  Public QT Slot Function.

- void **setPhyDelay** ()

  Public QT Slot Function.

- void **setPacketFreq** ()

  Public QT Slot Function.

- void **setTestDuration** ()

  Public QT Slot Function.

- void **setLogFile** ()

  Public QT Slot Function.

- void **setTrigger** ()

  Public QT Slot Function.

- void **setEventInterval** ()

  Public QT Slot Function.

- void **setTriggerFrequency** ()

  Public QT Slot Function.

Public Member Functions

- **configureTab** (**model** ∗m, QWidget ∗parent=0)

  Private Constructor Creates an instance of the **configureTab**(p. 95) *and sets us the appropriate initialization values. This also sets up all of the slot functions and the gui objects for the tab.*

- void **setModel** (**model** m)

  Private Function Sets the model for the **configureTab**(p. 95).

Private Member Functions

- void **sendProcCmd** (QString file, QString command)

  Private Function Sends commands to the proc file system. This function handles all of the interaction between the kernel level processes and the user level processes.

- void **startLayers** (QString regFunc)

  Private Function Starts the protocol layer query mechanism.

- void **registerLayers** (QString regFunc)

  Private Function Register the protocol layer Synchronous Pull functions.

- void **registerQoT** ()

  Private Function Register the protocol layer extension callback functions with the QoT stub layer.

- void **configureQoT** ()

  Private Function Setup the QoT protocol layer with the proper delay and query times.

- void **insertQoT** ()

  Private Function Insert QoT into the Linux protocol stack between the socket interface and the TCP & UDP layers.

- void **removeQoT** ()

  Private Function Stops protocol layer extensions from querying.

- void **startQoT** ()

  Private Function Start QoT traffic delay and information query timers.

- void **stopQoT** ()

  Private Function Stop QoT traffic delay and information query timers.

- void **unregisterQoT** ()

  Private Function Unregister the protocol layer extension callback functions with the QoT stub layer.

- void **initBrandx** ()

  Private Function Setup the Brand X protocol with the configuration.

- void **initPhy** ()

  Private Function Setup the physical layer stub with the proper delay and query times.

- void **startPhy** ()

  Private Function Start physical stub traffic delay and traffic monitoring.

- void **initApp** ()

  Private Function Configure the application protocol layer extension with the proper query interval.

- void **stopLayers** ()

  Private Function Stops protocol layer extensions from querying.

- void **cleanupLayers** ()

  Private Function Initializes the Brand X environment.

- void **cleanupBrandx** ()

  Private Function Cleans up Brand X protocols functionality.

- void **stopPhy** ()

  Private Function Stops physical layer delay and traffic monitoring.

- void **cleanupApp** ()

  Private Function Starts physical layer delay and traffic monitoring.

- void **setPacketSize** (int size)

  Public Set Method.

- void **appClientSocket** (QString data)

  Private Function Sends configuration packets to **appClient**(p. 72) *over port 30001.*

Private Attributes

- **model** ∗ **my_model**

  Private Variable.

A.6.1 Detailed Description

Public Class.

The **configureTab**(p. 95) class contains the graphical user interface code for the configuration tab. All controls and their respective slots and functions are contained in the **configureTab**(p. 95) class. This class is then utilizes within the **TabDialog**(p. 142) class.

Definition at line 230 of file tabdialog.h.

A.6.2 Constructor & Destructor Documentation

A.6.2.1 configureTab::configureTab (model * *m*, QWidget * *parent* = 0)

Private Constructor Creates an instance of the **configureTab**(p. 95) and sets us the appropriate initialization values. This also sets up all of the slot functions and the gui objects for the tab.

Parameters:

m model, New model to be used in conjunction with the **configureTab**(p. 95)

parent QWidget, Parent widget for the **configureTab**(p. 95), Optional

Definition at line 919 of file tabdialog.cpp.

References setEventInterval(), setLogFile(), setPacketFreq(), setPacketSize(), setPhyDelay(), setQotDelay(), setTestDuration(), setTrigger(), setTriggerFrequency(), startBaseline(), startBrandX(), startPull(), startPush(), startWorkload(), stopBaseline(), stopBrandX(), stopPull(), and stopPush().

A.6.3 Member Function Documentation

A.6.3.1 void configureTab::appClientSocket (QString *data*) [private]

Private Function Sends configuration packets to **appClient**(p. 72) over port 30001.

Parameters:

data QString, configuration data to be sent to **appClient**(p. 72)

Definition at line 380 of file tabdialog.cpp.

Referenced by initApp().

A.6.3.2 void configureTab::cleanupApp () [private]

Private Function Starts physical layer delay and traffic monitoring.

Definition at line 659 of file tabdialog.cpp.

A.6.3.3 void configureTab::cleanupBrandx () [private]

Private Function Cleans up Brand X protocols functionality.

Definition at line 567 of file tabdialog.cpp.

Referenced by stopBrandX().

A.6.3.4 void configureTab::cleanupLayers () [private]

Private Function Initializes the Brand X environment.

Definition at line 515 of file tabdialog.cpp.

References sendProcCmd().

Referenced by stopBrandX(), and stopPull().

A.6.3.5 void configureTab::configureQoT () [private]

Private Function Setup the QoT protocol layer with the proper delay and query times.

Definition at line 590 of file tabdialog.cpp.

References sendProcCmd().

Referenced by startBrandX(), startPull(), and startPush().

A.6.3.6 void configureTab::initApp () [private]

Private Function Configure the application protocol layer extension with the proper query interval.

Definition at line 632 of file tabdialog.cpp.

References appClientSocket().

Referenced by startWorkload().

A.6.3.7 void configureTab::initBrandx () [private]

Private Function Setup the Brand X protocol with the configuration.

Definition at line 559 of file tabdialog.cpp.

Referenced by startBrandX().

A.6.3.8 void configureTab::initPhy () [private]

Private Function Setup the physical layer stub with the proper delay and query times.

A.6.3.9 void configureTab::insertQoT () [private]

Private Function Insert QoT into the Linux protocol stack between the socket interface and the TCP & UDP layers.

Definition at line 606 of file tabdialog.cpp.

References sendProcCmd().

Referenced by startBrandX(), startPull(), and startPush().

A.6.3.10 void configureTab::registerLayers (QString *regFunc*) [private]

Private Function Register the protocol layer Synchronous Pull functions.

Parameters:

regFunc QString, Synchronouse Pull function

Definition at line 483 of file tabdialog.cpp.

References sendProcCmd().

Referenced by startPull().

A.6.3.11 void configureTab::registerQoT () [private]

Private Function Register the protocol layer extension callback functions with the QoT stub layer.

Definition at line 616 of file tabdialog.cpp.

References sendProcCmd().

Referenced by startPull().

A.6.3.12 void configureTab::removeQoT () [private]

Private Function Stops protocol layer extensions from querying.

Definition at line 611 of file tabdialog.cpp.

References sendProcCmd().

Referenced by stopBrandX(), stopPull(), and stopPush().

A.6.3.13 void configureTab::sendProcCmd (QString *file*, QString *command*) [private]

Private Function Sends commands to the proc file system. This function handles all of the interaction between the kernel level processes and the user level processes.

Parameters:

file QString, Name of the file that the command will be sent to.

command QString, Text that is to be forwarded to the kernel level process.

Definition at line 810 of file tabdialog.cpp.

Referenced by cleanupLayers(), configureQoT(), insertQoT(), registerLayers(), registerQoT(), removeQoT(), startLayers(), startPhy(), startQoT(), stopLayers(), stopPhy(), stopQoT(), and unregisterQoT().

A.6.3.14 void configureTab::setEventInterval () [slot]

Public QT Slot Function.

Changes the model value of the event driven interval as changes are made to the view.

Definition at line 870 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.15 void configureTab::setLogFile () [slot]

Public QT Slot Function.

Changes the model value of the log file as changes are made to the view.

Definition at line 847 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.16 void configureTab::setModel (model *m*)

Private Function Sets the model for the **configureTab**(p. 95).

Parameters:

m model, New model to be used in conjunction with the **configureTab**(p. 95)

A.6.3.17 void configureTab::setPacketFreq () [slot]

Public QT Slot Function.

Changes the model value of the packet frequency as changes are made to the view. This is called in connection with the packetFreq edit box

Definition at line 895 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.18 void configureTab::setPacketSize () [slot]

Public QT Slot Function.

Changes the model value of the packet size as changes are made to the view. This is called in connection with the packetSize edit box

Definition at line 829 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.19 void configureTab::setPacketSize (int *size*) [private]

Public Set Method.

This function sets the size of the network traffic in the model.

Parameters:

size Int, Size of packet in bytes

A.6.3.20 void configureTab::setPhyDelay () [slot]

Public QT Slot Function.

Changes the model value of the Physical layer delay as changes are made to the view. This is called in connection with the packetSize edit box

Definition at line 914 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.21 void configureTab::setQotDelay () [slot]

Public QT Slot Function.

Changes the model value of the QoT delay as changes are made to the view. This is called in connection with the packetSize edit box

Definition at line 905 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.22 void configureTab::setTestDuration () [slot]

Public QT Slot Function.

Changes the model value of the test duration as changes are made to the view.

Definition at line 838 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.23 void configureTab::setTrigger () [slot]

Public QT Slot Function.

Changes the model value of the trigger as changes are made to the view.

Definition at line 856 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.24 void configureTab::setTriggerFrequency () [slot]

Public QT Slot Function.

Changes the model value of the query interval as changes are made to the view.

Definition at line 879 of file tabdialog.cpp.

Referenced by configureTab().

A.6.3.25 void configureTab::startBaseline () [slot]

Public QT Slot Function.

Starts the control cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 662 of file tabdialog.cpp.

References startPhy().

Referenced by configureTab().

A.6.3.26 void configureTab::startBrandX () [slot]

Public QT Slot Function.

Starts the Brand X cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 700 of file tabdialog.cpp.

References configureQoT(), initBrandx(), insertQoT(), startLayers(), startPhy(), and startQoT().

Referenced by configureTab().

A.6.3.27 void configureTab::startLayers (QString *regFunc*) [private]

Private Function Starts the protocol layer query mechanism.

Parameters:

regFunc QString, Name of the function that the query mechanism should reference

Definition at line 450 of file tabdialog.cpp.

References sendProcCmd().

Referenced by startBrandX(), and startPush().

A.6.3.28 void configureTab::startPhy () [private]

Private Function Start physical stub traffic delay and traffic monitoring.

Definition at line 576 of file tabdialog.cpp.

References sendProcCmd().

Referenced by startBaseline(), startBrandX(), startPull(), and startPush().

A.6.3.29 void configureTab::startPull () [slot]

Public QT Slot Function.

Starts the Synchronous Pull cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 772 of file tabdialog.cpp.

References configureQoT(), insertQoT(), registerLayers(), registerQoT(), startPhy(), and startQoT().

Referenced by configureTab().

A.6.3.30 void configureTab::startPush () [slot]

Public QT Slot Function.

Starts the Synchronous Push cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 739 of file tabdialog.cpp.

References configureQoT(), insertQoT(), startLayers(), and startPhy().

Referenced by configureTab().

A.6.3.31 void configureTab::startQoT () [private]

Private Function Start QoT traffic delay and information query timers.

Definition at line 620 of file tabdialog.cpp.

References sendProcCmd().

Referenced by startBrandX(), and startPull().

A.6.3.32 void configureTab::startWorkload () [slot]

Public Slot Function.

This takes the interupt from the generate workload button and calls the appropriate workload generation utilities.

Definition at line 683 of file tabdialog.cpp.

References initApp().

Referenced by configureTab().

A.6.3.33 void configureTab::stopBaseline () [slot]

Public QT Slot Function.

Stops the control cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 687 of file tabdialog.cpp.

References stopPhy().

Referenced by configureTab().

A.6.3.34 void configureTab::stopBrandX () [slot]

Public QT Slot Function.

Stops the Brand X cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 720 of file tabdialog.cpp.

References cleanupBrandx(), cleanupLayers(), removeQoT(), stopLayers(), stopPhy(), and stopQoT().

Referenced by configureTab().

A.6.3.35 void configureTab::stopLayers () [private]

Private Function Stops protocol layer extensions from querying.

Definition at line 537 of file tabdialog.cpp.

References sendProcCmd().

Referenced by stopBrandX(), and stopPush().

A.6.3.36 void configureTab::stopPhy () [private]

Private Function Stops physical layer delay and traffic monitoring.

Definition at line 586 of file tabdialog.cpp.

References sendProcCmd().

Referenced by stopBaseline(), stopBrandX(), stopPull(), and stopPush().

A.6.3.37 void configureTab::stopPull () [slot]

Public QT Slot Function.

Stops the Synchronous Pull cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 793 of file tabdialog.cpp.

References cleanupLayers(), removeQoT(), stopPhy(), stopQoT(), and unregisterQoT().

Referenced by configureTab().

A.6.3.38 void configureTab::stopPush () [slot]

Public QT Slot Function.

Stops the Synchronous Push cross-layer architecture functionality includeing: Delays, querying, modification of the network protocol stack

Definition at line 757 of file tabdialog.cpp.

References removeQoT(), stopLayers(), and stopPhy().

Referenced by configureTab().

A.6.3.39 void configureTab::stopQoT () [private]

Private Function Stop QoT traffic delay and information query timers.

Definition at line 628 of file tabdialog.cpp.

References sendProcCmd().

Referenced by stopBrandX(), and stopPull().

A.6.3.40 void configureTab::unregisterQoT () [private]

Private Function Unregister the protocol layer extension callback functions with the QoT stub layer.

Definition at line 624 of file tabdialog.cpp.

References sendProcCmd().

Referenced by stopPull().

A.6.4 Field Documentation

A.6.4.1 model* configureTab::my_model [private]

Private Variable.

The my_model variable is a local instatiation of the Model class and stores the information for the **configureTab**(p. 95).

Definition at line 293 of file tabdialog.h.

The documentation for this class was generated from the following files:

- **tabdialog.h**
- **tabdialog.cpp**

A.7 model Class Reference

Public Model Class.

`#include <tabdialog.h>`

Public Member Functions

- **model** ()

  Public Constructor.

- QString **getPacketSize** ()

  Public Get Method.

- void **setPacketSize** (QString)

  Public Set Method.

- QString **getPacketFreq** ()

  Public Get Method.

- void **setPacketFreq** (QString)

  Public Set Method.

- QString **getTestDuration** ()

  Public Get Method.

- void **setTestDuration** (QString)

  Public Set Method.

- QString **getLogFile** ()

  Public Get Method.

- void **setLogFile** (QString)

  Public Set Method.

- QString **getTriggerMechanism** ()

  Public Get Method.

- void **setTriggerMechanism** (QString)

  Public Set Method.

- QString **getTimerRange** ()

  Public Get Method.

- void **setTimerRange** (QString)

  Public Set Method.

- QString **getTimerInterval** ()

  Public Get Method.

- void **setTimerInterval** (QString)

  Public Get Method.

- QString **getPhyDelay** ()

  Public Get Method.

- void **setPhyDelay** (QString)

  Public Set Method.

- QString **getQotDelay** ()

  Public Get Method.

- void **setQotDelay** (QString)

  Public Set Method.

Private Attributes

- QString **packetSize**

  packetSize stores the size in bytes of the workload packet payload

- QString **testDuration**

  testDuration stores the number of packets to send in a workload set

- QString **phyDelay**

  phyDelay stores the time in microseconds that the physical layer delays network packets before forwarding them on.

- QString **qotDelay**

  qotDelay stores the time in microseconds that QoT delays network packets before forwarding them on.

- QString **packetFreq**

  packetFreq stores the delay in microseconds between workload packets

- QString **logFile**

  logFile stores the name of the next log file, received from the gui control

- QString **avgFile**

  avgFile stores the name of the file to store the averages of the experiments

- QString **triggerMechanism**

  triggerMechanism stores the current trigger or activation mechanism for the experiment, Timer or Event.

- QString **timerRange**

  timerRange stores the probability of an Event driven activation mechanism to start an information query. The timerRange follows a uniform distribution

- QString **timerInterval**

  timerInterval stores the interval that a Timer driven activation mechanism starts an information query. This is set in microseconds but the Linux kernel is limited to milliseconds.

A.7.1 Detailed Description

Public Model Class.

The Model class contains all of the data and configuration for the graphical user interface. This data is utilized in the experiments run from the gui as well as in setting up an environment for experiments run from the proc interface.

Definition at line 76 of file tabdialog.h.

A.7.2 Constructor & Destructor Documentation

A.7.2.1 model::model ()

Public Constructor.

This is the constructor for the Model class. The Model class contains all of the configuration data for the Application Driver GUI

Definition at line 139 of file tabdialog.cpp.

References logFile, packetFreq, packetSize, phyDelay, qotDelay, testDuration, timerInterval, timerRange, and triggerMechanism.

A.7.3 Member Function Documentation

A.7.3.1 QString model::getLogFile ()

Public Get Method.

This function returns the log file name as specified in the model.

The logFile variable stores the name of the file that the test results will be stored in.

Definition at line 281 of file tabdialog.cpp.

References logFile.

A.7.3.2 QString model::getPacketFreq ()

Public Get Method.

This function returns the packet frequency as specified in the model.

Definition at line 186 of file tabdialog.cpp.

References packetFreq.

A.7.3.3 QString model::getPacketSize ()

Public Get Method.

This function returns the packet size as specified in the model.

The packetSize variable stores the size of the test data packet payload.

Definition at line 232 of file tabdialog.cpp.

References packetSize.

A.7.3.4 QString model::getPhyDelay ()

Public Get Method.

This function returns the physical layer network traffic delay time as specified in the model.

Definition at line 164 of file tabdialog.cpp.

References phyDelay.

A.7.3.5 QString model::getQotDelay ()

Public Get Method.

This function returns the QoT delay time as specified in the model.

Definition at line 208 of file tabdialog.cpp.

References qotDelay.

A.7.3.6 QString model::getTestDuration ()

Public Get Method.

This function returns the test duration as specified in the model.

The testDuration value stores the number of packets to send in a test.

Definition at line 255 of file tabdialog.cpp.

References testDuration.

A.7.3.7 QString model::getTimerInterval ()

Public Get Method.

This function returns the query interval for the Timer activation mechanism as specified in the model.

Definition at line 358 of file tabdialog.cpp.

References timerInterval.

A.7.3.8 QString model::getTimerRange ()

Public Get Method.

This function returns the timer range as specified in the model.

The timerRange variable stores the time

Definition at line 334 of file tabdialog.cpp.

References timerRange.

A.7.3.9 QString model::getTriggerMechanism ()

Public Get Method.

This function returns the trigger mechanism as specified in the model.

The triggerMechanism variable stores the activation mechanism to use in the test. 0 == Timer activation mechansim, 1 == Event activation mechanism.

Definition at line 308 of file tabdialog.cpp.

References triggerMechanism.

A.7.3.10 void model::setLogFile (QString *newFile*)

Public Set Method.

This function sets the log file name in the model.

The logFile variable stores the name of the file that the test results will be stored in.

Parameters:

newFile QString, Name of the new log file

Definition at line 295 of file tabdialog.cpp.

References logFile.

A.7.3.11 void model::setPacketFreq (QString *newFreq*)

Public Set Method.

This function sets the packet frequency in the model.

Parameters:

newFreq QString, Packet frequency in microseconds

Definition at line 197 of file tabdialog.cpp.

References packetFreq.

A.7.3.12 void model::setPacketSize (QString *newSize*)

Public Set Method.

This function sets the packet size delay time in the model.

Parameters:

newSize QString, Size of the test data packet payload.

Definition at line 243 of file tabdialog.cpp.

References packetSize.

A.7.3.13 void model::setPhyDelay (QString *newDelay*)

Public Set Method.

This function sets the physical layer network traffic delay time in the model.

Parameters:

newDelay QString, delay in microseconds

Definition at line 175 of file tabdialog.cpp.

References phyDelay.

A.7.3.14 void model::setQotDelay (QString *newDelay*)

Public Set Method.

This function sets the QoT delay time in the model.

Parameters:

newDelay QString, QoT delay time in microseconds

Definition at line 219 of file tabdialog.cpp.

References qotDelay.

A.7.3.15 void model::setTestDuration (QString *newDuration*)

Public Set Method.

This function sets the test duration in the model.

The testDuration value stores the number of packets to send in a test.

Parameters:

newDuration QString, number of packets in the workload burst

Definition at line 268 of file tabdialog.cpp.

References testDuration.

A.7.3.16 void model::setTimerInterval (QString *newInterval*)

Public Get Method.

This function returns the query interval for the Timer activation mechanism as specified in the model.

Parameters:

newInterval QString, the new timer interval

Definition at line 370 of file tabdialog.cpp.

References timerInterval.

A.7.3.17 void model::setTimerRange (QString *newRange*)

Public Set Method.

This function sets the timer range in the model. The timer range specifies the Event activation mechanism query interval.

Parameters:

newRange QString, new time

Definition at line 346 of file tabdialog.cpp.

References timerRange.

A.7.3.18 void model::setTriggerMechanism (QString *newTrigger*)

Public Set Method.

This function sets the trigger mechanism in the model.

The triggerMechanism variable stores the activation mechanism to use in the test. 0 == Timer activation mechansim, 1 == Event activation mechanism.

Parameters:

newTrigger QString, new trigger mechanism

Definition at line 322 of file tabdialog.cpp.

References triggerMechanism.

A.7.4 Field Documentation

A.7.4.1 QString model::avgFile [private]

avgFile stores the name of the file to store the averages of the experiments

Definition at line 118 of file tabdialog.h.

A.7.4.2 QString model::logFile [private]

logFile stores the name of the next log file, received from the gui control

Definition at line 116 of file tabdialog.h.

Referenced by getLogFile(), model(), and setLogFile().

A.7.4.3 QString model::packetFreq [private]

packetFreq stores the delay in microseconds between workload packets

Definition at line 112 of file tabdialog.h.

Referenced by getPacketFreq(), model(), and setPacketFreq().

A.7.4.4 QString model::packetSize [private]

packetSize stores the size in bytes of the workload packet payload

Definition at line 104 of file tabdialog.h.

Referenced by getPacketSize(), model(), and setPacketSize().

A.7.4.5 QString model::phyDelay [private]

phyDelay stores the time in microseconds that the physical layer delays network packets before forwarding them on.

Definition at line 108 of file tabdialog.h.

Referenced by getPhyDelay(), model(), and setPhyDelay().

A.7.4.6 QString model::qotDelay [private]

qotDelay stores the time in microseconds that QoT delays network packets before forwarding them on.

Definition at line 110 of file tabdialog.h.

Referenced by getQotDelay(), model(), and setQotDelay().

A.7.4.7 QString model::testDuration [private]

testDuration stores the number of packets to send in a workload set

Definition at line 106 of file tabdialog.h.

Referenced by getTestDuration(), model(), and setTestDuration().

A.7.4.8 QString model::timerInterval [private]

timerInterval stores the interval that a Timer driven activation mechanism starts an information query. This is set in microseconds but the Linux kernel is limited to milliseconds.

Definition at line 126 of file tabdialog.h.

Referenced by getTimerInterval(), model(), and setTimerInterval().

A.7.4.9 QString model::timerRange [private]

timerRange stores the probability of an Event driven activation mechanism to start an information query. The timerRange follows a uniform distribution

Definition at line 124 of file tabdialog.h.

Referenced by getTimerRange(), model(), and setTimerRange().

A.7.4.10 QString model::triggerMechanism [private]

triggerMechanism stores the current trigger or activation mechanism for the experiment, Timer or Event.

Definition at line 122 of file tabdialog.h.

Referenced by getTriggerMechanism(), model(), and setTriggerMechanism().

The documentation for this class was generated from the following files:

- **tabdialog.h**
- **tabdialog.cpp**

A.8 name_value Struct Reference

Internal data storage structure.

Data Fields

- char **name** [80]
- int **value**

A.8.1 Detailed Description

Internal data storage structure.

A stucture to hold name-value pairs of environment information within Brand X.

Definition at line 28 of file brandx.c.

A.8.2 Field Documentation

A.8.2.1 char name_value::name[80]

Character pointer stores name value

Definition at line 30 of file brandx.c.

A.8.2.2 int name_value::value

Integer stores data value

Definition at line 32 of file brandx.c.

The documentation for this struct was generated from the following file:

- **brandx.c**

A.9 packetData Struct Reference

Private Data Structure.

Data Fields

- int **rxBytes**
- int **txBytes**
- int **rxErrors**
- int **txErrors**
- int **packets**
- int **dropped**

A.9.1 Detailed Description

Private Data Structure.

Data structure to store the packet information for the received server packets. This structure is utilized to store the current information for the local protocol layer. This acts as a caching mechanism for data for use with a Synchronized Push or Pull architecture.

Definition at line 181 of file af_inet.c.

A.9.2 Field Documentation

A.9.2.1 int packetData::dropped

Stores the number of packets dropped in this protocol layer up to this point

Definition at line 187 of file af_inet.c.

Referenced by appEnvironmentUpdate(), ipEnvironmentUpdate(), loEnvironment-Update(), tcpEnvironmentUpdate(), and udpEnvironmentUpdate().

A.9.2.2 int packetData::packets

Stores the number of packets transmitted up to this point

Definition at line 186 of file af_inet.c.

Referenced by appEnvironmentUpdate(), ipEnvironmentUpdate(), loEnvironment-Update(), tcpEnvironmentUpdate(), and udpEnvironmentUpdate().

A.9.2.3 int packetData::rxBytes

Stores the number of bytes of data recieved up to this point

Definition at line 182 of file af_inet.c.

Referenced by appEnvironmentUpdate(), ipEnvironmentUpdate(), loEnvironmentUpdate(), tcpEnvironmentUpdate(), and udpEnvironmentUpdate().

A.9.2.4 int packetData::rxErrors

Stores any errors that occur during the recieve process

Definition at line 184 of file af_inet.c.

Referenced by appEnvironmentUpdate(), ipEnvironmentUpdate(), loEnvironmentUpdate(), tcpEnvironmentUpdate(), and udpEnvironmentUpdate().

A.9.2.5 int packetData::txBytes

Stores the number of bytes of data transmitted up to this point

Definition at line 183 of file af_inet.c.

Referenced by appEnvironmentUpdate(), ipEnvironmentUpdate(), loEnvironmentUpdate(), tcpEnvironmentUpdate(), and udpEnvironmentUpdate().

A.9.2.6 int packetData::txErrors

Stores any errors that occur during the transmission process

Definition at line 185 of file af_inet.c.

Referenced by appEnvironmentUpdate(), ipEnvironmentUpdate(), loEnvironmentUpdate(), tcpEnvironmentUpdate(), and udpEnvironmentUpdate().

The documentation for this struct was generated from the following files:

- **af_inet.c**
- **ip_output.c**
- **loopback.c**
- **tcp.c**
- **udp.c**

A.10 procTab Class Reference

Public Class.

`#include <tabdialog.h>`

Public Slots

- void **sendProcCmd** ()

  Public Slot.

Public Member Functions

- **procTab** (**model** *m, QWidget *parent=0)

  Private Constructor Creates an instance of the **procTab**(p. 122) *and sets us the appropriate initialization values. This also sets up all of the slot functions and the gui objects for the tab.*

Private Attributes

- **model** * **my_model**

  Private Variable.

- QString **file**

  Private Variable.

- QString **command**

  Private Variable.

A.10.1 Detailed Description

Public Class.

The **procTab**(p. 122) class contains the graphical user interface code for the proc interaction tab. All controls and their respective slots and functions are contained in the **procTab**(p. 122) class. This class is then utilizes within the **TabDialog**(p. 142) class.

Definition at line 172 of file tabdialog.h.

A.10.2 Constructor & Destructor Documentation

A.10.2.1 procTab::procTab (model * *m*, QWidget * *parent* = 0)

Private Constructor Creates an instance of the **procTab**(p. 122) and sets us the appropriate initialization values. This also sets up all of the slot functions and the gui objects for the tab.

Parameters:

m model, New model to be used in conjunction with the **configureTab**(p. 95)

parent QWidget, Parent widget for the **configureTab**(p. 95), Optional

Definition at line 96 of file tabdialog.cpp.

References sendProcCmd().

A.10.3 Member Function Documentation

A.10.3.1 void procTab::sendProcCmd () [slot]

Public Slot.

This function writes the text in the procCmdEdit text box to the /proc file listed in the procFileEdit text box.

Definition at line 57 of file tabdialog.cpp.

References FILE.

Referenced by procTab().

A.10.4 Field Documentation

A.10.4.1 QString procTab::command [private]

Private Variable.

The command string stores the data that is to be passed to the kernel process via the proc file system.

Definition at line 204 of file tabdialog.h.

A.10.4.2 QString procTab::file [private]

Private Variable.

The file variable stores the current file in the linux proc file system that configuration data will be stored at.

Definition at line 198 of file tabdialog.h.

A.10.4.3 model* procTab::my_model [private]

Private Variable.

The my_model variable is a local instatiation of the Model class and stores the information for the **configureTab**(p. 95).

Definition at line 192 of file tabdialog.h.

The documentation for this class was generated from the following files:

- **tabdialog.h**
- **tabdialog.cpp**

A.11 ServerSocket Class Reference

Public Class.

`#include <ServerSocket.h>`

Inheritance diagram for ServerSocket::

Public Member Functions

- **ServerSocket** (int port)

  Public Constructor.

- **ServerSocket** ()

  Public Constructor.

- ∼**ServerSocket** ()

  Public DeConstructor.

- void **BindUDP** (int port)

  Public Function.

- const **ServerSocket** & **operator**<< (const std::string &) const
- const **ServerSocket** & **operator**>> (std::string &) const
- int **rec_buffer** (char ∗buf, int length) const

  Public Function.

- int **rec_buffer_udp** (char ∗buf, int length) const

  Public Function.

- void **accept** (**ServerSocket** &)

  Public Function.

A.11.1 Detailed Description

Public Class.

Server **Socket**(p. 130) class. This class contains the interface for the server end of the socket connection. This is utilized during TCP and UDP connections to create a connection, bind to a port, send and receive data, and accept incomming connections.

Definition at line 19 of file ServerSocket.h.

A.11.2 Constructor & Destructor Documentation

A.11.2.1 ServerSocket::ServerSocket (int *port*)

Public Constructor.

This is a constructor for the **ServerSocket**(p. 125) class. This creates a TCP connection at the input port and a UDP connection at a predefined port number. The TCP port is then bound and set to listen at the specificed port number.

Parameters:

port int Port at which the TCP connection will be bound and listen.

Definition at line 7 of file ServerSocket.cpp.

References Socket::bind(), Socket::create(), and Socket::listen().

A.11.2.2 ServerSocket::ServerSocket ()

Public Constructor.

This is a constructor for the **ServerSocket**(p. 125) class. This creates a UDP connection at a predefined port number.

Definition at line 26 of file ServerSocket.cpp.

References Socket::create().

A.11.2.3 ServerSocket::∼ServerSocket ()

Public DeConstructor.

This is the deconstructor for the **ServerSocket**(p. 125) class. This cleans up the socket connections and frees memory.

Definition at line 35 of file ServerSocket.cpp.

A.11.3 Member Function Documentation

A.11.3.1 void ServerSocket::accept (ServerSocket &)

Public Function.

Sets the input socket to accept incomming socket connections. This function is used with the TCP socket, m_sock.

Definition at line 96 of file ServerSocket.cpp.

References Socket::accept().

Referenced by appServer::recievePackets(), and appClient::socketConfiguration().

A.11.3.2 void ServerSocket::BindUDP (int *port*)

Public Function.

This function binds the UDP connection to the specified port.

Parameters:

port int Port that the UDP socket will utilize.

Definition at line 40 of file ServerSocket.cpp.

References Socket::bindUDP().

Referenced by appServer::recievePackets().

A.11.3.3 const ServerSocket & ServerSocket::operator<< (const std::string &) const

Definition at line 49 of file ServerSocket.cpp.

References Socket::send().

A.11.3.4 const ServerSocket & ServerSocket::operator>> (std::string &) const

Definition at line 61 of file ServerSocket.cpp.

References Socket::recv().

A.11.3.5 int ServerSocket::rec_buffer (char * *buf*, int *length*) const

Public Function.

Receive a buffer of data from the host machine via a TCP connection. The function will block until the length of data provided by the parameter 'length' is received.

Parameters:

buf character pointer to the buffer for the received data.

length int length of data to expect from the host machine

Definition at line 71 of file ServerSocket.cpp.

References Socket::recv_buffer().

Referenced by appClient::socketConfiguration().

A.11.3.6 int ServerSocket::rec_buffer_udp (char * *buf*, int *length*) const

Public Function.

Receive a buffer of data from the host machine via a UDP socket connection. The function will block until the length of data provided by the parameter 'length' is received.

Parameters:

buf character pointer to the buffer for the received data.

length int length of data to expect from the host machine

Definition at line 83 of file ServerSocket.cpp.

References Socket::recv_buffer_udp().

Referenced by appServer::recievePackets().

The documentation for this class was generated from the following files:

- **ServerSocket.h**
- **ServerSocket.cpp**

A.12 Socket Class Reference

Public **Socket**(p. 130) Class.

`#include <Socket.h>`

Inheritance diagram for Socket::

Public Member Functions

- **Socket** ()

  Public Constructor.

- ∼**Socket** ()

  Public DeConstructor.

- bool **create** ()

  Public Function.

- bool **bind** (const int port)

  Public Function.

- bool **createUDP** ()

  Public Function.

- bool **bindUDP** (const int port)

  Public Function.

- bool **listen** () const

  Public Function.

- bool **accept** (**Socket** &) const

  Public Function.

- bool **connect** (const std::string host, const int port)

  Public Function.

- bool **send** (const std::string buffer) const

  Public Function.

- int **recv** (std::string &buffer) const

  Public Function.

- bool **send_buffer** (const char ∗buf, int length) const

  Public Function.

- int **recv_buffer** (char ∗retbuf, int length) const

  Public Function.

- bool **send_buffer_udp** (char ∗buf, int length) const

  Public Function.

- int **recv_buffer_udp** (char ∗retbuf, int length) const

  Public Function.

- void **set_non_blocking** (const bool blocking)

  Public Function.

- bool **is_valid** () const

  Public Function.

- bool **is_udp_valid** () const

  Public Function.

Private Attributes

- int **m_sock**

  Private variable.

- sockaddr_in **m_addr**

  Private variable.

- int **udp_sock**

  Private variable.

- sockaddr_in **udp_addr**

  Private variable.

A.12.1 Detailed Description

Public **Socket**(p. 130) Class.

This class handles all of the low level socket connection functionality, including: creation, bind, accept, listen, send and receive.

Definition at line 29 of file Socket.h.

A.12.2 Constructor & Destructor Documentation

A.12.2.1 Socket::Socket ()

Public Constructor.

This is the constructor for the **Socket**(p. 130) class

Definition at line 12 of file Socket.cpp.

References m_addr, and udp_addr.

A.12.2.2 Socket::~Socket ()

Public DeConstructor.

This is the deconstructor for the **Socket**(p. 130) class

Definition at line 21 of file Socket.cpp.

References is_udp_valid(), is_valid(), m_sock, and udp_sock.

A.12.3 Member Function Documentation

A.12.3.1 bool Socket::accept (Socket &) const

Public Function.

Sets the input socket to accept incomming socket connections. This function is used with the TCP socket, m_sock.

Definition at line 144 of file Socket.cpp.

References m_addr, and m_sock.

Referenced by ServerSocket::accept().

A.12.3.2 bool Socket::bind (const int *port*)

Public Function.

Binds the TCP socket to the input port. Returns zero if successful.

Parameters:

port int port to bind the TCP socket

Definition at line 78 of file Socket.cpp.

References is_udp_valid(), is_valid(), m_addr, and m_sock.

Referenced by bindUDP(), and ServerSocket::ServerSocket().

A.12.3.3 bool Socket::bindUDP (const int *port*)

Public Function.

Binds the UDP socket to the input port. Returns zero if successful.

Parameters:

port int port to bind the UDP socket

Definition at line 105 of file Socket.cpp.

References bind(), is_udp_valid(), udp_addr, and udp_sock.

Referenced by ServerSocket::BindUDP().

A.12.3.4 bool Socket::connect (const std::string *host*, const int *port*)

Public Function.

This function causes the TCP socket, m_sock, to connect to the input host at the input port.

Parameters:

host string Host machine to connect to

port int Port to use to connect to the host machine

Definition at line 290 of file Socket.cpp.

References is_valid(), m_addr, and m_sock.

Referenced by ClientSocket::ClientSocket().

A.12.3.5 bool Socket::create ()

Public Function.

This method instantiates socket connections objects and sets any applicable settings. Two connections are created, namely m_sock, a TCP SOCK_STREAM socket, and udp_-sock, a UDP SOCK_DGRAM socket. The sockets are set to reuse addresses and with no delay in order to minimize packet agglomeration during network processing.

Definition at line 30 of file Socket.cpp.

References is_udp_valid(), is_valid(), m_sock, and udp_sock.

Referenced by ClientSocket::ClientSocket(), and ServerSocket::ServerSocket().

A.12.3.6 bool Socket::createUDP ()

Public Function.

This method instantiates a socket connection objects and sets any applicable settings for a UDP socket connection. udp_sock, a UDP SOCK_DGRAM socket is created. The socket is set to reuse addresses and with the 'nodelay' flag in order to minimize packet agglomeration during network processing.

Definition at line 58 of file Socket.cpp.

References is_udp_valid(), and udp_sock.

A.12.3.7 bool Socket::is_udp_valid () const [inline]

Public Function.

Determines if the UDP socket is a valided initialized socket

Definition at line 224 of file Socket.h.

References udp_sock.

Referenced by bind(), bindUDP(), create(), createUDP(), and ∼Socket().

A.12.3.8 bool Socket::is_valid () const [inline]

Public Function.

Determines if the TCP socket is a valided initialized socket

Definition at line 217 of file Socket.h.

References m_sock.

Referenced by bind(), connect(), create(), listen(), and ∼Socket().

A.12.3.9 bool Socket::listen () const

Public Function.

Sets the TCP socket connection to listen with the number of possible connections of MAXCONNECTIONS

Definition at line 126 of file Socket.cpp.

References is_valid(), m_sock, and MAXCONNECTIONS.

Referenced by ServerSocket::ServerSocket().

A.12.3.10 int Socket::recv (std::string & *buffer*) const

Public Function.

Receive a buffer of data from the host machine. The maximum data size to receive is 66,000 bytes. The received data is returned in s.

Parameters:

buffer string buffer the buffer for the received data.

Definition at line 209 of file Socket.cpp.

References m_sock, and MAXRECV.

Referenced by ServerSocket::operator>>(), ClientSocket::operator>>(), and recv_buffer().

A.12.3.11 int Socket::recv_buffer (char * *retbuf*, int *length*) const

Public Function.

Receive a buffer of data from the host machine via a TCP connection. The function will block until the length of data provided by the parameter 'length' is received.

Parameters:

retbuf character pointer to the buffer for the received data.

length int length of data to expect from the host machine

Definition at line 235 of file Socket.cpp.

References m_sock, and recv().

Referenced by ServerSocket::rec_buffer().

A.12.3.12 int Socket::recv_buffer_udp (char * *retbuf*, int *length*) const

Public Function.

Receive a buffer of data from the host machine via a UDP socket connection. The function will block until the length of data provided by the parameter 'length' is received.

Parameters:

retbuf character pointer to the buffer for the received data.

length int length of data to expect from the host machine

Definition at line 260 of file Socket.cpp.

References udp_sock.

Referenced by ServerSocket::rec_buffer_udp().

A.12.3.13 bool Socket::send (const std::string *buffer*) const

Public Function.

Send a string buffer via the TCP socket connection to the destination machine. This requires a TCP socket connection to be already established otherwise an error is returned.

Parameters:

buffer string buffer to send to the host machine.

Definition at line 156 of file Socket.cpp.

References m_sock.

Referenced by ServerSocket::operator<<(), ClientSocket::operator<<(), and send_buffer().

A.12.3.14 bool Socket::send_buffer (const char * *buf*, int *length*) const

Public Function.

Send a character array buffer via the TCP socket connection to the destination machine. This requires a TCP socket connection to be already established otherwise an error is returned. This will send a guarenteed length in bytes to the host machine. If the input buffer exceeds the length then the data will be truncated. If the input buffer is less than the length than the message will be padded.

Parameters:

buf character pointer to the send buffer.

length int length of data to send to the host machine.

Reimplemented in **ClientSocket** (p. 92).

Definition at line 169 of file Socket.cpp.

References m_sock, and send().

Referenced by ClientSocket::send_buffer().

A.12.3.15 bool Socket::send_buffer_udp (char * *buf*, int *length*) const

Public Function.

Send a character array buffer via the UDP socket connection to the destination machine. This requires a UDP socket connection to be already established otherwise an error is returned. This will send a guarenteed length in bytes to the host machine. If the input buffer exceeds the length then the data will be truncated. If the input buffer is less than the length than the message will be padded.

Parameters:

buf character pointer to the send buffer.

length int length of data to send to the host machine.

Reimplemented in **ClientSocket** (p. 93).

Definition at line 182 of file Socket.cpp.

References udp_sock.

Referenced by ClientSocket::send_buffer_udp().

A.12.3.16 void Socket::set_non_blocking (const bool *blocking*)

Public Function.

Sets the TCP socket connection to be a blocking or non-blocking socket.

Parameters:

blocking boolean True sets the connection to be non-blocking, False sets the connection to be a blocking connection

Definition at line 309 of file Socket.cpp.

References m_sock.

A.12.4 Field Documentation

A.12.4.1 sockaddr_in Socket::m_addr [private]

Private variable.

TCP socket port

Definition at line 241 of file Socket.h.

Referenced by accept(), bind(), connect(), and Socket().

A.12.4.2 int Socket::m_sock [private]

Private variable.

TCP socket connection variable

Definition at line 234 of file Socket.h.

Referenced by accept(), bind(), connect(), create(), is_valid(), listen(), recv(), recv_buffer(), send(), send_buffer(), set_non_blocking(), and ∼Socket().

A.12.4.3 sockaddr_in Socket::udp_addr [private]

Private variable.

UDP socket port

Definition at line 256 of file Socket.h.

Referenced by bindUDP(), and Socket().

A.12.4.4 int Socket::udp_sock [private]

Private variable.

UDP socket connection variable

Definition at line 249 of file Socket.h.

Referenced by bindUDP(), create(), createUDP(), is_udp_valid(), recv_buffer_-udp(), send_buffer_udp(), and ∼Socket().

The documentation for this class was generated from the following files:

- **Socket.h**
- **Socket.cpp**

A.13 SocketException Class Reference

Public Class.

`#include <SocketException.h>`

Public Member Functions

- **SocketException** (std::string s)

  Public Constructor.

- ∼**SocketException** ()

  Public DeConstructor.

- std::string **description** ()

  Public Function.

Private Attributes

- std::string **m_s**

  Private Variable.

A.13.1 Detailed Description

Public Class.

Socket(p. 130) Exception Class. This handles exceptions in the socket connection and error reporting for the **Socket**(p. 130) class.

Definition at line 18 of file SocketException.h.

A.13.2 Constructor & Destructor Documentation

A.13.2.1 SocketException::SocketException (std::string *s*) [inline]

Public Constructor.

This is the constructor for the **Socket**(p. 130) Exception class

Definition at line 27 of file SocketException.h.

A.13.2.2 SocketException::~SocketException () [inline]

Public DeConstructor.

This is the deconstructor for the **Socket**(p. 130) Exception class

Definition at line 35 of file SocketException.h.

A.13.3 Member Function Documentation

A.13.3.1 std::string SocketException::description () [inline]

Public Function.

Returns the string description for a socket error number

Definition at line 43 of file SocketException.h.

References m_s.

Referenced by appClient::generateWorkload().

A.13.4 Field Documentation

A.13.4.1 std::string SocketException::m_s [private]

Private Variable.

Stores the current exception string value.

Definition at line 52 of file SocketException.h.

Referenced by description().

The documentation for this class was generated from the following file:

- **SocketException.h**

A.14 TabDialog Class Reference

Public Class.

`#include <tabdialog.h>`

Public Member Functions

- **TabDialog** (QWidget *parent=0)

  Private Constructor Creates an instance of the **TabDialog**(p. 142) *and sets us the appropriate initialization values. This also sets up all of the slot functions and the gui objects for the tab.*

Private Attributes

- QTabWidget * **tabWidget**
- **model** * **my_model**

  Private Variable.

A.14.1 Detailed Description

Public Class.

The **TabDialog**(p. 142) class is a holder for the various tab classes in the GUI.

Definition at line 137 of file tabdialog.h.

A.14.2 Constructor & Destructor Documentation

A.14.2.1 TabDialog::TabDialog (QWidget * *parent* = 0)

Private Constructor Creates an instance of the **TabDialog**(p. 142) and sets us the appropriate initialization values. This also sets up all of the slot functions and the gui objects for the tab.

Parameters:

parent QWidget, Parent widget for the **configureTab**(p. 95), Optional

A.14.3 Field Documentation

A.14.3.1 model* TabDialog::my_model [private]

Private Variable.

The my_model variable is a local instatiation of the Model class and stores the information for the **configureTab**(p. 95).

Definition at line 157 of file tabdialog.h.

A.14.3.2 QTabWidget* TabDialog::tabWidget [private]

Definition at line 151 of file tabdialog.h.

The documentation for this class was generated from the following file:

- **tabdialog.h**

A.15 af_inet.c File Reference

```
#include <linux/timer.h>
#include <linux/proc_fs.h>
#include <linux/random.h>
```

Data Structures

- struct **packetData**

  Private Data Structure.

Defines

- #define **TIMER** 0
- #define **EVENT** 1
- #define **NAMESIZE** 512
- #define **THRESHOLD** 25

Functions

- static void **appQueryData** (unsigned long input)

  Private Function.

- int **appCatchProcRead** (char *buf, char **start, off_t offset, int len, int *unused, void *data)

  Public Function.

- static int **appQueryRead** (char ∗name)

  Public Function.

- int **appCatchProcWrite** (struct file ∗file, const char __user ∗buffer, unsigned long count, void ∗data)

  Public Function.

- static int **appEnvironmentUpdate** (struct sock ∗sk)

Variables

- static struct timer_list **app_timer**
- static int(∗ **app_func**)(char ∗name, int value)
- static struct proc_dir_entry ∗ **app_test_entry**
- static int **appTimerInterval** = 1000
- static int **appTimerRange** = 100
- static int **appQuery** = 0
- static int **appTrigger** = 0
- static char **appQueryName** [NAMESIZE]
- static int **appRegistered** = 0
- static int **debug** = 0
- static int **appState** = 0
- semaphore **app_sem**
- **packetData appData**

A.15.1 Define Documentation

A.15.1.1 #define EVENT 1

EVENT defines the Event Activation Mechanism

Definition at line 153 of file af_inet.c.

Referenced by ip_queue_xmit(), trafficStation(), and udpTrafficStation().

A.15.1.2 #define NAMESIZE 512

NAMESIZE defines the maximim length of a name in a name-value pair

Definition at line 154 of file af_inet.c.

Referenced by appCatchProcWrite(), ipCatchProcWrite(), loCatchProcWrite(), tcpCatchProcWrite(), and udpCatchProcWrite().

A.15.1.3 #define THRESHOLD 25

THRESHOLD defines the change in environment status before a Information Query is initiated with an Event Activation Mechanism

Definition at line 155 of file af_inet.c.

A.15.1.4 #define TIMER 0

TIMER defines the Timer Activation Mechanism

Definition at line 152 of file af_inet.c.

Referenced by appQueryData(), init_module(), ipQueryData(), loQueryData(), tcpQueryData(), and udpQueryData().

A.15.2 Function Documentation

A.15.2.1 int appCatchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *unused*, void * *data*)

Public Function.

This function catches anything trying to read from /proc/brandxapp. This function is called to handle the output of information to user space.

Parameters:

buf character pointer, The buffer where the data is to be inserted

start double character pointer, If you don't want to use the buffer allocated by the kernel

len int, Current position in the file

unused int, Size of the buffer in the first argument

data void pointer, For future use

Definition at line 256 of file af_inet.c.

References appQuery, appRegistered, appTimerInterval, appTimerRange, and appTrigger.

A.15.2.2 int appCatchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*)

Public Function.

This function catches anything trying to write to /proc/brandxapp. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

Definition at line 308 of file af_inet.c.

References app_timer, appQuery, appQueryData(), appQueryName, appQueryRead(), appRegistered, appTimerInterval, appTimerRange, appTrigger, debug, and NAMESIZE.

A.15.2.3 static int appEnvironmentUpdate (struct sock * *sk*) [static]

/brief Public Function

This function updates the internal cache stored in the local protocol layer. This is utilized in decreasing response time with a Synchronized Pull architecture.

Definition at line 480 of file af_inet.c.

References appData, debug, packetData::dropped, packetData::packets, packetData::rxBytes, packetData::rxErrors, packetData::txBytes, and packetData::txErrors.

A.15.2.4 static void appQueryData (unsigned long *input*) [static]

Private Function.

Timer activated information update mechanism. This function is utilized during information updates based on a Timer activation mechanism. Special care must be taken to remove the registered tcp_timer value before this function is removed or is will cause a segmenation fault.

Definition at line 200 of file af_inet.c.

References app_func, app_sem, app_timer, appQuery, appQueryName, appTimerInterval, appTrigger, debug, and TIMER.

Referenced by appCatchProcWrite().

A.15.2.5 static int appQueryRead (char * *name*) [static]

Public Function.

Intermodule Communication function, calls the registered callback function with updated environment information. This is utilized during the Synchronous Pull architecture.

Definition at line 277 of file af_inet.c.

References app_func, appQueryName, and debug.

Referenced by appCatchProcWrite().

A.15.3 Variable Documentation

A.15.3.1 int(* app_func)(char *name, int value) [static]

Stores the pointer to the function launched with the timer object

Definition at line 158 of file af_inet.c.

Referenced by appQueryData(), and appQueryRead().

A.15.3.2 struct semaphore app_sem

Stores the semaphore for the local layer race condition control

Definition at line 168 of file af_inet.c.

Referenced by appQueryData().

A.15.3.3 struct proc_dir_entry* app_test_entry [static]

Stores the entry for the proc file system in the Linux Kernel

Definition at line 159 of file af_inet.c.

A.15.3.4 struct timer_list app_timer [static]

Stores the Linux kernel timer object utilized with a Timer Activation Mechanism

Definition at line 157 of file af_inet.c.

Referenced by appCatchProcWrite(), and appQueryData().

A.15.3.5 struct packetData appData

Referenced by appEnvironmentUpdate(), readFromBillboard(), and writeOnBillboard().

A.15.3.6 int appQuery = 0 [static]

Stores if the protocol layer is actively performing information queries

Definition at line 162 of file af_inet.c.

Referenced by appCatchProcRead(), appCatchProcWrite(), and appQueryData().

A.15.3.7 char appQueryName[NAMESIZE] [static]

Stores the names of the different architectures utilized in the experiments

Definition at line 164 of file af_inet.c.

Referenced by appCatchProcWrite(), appQueryData(), and appQueryRead().

A.15.3.8 int appRegistered = 0 [static]

Stores if the protocol layer IMC functions have been registered with the Linux Kernel

Definition at line 165 of file af_inet.c.

Referenced by appCatchProcRead(), and appCatchProcWrite().

A.15.3.9 int appState = 0 [static]

Stores the current state of the information query

Definition at line 167 of file af_inet.c.

A.15.3.10 int appTimerInterval = 1000 [static]

Defines the timer interval for Timer driven information querys

Definition at line 160 of file af_inet.c.

Referenced by appCatchProcRead(), appCatchProcWrite(), and appQueryData().

A.15.3.11 int appTimerRange = 100 [static]

Defines the timer range for Event driven information querys

Definition at line 161 of file af_inet.c.

Referenced by appCatchProcRead(), and appCatchProcWrite().

A.15.3.12 int appTrigger = 0 [static]

Defines the activation mechanims utilized in the current experiment

Definition at line 163 of file af_inet.c.

Referenced by appCatchProcRead(), appCatchProcWrite(), and appQueryData().

A.15.3.13 int debug = 0 [static]

Stores the current DEBUG level

Definition at line 166 of file af_inet.c.

A.16 appClient.cpp File Reference

```
#include "appClient.h"
```

Functions

- void **writeToFile** (char *fileName, char *data)
  Private Function.
- int **brandxStart** (int timer, int range, int trigger, int delay)
  Private Function.
- int **brandxStop** ()
  Private Function.
- int **controlStart** (int timer, int range, int trigger, int delay)
  Private Function.
- int **controlStop** ()
  Private Function.
- int **pushStart** (int timer, int range, int trigger, int delay)
  Private Function.
- int **pushStop** ()
  Private Function.
- int **pullStart** (int timer, int range, int trigger, int delay)
  Private Function.
- int **pullStop** ()
  Private Function.
- int **main** (int argc, int argv[ ])
  Main Function.

A.16.1 Function Documentation

A.16.1.1 int brandxStart (int *timer*, int *range*, int *trigger*, int *delay*)

Private Function.

This function starts the Brand X cross-layer architecture with the specified configuration. The input parameters are used to control the Brand X attributes and create various test scenerios.

Parameters:

timer int Query interaval time

range int Event range variable

trigger int Timer or Event activation mechanism

delay int QoT and Physical layer delay times

Definition at line 425 of file appClient.cpp.

References BRANDX, and writeToFile().

Referenced by main().

A.16.1.2 int brandxStop ()

Private Function.

This function stops the Brand X cross-layer architecture and clear any structures or memory that has been modified in the network protocol stack.

Definition at line 498 of file appClient.cpp.

References writeToFile().

Referenced by main().

A.16.1.3 int controlStart (int *timer*, int *range*, int *trigger*, int *delay*)

Private Function.

This function starts the Control cross-layer architecture with the specified configuration. The input parameters are used to control the Control attributes and create various test scenerios.

Parameters:

timer int Query interaval time

range int Event range variable

trigger int Timer or Event activation mechanism

delay int QoT and Physical layer delay times

Definition at line 539 of file appClient.cpp.

References CONTROL, and writeToFile().

A.16.1.4 int controlStop ()

Private Function.

This function stops the Control cross-layer architecture and clear any structures or memory that has been modified in the network protocol stack.

Definition at line 579 of file appClient.cpp.

References writeToFile().

A.16.1.5 int main (int *argc*, int *argv*[ ])

Main Function.

This function starts the Application Client and work load generation. User input at the command line or scripted values are used to control this function and launch automated testing. This function can be linked with the gui to provide interface support or it can be run in a scripted stand-alone mode.

Definition at line 810 of file appClient.cpp.

References brandxStart(), brandxStop(), appClient::generateWorkload(), names, appClient::setAvgFile(), appClient::setBurstSize(), appClient::setLogFile(), appClient::setPacketInterval(), and appClient::setPacketSize().

A.16.1.6 int pullStart (int *timer*, int *range*, int *trigger*, int *delay*)

Private Function.

This function starts the Synchronous Pull cross-layer architecture with the specified configuration. The input parameters are used to control the Synchronous Pull attributes and create various test scenerios.

Parameters:

timer int Query interaval time

range int Event range variable

trigger int Timer or Event activation mechanism

delay int QoT and Physical layer delay times

Definition at line 707 of file appClient.cpp.

References writeToFile().

A.16.1.7 int pullStop ()

Private Function.

This function stops the Synchronous Pull cross-layer architecture and clear any structures or memory that has been modified in the network protocol stack.

Definition at line 783 of file appClient.cpp.

References writeToFile().

A.16.1.8 int pushStart (int *timer*, int *range*, int *trigger*, int *delay*)

Private Function.

This function starts the Synchronous Push cross-layer architecture with the specified configuration. The input parameters are used to control the Synchronous Push attributes and create various test scenerios.

Parameters:

timer int Query interaval time

range int Event range variable

trigger int Timer or Event activation mechanism

delay int QoT and Physical layer delay times

Definition at line 604 of file appClient.cpp.

References PUSH, and writeToFile().

A.16.1.9 int pushStop ()

Private Function.

This function stops the Synchronous Push cross-layer architecture and clear any structures or memory that has been modified in the network protocol stack.

Definition at line 674 of file appClient.cpp.

References writeToFile().

A.16.1.10 void writeToFile (char * *fileName*, char * *data*)

Private Function.

This function writes the contents of the data parameter to the specified fileName. This mechanisms communicates via the proc file system to go between the User level Linux environment to Kernel level processes.

Parameters:

fileName character pointer holds the proc filename for the file write

data character pointer holds the data to be written to the file specified in fileName

Definition at line 404 of file appClient.cpp.

Referenced by brandxStart(), brandxStop(), controlStart(), controlStop(), pullStart(), pullStop(), pushStart(), and pushStop().

A.17 appClient.h File Reference

```
#include "ClientSocket.h"
#include "ServerSocket.h"
#include "SocketException.h"
#include <iostream>
#include <string>
#include <sys/time.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <fstream>
```

Namespaces

- namespace **std**

Data Structures

- class **appClient**

  Traffic generation class.

- struct **appClient::experiment**

  Private experiment structure.

Defines

- #define **MAX_SIZE** 66000

  Maximum size for workload traffic.

- #define **STDOUT** 0

  Defined value for logging data to stdout (debug).

- #define **FILE** 1

  Defined value for logging data to a file.

- #define **INIT_SIZE** 128

Initial size of workload traffic.

- #define **CONTROL** 0
- #define **PUSH** 1
- #define **PULL** 2
- #define **BRANDX** 3

Variables

- static char **names** [ ][16] = {{"control"},{"push"},{"pull"},{"brandx"}}

A.17.1 Define Documentation

A.17.1.1 #define BRANDX 3

Definition at line 29 of file appClient.h.

Referenced by brandxStart(), and qotTimerQuery().

A.17.1.2 #define CONTROL 0

Definition at line 26 of file appClient.h.

Referenced by controlStart().

A.17.1.3 #define FILE 1

Defined value for logging data to a file.

Definition at line 23 of file appClient.h.

Referenced by appServer::log(), appClient::log(), and procTab::sendProcCmd().

A.17.1.4 #define INIT_SIZE 128

Initial size of workload traffic.

Definition at line 25 of file appClient.h.

Referenced by appClient::generateWorkload(), appServer::recievePackets(), and appClient::socketConfiguration().

A.17.1.5 #define MAX_SIZE 66000

Maximum size for workload traffic.

Definition at line 19 of file appClient.h.

Referenced by appClient::generateWorkload(), appServer::recievePackets(), and appClient::socketConfiguration().

A.17.1.6 #define PULL 2

Definition at line 28 of file appClient.h.

A.17.1.7 #define PUSH 1

Definition at line 27 of file appClient.h.

Referenced by pushStart().

A.17.1.8 #define STDOUT 0

Defined value for logging data to stdout (debug).

Definition at line 21 of file appClient.h.

Referenced by appServer::log(), appClient::log(), appServer::recievePackets(), and appClient::socketConfiguration().

A.17.2 Variable Documentation

A.17.2.1 char names[ ][16] = {{"control"},{"push"},{"pull"},{"brandx"}} [static]

Definition at line 30 of file appClient.h.

Referenced by main().

A.18 appServer.cpp File Reference

```
#include "appServer.h"
```

Functions

- int **main** (int argc, int argv[ ])

  Main Function.

A.18.1 Function Documentation

A.18.1.1 int main (int *argc*, int *argv*[ ])

Main Function.

This function starts the Application Server and work load reception.

Definition at line 280 of file appServer.cpp.

References appServer::recievePackets().

A.19 appServer.h File Reference

```
#include <iostream>
#include "ServerSocket.h"
#include "SocketException.h"
#include <string>
#include <sys/time.h>
#include <fstream>
#include <unistd.h>
```

Data Structures

- class **appServer**

  Traffic reception class.

- struct **appServer::packetInfo**

  Private Data Structure.

Defines

- #define **MAX_SIZE** 66000

  Maximum size for workload traffic.

- #define **STDOUT** 0

  Defined value for logging data to stdout (debug).

- #define **FILE** 1

  Defined value for logging data to a file.

- #define **INIT_SIZE** 128

  Initial size of workload traffic.

A.19.1 Define Documentation

A.19.1.1 #define FILE 1

Defined value for logging data to a file.

Definition at line 17 of file appServer.h.

A.19.1.2 #define INIT_SIZE 128

Initial size of workload traffic.

Definition at line 19 of file appServer.h.

A.19.1.3 #define MAX_SIZE 66000

Maximum size for workload traffic.

Definition at line 13 of file appServer.h.

A.19.1.4 #define STDOUT 0

Defined value for logging data to stdout (debug).

Definition at line 15 of file appServer.h.

A.20 brandx.c File Reference

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
```

Data Structures

- struct **name_value**

  Internal data storage structure.

Defines

- #define **SYSLOG** 0
- #define **FILELOG** 1
- #define **MAXPROC** 1024

Functions

- void **logData** (char *data, int type)

  Public Function Log data out to relevant areas.

- void **writeOnBillboard** (const char *name, int value)

  Public Function Write a name-value pair to the billboard for latter use by QoT.

- int **readFromBillboard** (char *name)

  Public Function Read a value from the billboard. The name value pair must already exist on the billboard or an error is returned.

- static int **catchProcRead** (char *buf, char **start, off_t offset, int len, int *eof, void *type)

  Public Function Read data from the Brand X proc file, brandx. This is used for Kernel to User level communication and configuration. This is utilized when a user level process attempts to read the brandx proc file.

- int **catchProcWrite** (struct file *file, const char __user *buffer, unsigned long count, void *data)

  Public Function Write data to the Brand X proc file, brandx. This is utilized when a user level process writes to the Brand X proc file. Utilizing this method user level processes can communicate with kernel level processes.

- void **regProcFunctions** ()

  Public Function Register the proc filesystem functions with the Linux kernel.

- void **unregProcFunctions** ()

  Public Function Unregister the proc filesystem functions with the Linux kernel.

- void **billboard_write** (char ∗name, int value)

  Public Function Write a name-value pair to the billboard. This function is called via the Inter process communication mechanism available through the Linux Kernel.

- int **billboard_read** (char ∗name)

  Public Function Read a name-value pair from the billboard. This function is called via the Inter process communication mechanism available through the Linux Kernel.

- void **regCallbackFunctions** ()

  Public Function Register the cross intermodule callback functions, billboard_write and billboard_read.

- void **unregCallbackFunctions** ()

  Public Function Unregister the cross intermodule callback functions, billboard_write and billboard_read.

- int **init_module** ()

  Module Initialization Function.

- void **cleanup_module** ()

  Module Destroy Function.

- **MODULE_LICENSE** ("GPL")

A.20.1 Define Documentation

A.20.1.1 #define FILELOG 1

Definition at line 18 of file brandx.c.

Referenced by logData().

A.20.1.2 #define MAXPROC 1024

Definition at line 19 of file brandx.c.

Referenced by qotCatchProcWrite().

A.20.1.3 #define SYSLOG 0

Need to: 1.Add a mechanism to erase from the billboard 2.Change hardcoded data and config value to dynamic list 3.Add the ability to query for more than one value at a time

Definition at line 17 of file brandx.c.

Referenced by catchProcRead(), logData(), readFromBillboard(), regCallback-Functions(), regProcFunctions(), unregCallbackFunctions(), unregProcFunctions(), and writeOnBillboard().

A.20.2 Function Documentation

A.20.2.1 int billboard_read (char * *name*)

Public Function Read a name-value pair from the billboard. This function is called via the Inter process communication mechanism available through the Linux Kernel.

Parameters:

name character pointer holds the name of the pair to be read from the billboard.

Definition at line 302 of file brandx.c.

Referenced by regCallbackFunctions().

A.20.2.2 void billboard_write (char * *name*, int *value*)

Public Function Write a name-value pair to the billboard. This function is called via the Inter process communication mechanism available through the Linux Kernel.

Parameters:

name character pointer holds the name of the pair to be written to the billboard.

value int holds the data to be writte to the billboard.

Definition at line 291 of file brandx.c.

References debug.

Referenced by regCallbackFunctions().

A.20.2.3 static int catchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *eof*, void * *type*) [static]

Public Function Read data from the Brand X proc file, brandx. This is used for Kernel to User level communication and configuration. This is utilized when a user level process attempts to read the brandx proc file.

Parameters:

buf character pointer holds the buffer of the user level data

start character double pointer holds the start position to write data to the buffer

len int indicates the length of data to write to the input buffer

type void pointer not used

Definition at line 170 of file brandx.c.

References logData(), readFromBillboard(), and SYSLOG.

Referenced by regProcFunctions().

A.20.2.4 int catchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*)

Public Function Write data to the Brand X proc file, brandx. This is utilized when a user level process writes to the Brand X proc file. Utilizing this method user level processes can communicate with kernel level processes.

Parameters:

file file structure holds a pointer to the file that is written to.

buffer character pointer is the user level data, this must be copied because it is user level memory.

count unsigned long holds the length of the data that was written

data void pointer not used

Definition at line 221 of file brandx.c.

References debug.

Referenced by init_module(), and regProcFunctions().

A.20.2.5 void cleanup_module (void)

Module Destroy Function.

Called when the module is unloaded into the Linux Kernel. All functions are implemented as pluggable kernel modules. This function unregisters the inter-process communication functions and removes the proc file system entries.

Definition at line 348 of file brandx.c.

A.20.2.6 int init_module (void)

Module Initialization Function.

Called when the module is loaded into the Linux Kernel. All functions are implemented as pluggable kernel modules. This function registers the inter-process communication functions and the proc file system entries.

Definition at line 334 of file brandx.c.

A.20.2.7 void logData (char * *data*, int *type*)

Public Function Log data out to relevant areas.

Parameters:

data character pointer to message buffer to be logged.

type specifies the data logging system; 0=syslog, 1=file.

Definition at line 61 of file brandx.c.

References FILELOG, and SYSLOG.

Referenced by catchProcRead(), readFromBillboard(), regCallbackFunctions(), regProcFunctions(), unregCallbackFunctions(), unregProcFunctions(), and writeOnBillboard().

A.20.2.8 MODULE_LICENSE ("GPL")

A.20.2.9 int readFromBillboard (char * *name*)

Public Function Read a value from the billboard. The name value pair must already exist on the billboard or an error is returned.

Parameters:

name character pointer holds the name of the name-value requested.

Definition at line 118 of file brandx.c.

References appData, debug, ipData, logData(), SYSLOG, tcpData, and udpData.

Referenced by catchProcRead().

A.20.2.10 void regCallbackFunctions (void)

Public Function Register the cross intermodule callback functions, billboard_write and billboard_read.

Definition at line 311 of file brandx.c.

Referenced by init_module().

A.20.2.11 void regProcFunctions ()

Public Function Register the proc filesystem functions with the Linux kernel.

Definition at line 268 of file brandx.c.

References catchProcRead(), catchProcWrite(), logData(), and SYSLOG.

Referenced by init_module().

A.20.2.12 void unregCallbackFunctions (void)

Public Function Unregister the cross intermodule callback functions, billboard_-write and billboard_read.

Definition at line 320 of file brandx.c.

Referenced by cleanup_module().

A.20.2.13 void unregProcFunctions ()

Public Function Unregister the proc filesystem functions with the Linux kernel.

Definition at line 278 of file brandx.c.

References logData(), and SYSLOG.

Referenced by cleanup_module().

A.20.2.14 void writeOnBillboard (const char * *name*, int *value*)

Public Function Write a name-value pair to the billboard for latter use by QoT.

Parameters:

name character pointer holds the name of the source protocol layer

value int holds the value written to the billboard

Definition at line 76 of file brandx.c.

References appData, debug, ipData, logData(), SYSLOG, tcpData, and udpData.

A.21 ClientSocket.cpp File Reference

```
#include "ClientSocket.h"
#include "SocketException.h"
```

A.22 ClientSocket.h File Reference

```
#include "Socket.h"
```

Data Structures

- class **ClientSocket**

  Public Class.

A.23 ip_output.c File Reference

```
#include <asm/uaccess.h>
#include <asm/system.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/config.h>
#include <linux/socket.h>
#include <linux/sockios.h>
#include <linux/in.h>
#include <linux/inet.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/proc_fs.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/timer.h>
#include <net/snmp.h>
#include <net/ip.h>
#include <net/protocol.h>
#include <net/route.h>
#include <net/tcp.h>
#include <net/udp.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <net/arp.h>
#include <net/icmp.h>
#include <net/raw.h>
```

```
#include <net/checksum.h>
#include <net/inetpeer.h>
#include <linux/igmp.h>
#include <linux/netfilter_ipv4.h>
#include <linux/netfilter_bridge.h>
#include <linux/mroute.h>
#include <linux/netlink.h>
```

Data Structures

- struct **packetData**

  Private Data Structure.

Defines

- #define **TIMER** 0
- #define **EVENT** 1
- #define **NAMESIZE** 512
- #define **THRESHOLD** 25

Functions

- __inline__ void **ip_send_check** (struct iphdr ∗iph)
- static void **ipQueryData** (unsigned long input)

  Private Function.

- int **ipCatchProcRead** (char ∗buf, char ∗∗start, off_t offset, int len, int ∗unused, void ∗data)

  Public Function.

- int **ipQueryRead** (char ∗name)

  Public Function.

- int **ipCatchProcWrite** (struct file ∗file, const char __user ∗buffer, unsigned long count, void ∗data)

  Public Function.

- static int **ipEnvironmentUpdate** (struct sk_buff ∗skb)

- static int **ip_dev_loopback_xmit** (struct sk_buff ∗newskb)
- static int **ip_select_ttl** (struct inet_opt ∗inet, struct dst_entry ∗dst)
- int **ip_build_and_send_pkt** (struct sk_buff ∗skb, struct sock ∗sk, u32 saddr, u32 daddr, struct ip_options ∗opt)
- static int **ip_finish_output2** (struct sk_buff ∗skb)
- int **ip_finish_output** (struct sk_buff ∗skb)
- int **ip_mc_output** (struct sk_buff ∗∗pskb)
- static int **ip_output2** (struct sk_buff ∗skb)
- int **ip_output** (struct sk_buff ∗∗pskb)
- int **ip_queue_xmit** (struct sk_buff ∗skb, int ipfragok)
- static void **ip_copy_metadata** (struct sk_buff ∗to, struct sk_buff ∗from)
- int **ip_fragment** (struct sk_buff ∗skb, int(∗output)(struct sk_buff ∗))
- int **ip_generic_getfrag** (void ∗from, char ∗to, int offset, int len, int odd, struct sk_buff ∗skb)
- static unsigned int **csum_page** (struct page ∗page, int offset, int copy)
- int **ip_append_data** (struct sock ∗sk, int getfrag(void ∗from, char ∗to, int offset, int len, int odd, struct sk_buff ∗skb), void ∗from, int length, int transhdrlen, struct ipcm_cookie ∗ipc, struct rtable ∗rt, unsigned int flags)
- ssize_t **ip_append_page** (struct sock ∗sk, struct page ∗page, int offset, size_t size, int flags)
- int **ip_push_pending_frames** (struct sock ∗sk)
- void **ip_flush_pending_frames** (struct sock ∗sk)
- static int **ip_reply_glue_bits** (void ∗dptr, char ∗to, int offset, int len, int odd, struct sk_buff ∗skb)
- void **ip_send_reply** (struct sock ∗sk, struct sk_buff ∗skb, struct ip_reply_arg ∗arg, unsigned int len)
- void __init **ip_init** (void)
- **EXPORT_SYMBOL** (ip_finish_output)
- **EXPORT_SYMBOL** (ip_fragment)
- **EXPORT_SYMBOL** (ip_generic_getfrag)
- **EXPORT_SYMBOL** (ip_queue_xmit)
- **EXPORT_SYMBOL** (ip_send_check)

Variables

- int **sysctl_ip_dynaddr**
- int **sysctl_ip_default_ttl** = IPDEFTTL
- static struct timer_list **ip_timer**
- static int(∗ **ip_func**)(char ∗name, int value)
- static struct proc_dir_entry ∗ **ip_test_entry**
- static int **ipTimerInterval** = 1000

- static int **ipTimerRange** = 100
- static int **ipQuery** = 0
- static int **ipTrigger** = 0
- static char **ipQueryName** [NAMESIZE]
- static int **ipRegistered** = 0
- static int **debug** = 0
- static int **ipState** = 0
- semaphore **ip_sem**
- **packetData ipData**
- static struct packet_type **ip_packet_type**

A.23.1 Define Documentation

A.23.1.1 #define EVENT 1

EVENT defines the Event Activation Mechanism

Definition at line 113 of file ip_output.c.

A.23.1.2 #define NAMESIZE 512

NAMESIZE defines the maximim length of a name in a name-value pair

Definition at line 114 of file ip_output.c.

A.23.1.3 #define THRESHOLD 25

THRESHOLD defines the change in environment status before a Information Query is initiated with an Event Activation Mechanism

Definition at line 115 of file ip_output.c.

A.23.1.4 #define TIMER 0

TIMER defines the Timer Activation Mechanism

Definition at line 112 of file ip_output.c.

A.23.2 Function Documentation

A.23.2.1 static unsigned int csum_page (struct page * *page*, int *offset*, int *copy*) [inline, static]

Definition at line 1137 of file ip_output.c.

Referenced by ip_append_page().

A.23.2.2 EXPORT_SYMBOL (ip_send_check)

A.23.2.3 EXPORT_SYMBOL (ip_queue_xmit)

A.23.2.4 EXPORT_SYMBOL (ip_generic_getfrag)

A.23.2.5 EXPORT_SYMBOL (ip_fragment)

A.23.2.6 EXPORT_SYMBOL (ip_finish_output)

A.23.2.7 int ip_append_data (struct sock * *sk*, int *getfrag*(void *from, char *to, int offset, int len,int odd, struct sk_buff *skb), void * *from*, int *length*, int *transhdrlen*, struct ipcm_cookie * *ipc*, struct rtable * *rt*, unsigned int *flags*)

Definition at line 1158 of file ip_output.c.

Referenced by ip_send_reply().

A.23.2.8 ssize_t ip_append_page (struct sock * *sk*, struct page * *page*, int *offset*, size_t *size*, int *flags*)

Definition at line 1423 of file ip_output.c.

References csum_page().

A.23.2.9 int ip_build_and_send_pkt (struct sk_buff * *skb*, struct sock * *sk*, u32 *saddr*, u32 *daddr*, struct ip_options * *opt*)

Definition at line 512 of file ip_output.c.

References ip_select_ttl(), and ip_send_check().

A.23.2.10 static void ip_copy_metadata (struct sk_buff * *to*, struct sk_buff * *from*) [static]

Definition at line 832 of file ip_output.c.

Referenced by ip_fragment().

A.23.2.11 static int ip_dev_loopback_xmit (struct sk_buff * *newskb*) [static]

Definition at line 471 of file ip_output.c.

Referenced by ip_mc_output().

A.23.2.12 int ip_finish_output (struct sk_buff * *skb*)

Definition at line 597 of file ip_output.c.

References ip_finish_output2().

Referenced by ip_mc_output(), and ip_output2().

A.23.2.13 static int ip_finish_output2 (struct sk_buff * *skb*) [inline, static]

Definition at line 553 of file ip_output.c.

Referenced by ip_finish_output().

A.23.2.14 void ip_flush_pending_frames (struct sock * *sk*)

Definition at line 1668 of file ip_output.c.

A.23.2.15 int ip_fragment (struct sk_buff * *skb*, int(*)(struct sk_buff *) *output*)

Definition at line 872 of file ip_output.c.

References ip_copy_metadata(), and ip_send_check().

Referenced by ip_mc_output(), and ip_output2().

A.23.2.16 int ip_generic_getfrag (void * *from*, char * *to*, int *offset*, int *len*, int *odd*, struct sk_buff * *skb*)

Definition at line 1120 of file ip_output.c.

A.23.2.17 void __init ip_init (void)

register the function for the proc FS

Definition at line 1786 of file ip_output.c.

References ip_sem, ip_test_entry, ipCatchProcRead(), and ipCatchProcWrite().

A.23.2.18 int ip_mc_output (struct sk_buff ** *pskb*)

Definition at line 608 of file ip_output.c.

References ip_dev_loopback_xmit(), ip_finish_output(), and ip_fragment().

A.23.2.19 int ip_output (struct sk_buff ∗∗ *pskb*)

Definition at line 680 of file ip_output.c.

References debug, ip_output2(), and ip_sem.

A.23.2.20 static int ip_output2 (struct sk_buff ∗ *skb*) [inline, static]

Definition at line 669 of file ip_output.c.

References ip_finish_output(), and ip_fragment().

Referenced by ip_output().

A.23.2.21 int ip_push_pending_frames (struct sock ∗ *sk*)

Definition at line 1561 of file ip_output.c.

References ip_select_ttl(), and ip_send_check().

Referenced by ip_send_reply().

A.23.2.22 int ip_queue_xmit (struct sk_buff ∗ *skb*, int *ipfragok*)

Definition at line 704 of file ip_output.c.

References debug, EVENT, ip_select_ttl(), ip_send_check(), ipEnvironment-Update(), ipQuery, ipQueryData(), ipState, ipTimerRange, and ipTrigger.

A.23.2.23 static int ip_reply_glue_bits (void ∗ *dptr*, char ∗ *to*, int *offset*, int *len*, int *odd*, struct sk_buff ∗ *skb*) [static]

Definition at line 1691 of file ip_output.c.

Referenced by ip_send_reply().

A.23.2.24 static int ip_select_ttl (struct inet_opt ∗ *inet*, struct dst_entry ∗ *dst*) [inline, static]

Definition at line 490 of file ip_output.c.

Referenced by ip_build_and_send_pkt(), ip_push_pending_frames(), and ip_-queue_xmit().

A.23.2.25 __inline__ void ip_send_check (struct iphdr ∗ *iph*)

Definition at line 99 of file ip_output.c.

Referenced by ip_build_and_send_pkt(), ip_fragment(), ip_push_pending_-frames(), and ip_queue_xmit().

A.23.2.26 void ip_send_reply (struct sock * *sk*, struct sk_buff * *skb*, struct ip_reply_arg * *arg*, unsigned int *len*)

Definition at line 1710 of file ip_output.c.

References ip_append_data(), ip_push_pending_frames(), and ip_reply_glue_-bits().

A.23.2.27 int ipCatchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *unused*, void * *data*)

Public Function.

This function catches anything trying to read from /proc/brandxip. This function is called to handle the output of information to user space.

Parameters:

buf character pointer, The buffer where the data is to be inserted

start double character pointer, If you don't want to use the buffer allocated by the kernel

len int, Current position in the file

unused int, Size of the buffer in the first argument

data void pointer, For future use

Definition at line 216 of file ip_output.c.

References ipQuery, ipRegistered, ipTimerInterval, ipTimerRange, and ipTrigger.

Referenced by ip_init().

A.23.2.28 int ipCatchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*)

Public Function.

This function catches anything trying to write to /proc/brandxtcp. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

Definition at line 268 of file ip_output.c.

References debug, ip_timer, ipQuery, ipQueryData(), ipQueryName, ipQueryRead(), ipRegistered, ipTimerInterval, ipTimerRange, ipTrigger, and NAMESIZE.

Referenced by ip_init().

A.23.2.29 static int ipEnvironmentUpdate (struct sk_buff * *skb*) [static]

/brief Public Function

This function updates the internal cache stored in the local protocol layer. This is utilized in decreasing response time with a Synchronized Pull architecture.

Definition at line 440 of file ip_output.c.

References debug, packetData::dropped, ipData, packetData::packets, packetData::rxBytes, packetData::rxErrors, packetData::txBytes, and packetData::txErrors.

Referenced by ip_queue_xmit().

A.23.2.30 static void ipQueryData (unsigned long *input*) [static]

Private Function.

Timer activated information update mechanism. This function is utilized during information updates based on a Timer activation mechanism. Special care must be taken to remove the registered tcp_timer value before this function is removed or is will cause a segmenation fault.

Make a call to brand X to retrieve data

Definition at line 160 of file ip_output.c.

References debug, ip_func, ip_sem, ip_timer, ipQuery, ipQueryName, ipTimerInterval, ipTrigger, and TIMER.

Referenced by ip_queue_xmit(), and ipCatchProcWrite().

A.23.2.31 int ipQueryRead (char * *name*)

Public Function.

Intermodule Communication function, calls the registered callback function with updated environment information. This is utilized during the Synchronous Pull architecture.

Make a call to brand X to retrieve data

Definition at line 237 of file ip_output.c.

References debug, ip_func, and ipQueryName.

Referenced by ipCatchProcWrite().

A.23.3 Variable Documentation

A.23.3.1 int debug = 0 [static]

Stores the current DEBUG level

Definition at line 126 of file ip_output.c.

A.23.3.2 int(* ip_func)(char *name, int value) [static]

Stores the pointer to the function launched with the timer object

Definition at line 118 of file ip_output.c.

Referenced by ipQueryData(), and ipQueryRead().

A.23.3.3 struct packet_type ip_packet_type [static]

Initial value:

```
 {
        .type = __constant_htons(ETH_P_IP),
        .func = ip_rcv,
}
```

Definition at line 1777 of file ip_output.c.

A.23.3.4 struct semaphore ip_sem

Stores the semaphore for the local layer race condition control

Definition at line 128 of file ip_output.c.

Referenced by ip_init(), ip_output(), and ipQueryData().

A.23.3.5 struct proc_dir_entry* ip_test_entry [static]

Stores the entry for the proc file system in the Linux Kernel

Definition at line 119 of file ip_output.c.

Referenced by ip_init().

A.23.3.6 struct timer_list ip_timer [static]

Stores the Linux kernel timer object utilized with a Timer Activation Mechanism

Definition at line 117 of file ip_output.c.

Referenced by ipCatchProcWrite(), and ipQueryData().

A.23.3.7 struct packetData ipData

Referenced by ipEnvironmentUpdate(), readFromBillboard(), and writeOnBillboard().

A.23.3.8 int ipQuery = 0 [static]

Stores if the protocol layer is actively performing information queries

Definition at line 122 of file ip_output.c.

Referenced by ip_queue_xmit(), ipCatchProcRead(), ipCatchProcWrite(), and ipQueryData().

A.23.3.9 char ipQueryName[NAMESIZE] [static]

Stores the names of the different architectures utilized in the experiments

Definition at line 124 of file ip_output.c.

Referenced by ipCatchProcWrite(), ipQueryData(), and ipQueryRead().

A.23.3.10 int ipRegistered = 0 [static]

Stores if the protocol layer IMC functions have been registered with the Linux Kernel

Definition at line 125 of file ip_output.c.

Referenced by ipCatchProcRead(), and ipCatchProcWrite().

A.23.3.11 int ipState = 0 [static]

Stores the current state of the information query

Definition at line 127 of file ip_output.c.

Referenced by ip_queue_xmit().

A.23.3.12 int ipTimerInterval = 1000 [static]

Defines the timer interval for Timer driven information querys

Definition at line 120 of file ip_output.c.

Referenced by ipCatchProcRead(), ipCatchProcWrite(), and ipQueryData().

A.23.3.13 int ipTimerRange = 100 [static]

Defines the timer range for Event driven information querys

Definition at line 121 of file ip_output.c.

Referenced by ip_queue_xmit(), ipCatchProcRead(), and ipCatchProcWrite().

A.23.3.14 int ipTrigger = 0 [static]

Defines the activation mechanims utilized in the current experiment

Definition at line 123 of file ip_output.c.

Referenced by ip_queue_xmit(), ipCatchProcRead(), ipCatchProcWrite(), and ipQueryData().

A.23.3.15 int sysctl_ip_default_ttl = IPDEFTTL

Definition at line 96 of file ip_output.c.

A.23.3.16 int sysctl_ip_dynaddr

Definition at line 95 of file ip_output.c.

A.24 loopback.c File Reference

```
#include <linux/timer.h>
#include <linux/proc_fs.h>
```

Data Structures

- struct **packetData**

  Private Data Structure.

Defines

- #define **TIMER** 0
- #define **EVENT** 1
- #define **NAMESIZE** 512
- #define **THRESHOLD** 25

Functions

- static void **loQueryData** (unsigned long input)

  Private Function.

- int **loCatchProcRead** (char ∗buf, char ∗∗start, off_t offset, int len, int ∗unused, void ∗data)

  Public Function.

- static int **linkQueryRead** (char ∗name)

  Public Function.

- int **loCatchProcWrite** (struct file ∗file, const char __user ∗buffer, unsigned long count, void ∗data)

  Public Function.

- static int **loEnvironmentUpdate** (struct net_device_stats ∗lb_stats)

Variables

- static struct timer_list **lo_timer**
- static int(∗ **lo_func**)(char ∗name, int value)
- static struct proc_dir_entry ∗ **lo_test_entry**

- static int **loTimerInterval** = 1000
- static int **loTimerRange** = 100
- static int **loQuery** = 0
- static int **loTrigger** = 0
- static char **loQueryName** [NAMESIZE]
- static int **loRegistered** = 0
- static int **debug** = 0
- static int **loState** = 0
- semaphore **lo_sem**
- **packetData loData**

A.24.1 Define Documentation

A.24.1.1 #define EVENT 1

EVENT defines the Event Activation Mechanism

Definition at line 82 of file loopback.c.

A.24.1.2 #define NAMESIZE 512

NAMESIZE defines the maximim length of a name in a name-value pair

Definition at line 83 of file loopback.c.

A.24.1.3 #define THRESHOLD 25

THRESHOLD defines the change in environment status before a Information Query is initiated with an Event Activation Mechanism

Definition at line 84 of file loopback.c.

A.24.1.4 #define TIMER 0

TIMER defines the Timer Activation Mechanism

Definition at line 81 of file loopback.c.

A.24.2 Function Documentation

A.24.2.1 static int linkQueryRead (char * *name*) [static]

Public Function.

Intermodule Communication function, calls the registered callback function with updated environment information. This is utilized during the Synchronous Pull architecture.

Definition at line 206 of file loopback.c.

References debug, lo_func, and loQueryName.

Referenced by loCatchProcWrite(), and udpCatchProcWrite().

A.24.2.2 int loCatchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *unused*, void * *data*)

Public Function.

This function catches anything trying to read from /proc/brandxlo. This function is called to handle the output of information to user space.

Parameters:

buf character pointer, The buffer where the data is to be inserted

start double character pointer, If you don't want to use the buffer allocated by the kernel

len int, Current position in the file

unused int, Size of the buffer in the first argument

data void pointer, For future use

Definition at line 185 of file loopback.c.

References loQuery, loRegistered, loTimerInterval, loTimerRange, and loTrigger.

A.24.2.3 int loCatchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*)

Public Function.

This function catches anything trying to write to /proc/brandxlo. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

Definition at line 237 of file loopback.c.

References debug, linkQueryRead(), lo_timer, loQuery, loQueryData(), loQueryName, loRegistered, loTimerInterval, loTimerRange, loTrigger, and NAMESIZE.

A.24.2.4 static int loEnvironmentUpdate (struct net_device_stats * *lb_stats*) [static]

/brief Public Function

This function updates the internal cache stored in the local protocol layer. This is utilized in decreasing response time with a Synchronized Pull architecture.

Definition at line 425 of file loopback.c.

References debug, packetData::dropped, loData, packetData::packets, packetData::rxBytes, packetData::rxErrors, packetData::txBytes, and packetData::txErrors.

A.24.2.5 static void loQueryData (unsigned long *input*) [static]

Private Function.

Timer activated information update mechanism. This function is utilized during information updates based on a Timer activation mechanism. Special care must be taken to remove the registered tcp_timer value before this function is removed or is will cause a segmenation fault.

Definition at line 128 of file loopback.c.

References debug, lo_func, lo_sem, lo_timer, loQuery, loQueryName, loTimerInterval, loTrigger, and TIMER.

Referenced by loCatchProcWrite().

A.24.3 Variable Documentation

A.24.3.1 int debug = 0 [static]

Stores the current DEBUG level

Definition at line 95 of file loopback.c.

A.24.3.2 int(* lo_func)(char *name, int value) [static]

Stores the pointer to the function launched with the timer object

Definition at line 87 of file loopback.c.

Referenced by linkQueryRead(), and loQueryData().

A.24.3.3 struct semaphore lo_sem

Stores the semaphore for the local layer race condition control

Definition at line 97 of file loopback.c.

Referenced by loQueryData().

A.24.3.4 struct proc_dir_entry* lo_test_entry [static]

Stores the entry for the proc file system in the Linux Kernel

Definition at line 88 of file loopback.c.

A.24.3.5 struct timer_list lo_timer [static]

Stores the Linux kernel timer object utilized with a Timer Activation Mechanism

Definition at line 86 of file loopback.c.

Referenced by loCatchProcWrite(), and loQueryData().

A.24.3.6 struct packetData loData

Referenced by loEnvironmentUpdate().

A.24.3.7 int loQuery = 0 [static]

Stores if the protocol layer is actively performing information queries

Definition at line 91 of file loopback.c.

Referenced by loCatchProcRead(), loCatchProcWrite(), and loQueryData().

A.24.3.8 char loQueryName[NAMESIZE] [static]

Stores the names of the different architectures utilized in the experiments

Definition at line 93 of file loopback.c.

Referenced by linkQueryRead(), loCatchProcWrite(), and loQueryData().

A.24.3.9 int loRegistered = 0 [static]

Stores if the protocol layer IMC functions have been registered with the Linux Kernel

Definition at line 94 of file loopback.c.

Referenced by loCatchProcRead(), and loCatchProcWrite().

A.24.3.10 int loState = 0 [static]

Stores the current state of the information query

Definition at line 96 of file loopback.c.

A.24.3.11 int loTimerInterval = 1000 [static]

Defines the timer interval for Timer driven information querys

Definition at line 89 of file loopback.c.

Referenced by loCatchProcRead(), loCatchProcWrite(), and loQueryData().

A.24.3.12 int loTimerRange = 100 [static]

Defines the timer range for Event driven information querys

Definition at line 90 of file loopback.c.

Referenced by loCatchProcRead(), and loCatchProcWrite().

A.24.3.13 int loTrigger = 0 [static]

Defines the activation mechanims utilized in the current experiment

Definition at line 92 of file loopback.c.

Referenced by loCatchProcRead(), loCatchProcWrite(), and loQueryData().

A.25 phystub.c File Reference

Functions

- unsigned int **main_hook** (unsigned int hooknum, struct sk_buff ∗∗skb, const struct net_device ∗in, const struct net_device ∗out, int(∗okfn)(struct sk_buff ∗))

  Public Function Protype.

- int **otp_func** (struct sk_buff ∗skb, struct device ∗dv, struct packet_type ∗pt)

  Public Function.

- __u32 **in_aton** (const char ∗str)
- static void **catchProcWrite** (struct file ∗file, const char __user ∗buffer, unsigned long count, void ∗data)

  Public Function.

- int **init_module** ()

  Module Initialization Function.

- void **cleanup_module** ()

  Module Destroy Function.

A.25.1 Function Documentation

A.25.1.1 static void catchProcWrite (struct file ∗ *file*, const char __user ∗ *buffer*, unsigned long *count*, void ∗ *data*) [static]

Public Function.

This function catches anything trying to write to /proc/phystub. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

Start the query process

Stop the query process

Set the ipTrigger mechanism

Set the timer delay

Definition at line 204 of file phystub.c.

References debug, and main_hook().

A.25.1.2 void cleanup_module (void)

Module Destroy Function.

Called when the module is unloaded into the Linux Kernel. All functions are implemented as pluggable kernel modules. This function unregisters the inter-process communication functions and removes the proc file system entries.

Definition at line 317 of file phystub.c.

A.25.1.3 __u32 in_aton (const char * *str*)

Convert an ASCII string to binary IP.

Definition at line 169 of file phystub.c.

Referenced by otp_func().

A.25.1.4 int init_module (void)

Module Initialization Function.

register the function for the proc FS

Definition at line 305 of file phystub.c.

References catchProcWrite().

A.25.1.5 unsigned int main_hook (unsigned int *hooknum*, struct sk_buff ** *skb*, const struct net_device * *in*, const struct net_device * *out*, int(*)(struct sk_buff *) *okfn*)

Public Function Protype.

Function prototype in <linux/netfilter>. This function is hooked everytime a packet is processed in the network protocol stack. This is utilized to change any packet values necessary and add latency to simulate various ambiant wireless environments.

Definition at line 89 of file phystub.c.

References debug.

Referenced by catchProcWrite().

A.25.1.6 int otp_func (struct sk_buff * *skb*, struct device * *dv*, struct packet_type * *pt*)

Public Function.

Packet Handler Function

Definition at line 125 of file phystub.c.

References in_aton().

A.26 qotstub.c File Reference

```
#include "qotstub.h"
```

Functions

- static int **qotCatchProcRead** (char ∗buf, char ∗∗start, off_t offset, int len, int ∗unused, void ∗data)

  A public function.

- static int **qotCatchProcWrite** (struct file ∗file, const char __user ∗buffer, unsigned long count, void ∗data)

  Public Function.

- int **trafficStation** (struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t size)

  Public Function.

- int **udpTrafficStation** (struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t size)

  Public Function.

- void **setDelay** (int millisecondDelay)

  Public Function.

- int **getDelay** ()

  Public Function.

- void **setTimerInterval** (int newInterval)

  Public Function.

- int **getTimerInterval** ()

  Public Function.

- void **setTrigger** (int newTrigger)

  Public Function.

- int **getTrigger** ()

  Public Function.

- static int **qotEventQuery** (unsigned long input)

Public Function.

- static void **qotTimerQuery** (unsigned long input)

  Public Function.

- void **writeData** (const char ∗name, int value)

  Public Function.

- int **readData** (char ∗name)

  Private Function.

- void **qot_write** (char ∗name, int value)

  Public InterModule Communication Functions.

- int **qot_read** (char ∗name)

  Public InterModule Communication Functions.

- void **regCallbackFunctions** ()

  Public Function Register the cross intermodule callback functions, billboard_write and billboard_read.

- void **unregCallbackFunctions** ()

  Public Function Unregister the cross intermodule callback functions, billboard_write and billboard_read.

- void **insertIntoStack** ()

  Private Function.

- void **removeFromStack** ()

  Private Function.

- int **init_module** ()

  Module Initialization Function.

- void **cleanup_module** ()

  Module Destroy Function.

- **MODULE_AUTHOR** ("Greg DeHart")
- **MODULE_DESCRIPTION** ("QoT Stub")
- **MODULE_LICENSE** ("GPL")

A.26.1 Function Documentation

A.26.1.1 void cleanup_module (void)

Module Destroy Function.

This function removes the module; it simply unregisters the directory entry from the /proc file system and releases the Brand X callback.

unregister the function from the proc FS

Definition at line 788 of file qotstub.c.

References unregCallbackFunctions(), and unregProcFunctions().

A.26.1.2 int getDelay (void)

Public Function.

Return the current QoT simulation delay.

Definition at line 395 of file qotstub.c.

References qotDelay.

A.26.1.3 int getTimerInterval (void)

Public Function.

Return the QoT environment information query interval in milliseconds.

Definition at line 416 of file qotstub.c.

References qotTimerRange.

A.26.1.4 int getTrigger (void)

Public Function.

Return the current trigger mechanism. **TIMER(0)**(p. 226) or **EVENT(1)**(p. 226) are the two trigger mechanisms available.

Definition at line 437 of file qotstub.c.

References qotTrigger.

A.26.1.5 int init_module (void)

Module Initialization Function.

This function is called by the Linux kernel when the module is loaded. It performs basic setup routines and enters the proc file system hooks.

Set the query flag to false (0)

Definition at line 753 of file qotstub.c.

References qot_sem, qotCatchProcRead(), qotCatchProcWrite(), qotDelay, qotQuery, qotTimerRange, qotTrigger, regCallbackFunctions(), regProcFunctions(), and TIMER.

A.26.1.6 void insertIntoStack (void)

Private Function.

Insert the QoT layer into the existing TCP/IP Linux network protocol stack. This is accomplished by redirecting the current function pointer structure to reference QoT functions as a pass through system.

Definition at line 719 of file qotstub.c.

References originalTCPSend, originalUDPSend, trafficStation(), and udpTrafficStation().

Referenced by qotCatchProcWrite().

A.26.1.7 MODULE_AUTHOR ("Greg DeHart")

A.26.1.8 MODULE_DESCRIPTION ("QoT Stub")

A.26.1.9 MODULE_LICENSE ("GPL")

A.26.1.10 int qot_read (char * *name*)

Public InterModule Communication Functions.

This function is registered with the Linux Kernel and is utilized to pass data between network protocol layers, Brand X and the QoT layer. Brand X utilizes this function to read configuration settings from QoT.

Definition at line 680 of file qotstub.c.

References readData().

Referenced by regCallbackFunctions().

A.26.1.11 void qot_write (char * *name*, int *value*)

Public InterModule Communication Functions.

This function is registered with the Linux Kernel and is utilized to pass data between network protocol layers, Brand X and the QoT layer. The network protocol layers and Brand X utilize this function to pass data to QoT.

Definition at line 668 of file qotstub.c.

References writeData().

Referenced by regCallbackFunctions().

A.26.1.12 static int qotCatchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *unused*, void * *data*) [static]

A public function.

This function is what the /proc FS will call when anything tries to read /proc/qotstub.

Parameters:

buf a character pointer

start a double character pointer

offset an offset into start

len an integer for the size of buf

unused an integer

Definition at line 14 of file qotstub.c.

References qotTrigger.

Referenced by init_module().

A.26.1.13 static int qotCatchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*) [static]

Public Function.

This function catches anything trying to write to /proc/qotstub. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

Stop the query process

Set the timer delay

Set the trigger mechanism

Set the trigger mechanism

Set the timer delay

Set the ipTrigger mechanism

Register the query functions

Cleanup the query functions

release Brand X callback function

Insert into TCP/IP protocol stack

Remove from TCP/IP protocol stack

Definition at line 35 of file qotstub.c.

References appFunc, brandxFunc, debug, insertIntoStack(), ipFunc, linkFunc, MAXPROC, qot_timer, qotDelay, qotMode, qotQuery, qotTimerQuery(), qotTimerRange, qotTrigger, removeFromStack(), tcpFunc, and TRIGGER.

Referenced by init_module().

A.26.1.14 static int qotEventQuery (unsigned long *input*) [static]

Public Function.

This method is called from the traffic handling functions based on current traffic patterns. This function is called from within the semaphore in order to ensure we don't reenter the calls to other protocols and overright the return data.

Definition at line 450 of file qotstub.c.

References appFunc, debug, ipFunc, linkFunc, qotQuery, qotTimerRange, and tcpFunc.

Referenced by qotTimerQuery(), trafficStation(), and udpTrafficStation().

A.26.1.15 static void qotTimerQuery (unsigned long *input*) [static]

Public Function.

This method performs periodic environment queries. The queries are based on the input for the specific experiment. The experiment values are required to be pre-determined and setup in the QoT environment.

Make a call to brand X to retrieve data

Definition at line 509 of file qotstub.c.

References BRANDX, debug, qot_sem, qot_timer, qotEventQuery(), qotMode, qotQuery, qotTimerRange, and qotTrigger.

Referenced by qotCatchProcWrite().

A.26.1.16 int readData (char * *name*)

Private Function.

Read a value from the Brand X billboard for use in the QoT internal decision algorithms

Definition at line 633 of file qotstub.c.

References data_store, and debug.

Referenced by qot_read().

A.26.1.17 void regCallbackFunctions (void)

Public Function Register the cross intermodule callback functions, billboard_write and billboard_read.

Register the cross intermodule callback functions

Definition at line 691 of file qotstub.c.

References billboard_read(), billboard_write(), logData(), qot_read(), qot_write(), and SYSLOG.

A.26.1.18 void removeFromStack (void)

Private Function.

This function restores the original TCP/IP Linux network protocol stack. The QoT functional layer is removed from any processing.

put the pointer back to tcp's original message sender

Definition at line 736 of file qotstub.c.

References originalTCPSend, and originalUDPSend.

Referenced by qotCatchProcWrite().

A.26.1.19 void setDelay (int *millisecondDelay*)

Public Function.

Set the time, in milliseconds, that QoT will delay a packet in the TCP/IP stack as it passes through. Initially set to 10 milliseconds.

Definition at line 385 of file qotstub.c.

References qotDelay.

A.26.1.20 void setTimerInterval (int *newInterval*)

Public Function.

Set the time interval, in milliseconds, that QoT will query for environment information. This is only used if the Trigger Mechanism is set to TIMER.

Definition at line 407 of file qotstub.c.

References qotTimerRange.

A.26.1.21 void setTrigger (int *newTrigger*)

Public Function.

Set the Trigger mechanism for environmental information querying. **TIMER(0)**(p. 226) or **EVENT(1)**(p. 226) are the two trigger mechanisms available.

Definition at line 427 of file qotstub.c.

References qotTrigger.

A.26.1.22 int trafficStation (struct kiocb * *iocb*, struct sock * *sk*, struct msghdr * *msg*, size_t *size*)

Public Function.

This function intercepts all traffic between the upper protocol layers and the TCP protocol. Data packets are routed through this function for any cross-layer processing necessary.

Definition at line 267 of file qotstub.c.

References debug, EVENT, qot_sem, qotDelay, qotEventQuery(), qotQuery, qotState, qotTimerRange, and qotTrigger.

Referenced by insertIntoStack().

A.26.1.23 int udpTrafficStation (struct kiocb * *iocb*, struct sock * *sk*, struct msghdr * *msg*, size_t *size*)

Public Function.

This function intercepts all traffic between the upper protocol layers and the UDP protocol. Data packets are routed through this function for any cross-layer processing necessary.

Definition at line 327 of file qotstub.c.

References debug, EVENT, qot_sem, qotDelay, qotEventQuery(), qotQuery, qotState, qotTimerRange, and qotTrigger.

Referenced by insertIntoStack().

A.26.1.24 void unregCallbackFunctions (void)

Public Function Unregister the cross intermodule callback functions, billboard_-write and billboard_read.

Unregister the cross intermodule callback functions

Definition at line 704 of file qotstub.c.

References logData(), and SYSLOG.

A.26.1.25 void writeData (const char * *name*, int *value*)

Public Function.

Write a name-value pair to the Brand X billboard for configuration of the network protocol stack.

Definition at line 565 of file qotstub.c.

References data_store, debug, and qot_sem.

Referenced by qot_write().

A.27 qotstub.h File Reference

```
#include <linux/kernel.h>
#include <linux/netdevice.h>
#include <net/tcp.h>
#include <net/ip.h>
#include <linux/skbuff.h>
#include <linux/tcp.h>
#include <net/udp.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/delay.h>
#include <linux/errno.h>
#include <linux/timer.h>
#include <asm/delay.h>
#include <linux/random.h>
```

Functions

- static int **qotCatchProcRead** (char *buf, char **start, off_t offset, int len, int *unused, void *data)

  A public function.

- static int **qotCatchProcWrite** (struct file *file, const char __user *buffer, unsigned long count, void *data)

  Public Function.

- int **udpTrafficStation** (struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t size)

  Public Function.

- int **trafficStation** (struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t size)

  Public Function.

- void **setDelay** (int millisecondDelay)

  Public Function.

- int **getDelay** (void)

  Public Function.

- void **setTimerInterval** (int newInterval)

  Public Function.

- int **getTimerInterval** (void)

  Public Function.

- void **setTrigger** (int newTrigger)

  Public Function.

- int **getTrigger** (void)

  Public Function.

- static void **qotTimerQuery** (unsigned long input)

  Public Function.

- static int **qotEventQuery** (unsigned long input)

  Public Function.

- int **init_module** (void)

  Module Initialization Function.

- void **cleanup_module** (void)

  Module Destroy Function.

- void **regCallbackFunctions** (void)

  Public Function Register the cross intermodule callback functions, billboard_write and billboard_read.

- void **unregCallbackFunctions** (void)

  Public Function Unregister the cross intermodule callback functions, billboard_write and billboard_read.

- int **qot_read** (char ∗name)

  Public InterModule Communication Functions.

- void **qot_write** (char ∗name, int value)

  Public InterModule Communication Functions.

- int **readData** (char ∗name)

  Private Function.

- void **writeData** (const char ∗name, int value)

  Public Function.

- void **insertIntoStack** (void)

  Private Function.

- void **removeFromStack** (void)

  Private Function.

Variables

- int(∗ **originalUDPSend**)(struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t size)
- int(∗ **originalTCPSend**)(struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t size)
- int(∗ **originalRecv**)(struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t len, int nonblock, int flags, int ∗addr_len)
- static int **qotDelay**
- static int **qotTimerRange**
- static int **qotTrigger**
- static int **qotQuery**
- timer_list **qot_timer**
- static int(∗ **brandxFunc**)(char ∗name)
- static int(∗ **appFunc**)(char ∗name)
- static int(∗ **tcpFunc**)(char ∗name)
- static int(∗ **ipFunc**)(char ∗name)
- static int(∗ **linkFunc**)(char ∗name)
- static int **debug** = 0
- static int **qotState** = 0
- static int **qotMode** = 0
- semaphore **qot_sem**
- protocol_data **data_store** [5]
- const char **TRIGGER** [ ] = "trigger"

A.27.1 Function Documentation

A.27.1.1 void cleanup_module (void)

Module Destroy Function.

This function removes the module; it simply unregisters the directory entry from the /proc file system and releases the Brand X callback.

unregister the function from the proc FS

Definition at line 348 of file brandx.c.

References unregCallbackFunctions(), and unregProcFunctions().

A.27.1.2 int getDelay (void)

Public Function.

Return the current QoT simulation delay.

Definition at line 395 of file qotstub.c.

References qotDelay.

A.27.1.3 int getTimerInterval (void)

Public Function.

Return the QoT environment information query interval in milliseconds.

Definition at line 416 of file qotstub.c.

References qotTimerRange.

A.27.1.4 int getTrigger (void)

Public Function.

Return the current trigger mechanism. **TIMER(0)**(p. 226) or **EVENT(1)**(p. 226) are the two trigger mechanisms available.

Definition at line 437 of file qotstub.c.

References qotTrigger.

A.27.1.5 int init_module (void)

Module Initialization Function.

This function is called by the Linux kernel when the module is loaded. It performs basic setup routines and enters the proc file system hooks.

Set the query flag to false (0)

Definition at line 334 of file brandx.c.

References qot_sem, qotCatchProcRead(), qotCatchProcWrite(), qotDelay, qotQuery, qotTimerRange, qotTrigger, regCallbackFunctions(), regProcFunctions(), and TIMER.

A.27.1.6 void insertIntoStack (void)

Private Function.

Insert the QoT layer into the existing TCP/IP Linux network protocol stack. This is accomplished by redirecting the current function pointer structure to reference QoT functions as a pass through system.

Definition at line 719 of file qotstub.c.

References originalTCPSend, originalUDPSend, trafficStation(), and udpTrafficStation().

Referenced by qotCatchProcWrite().

A.27.1.7 int qot_read (char * *name*)

Public InterModule Communication Functions.

This function is registered with the Linux Kernel and is utilized to pass data between network protocol layers, Brand X and the QoT layer. Brand X utilizes this function to read configuration settings from QoT.

Definition at line 680 of file qotstub.c.

References readData().

Referenced by regCallbackFunctions().

A.27.1.8 void qot_write (char * *name*, int *value*)

Public InterModule Communication Functions.

This function is registered with the Linux Kernel and is utilized to pass data between network protocol layers, Brand X and the QoT layer. The network protocol layers and Brand X utilize this function to pass data to QoT.

Definition at line 668 of file qotstub.c.

References writeData().

Referenced by regCallbackFunctions().

A.27.1.9 static int qotCatchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *unused*, void * *data*) [static]

A public function.

This function is what the /proc FS will call when anything tries to read /proc/qotstub.

Parameters:

buf a character pointer

start a double character pointer

offset an offset into start

len an integer for the size of buf

unused an integer

A.27.1.10 static int qotCatchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*) [static]

Public Function.

This function catches anything trying to write to /proc/qotstub. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

A.27.1.11 static int qotEventQuery (unsigned long *input*) [static]

Public Function.

This method is called from the traffic handling functions based on current traffic patterns. This function is called from within the semaphore in order to ensure we don't reenter the calls to other protocols and overright the return data.

A.27.1.12 static void qotTimerQuery (unsigned long *input*) [static]

Public Function.

This method performs periodic environment queries. The queries are based on the input for the specific experiment. The experiment values are required to be pre-determined and setup in the QoT environment.

A.27.1.13 int readData (char * *name*)

Private Function.

Read a value from the Brand X billboard for use in the QoT internal decision algorithms

Definition at line 633 of file qotstub.c.

References data_store, and debug.

Referenced by qot_read().

A.27.1.14 void regCallbackFunctions (void)

Public Function Register the cross intermodule callback functions, billboard_write and billboard_read.

Register the cross intermodule callback functions

Definition at line 311 of file brandx.c.

References billboard_read(), billboard_write(), logData(), qot_read(), qot_write(), and SYSLOG.

A.27.1.15 void removeFromStack (void)

Private Function.

This function restores the original TCP/IP Linux network protocol stack. The QoT functional layer is removed from any processing.

put the pointer back to tcp's original message sender

Definition at line 736 of file qotstub.c.

References originalTCPSend, and originalUDPSend.

Referenced by qotCatchProcWrite().

A.27.1.16 void setDelay (int *millisecondDelay*)

Public Function.

Set the time, in milliseconds, that QoT will delay a packet in the TCP/IP stack as it passes through. Initially set to 10 milliseconds.

Definition at line 385 of file qotstub.c.

References qotDelay.

A.27.1.17 void setTimerInterval (int *newInterval*)

Public Function.

Set the time interval, in milliseconds, that QoT will query for environment information. This is only used if the Trigger Mechanism is set to TIMER.

Definition at line 407 of file qotstub.c.

References qotTimerRange.

A.27.1.18 void setTrigger (int *newTrigger*)

Public Function.

Set the Trigger mechanism for environmental information querying. **TIMER(0)**(p. 226) or **EVENT(1)**(p. 226) are the two trigger mechanisms available.

Definition at line 427 of file qotstub.c.

References qotTrigger.

A.27.1.19 int trafficStation (struct kiocb * *iocb*, struct sock * *sk*, struct msghdr * *msg*, size_t *size*)

Public Function.

This function intercepts all traffic between the upper protocol layers and the TCP protocol. Data packets are routed through this function for any cross-layer processing necessary.

Definition at line 267 of file qotstub.c.

References debug, EVENT, qot_sem, qotDelay, qotEventQuery(), qotQuery, qotState, qotTimerRange, and qotTrigger.

Referenced by insertIntoStack().

A.27.1.20 int udpTrafficStation (struct kiocb * *iocb*, struct sock * *sk*, struct msghdr * *msg*, size_t *size*)

Public Function.

This function intercepts all traffic between the upper protocol layers and the UDP protocol. Data packets are routed through this function for any cross-layer processing necessary.

Definition at line 327 of file qotstub.c.

References debug, EVENT, qot_sem, qotDelay, qotEventQuery(), qotQuery, qotState, qotTimerRange, and qotTrigger.

Referenced by insertIntoStack().

A.27.1.21 void unregCallbackFunctions (void)

Public Function Unregister the cross intermodule callback functions, billboard_-write and billboard_read.

Unregister the cross intermodule callback functions

Definition at line 320 of file brandx.c.

References logData(), and SYSLOG.

A.27.1.22 void writeData (const char * *name*, int *value*)

Public Function.

Write a name-value pair to the Brand X billboard for configuration of the network protocol stack.

Definition at line 565 of file qotstub.c.

References data_store, debug, and qot_sem.

Referenced by qot_write().

A.27.2 Variable Documentation

A.27.2.1 int(* appFunc)(char *name) [static]

Stores the function pointer to the application protocol layer IMC function

Definition at line 38 of file qotstub.h.

Referenced by qotCatchProcWrite(), and qotEventQuery().

A.27.2.2 int(* brandxFunc)(char *name) [static]

Stores the function pointer to the Brand X IMC function

Definition at line 37 of file qotstub.h.

Referenced by qotCatchProcWrite().

A.27.2.3 struct protocol_data data_store[5]

Stores the current status of the network protocol stack

Definition at line 46 of file qotstub.h.

Referenced by readData(), and writeData().

A.27.2.4 int debug = 0 [static]

Stores if the layer is in debug mode

Definition at line 42 of file qotstub.h.

Referenced by appCatchProcWrite(), appEnvironmentUpdate(), appQueryData(), appQueryRead(), billboard_write(), catchProcWrite(), ip_output(), ip_queue_xmit(), ipCatchProcWrite(), ipEnvironmentUpdate(), ipQueryData(), ipQueryRead(), linkQueryRead(), loCatchProcWrite(), loEnvironmentUpdate(), loQueryData(), main_hook(), qotCatchProcWrite(), qotEventQuery(), qotTimerQuery(), readData(), readFromBillboard(), tcpCatchProcWrite(), tcpEnvironmentUpdate(), tcpQueryData(), tcpQueryRead(), trafficStation(), udpCatchProcWrite(), udpEnvironmentUpdate(), udpQueryData(), udpTrafficStation(), writeData(), and writeOnBillboard().

A.27.2.5 int(∗ ipFunc)(char ∗name) [static]

Stores the function pointer to the IP protocol layer IMC function

Definition at line 40 of file qotstub.h.

Referenced by qotCatchProcWrite(), and qotEventQuery().

A.27.2.6 int(∗ linkFunc)(char ∗name) [static]

Stores the function pointer to the MAC protocol layer IMC function

Definition at line 41 of file qotstub.h.

Referenced by qotCatchProcWrite(), and qotEventQuery().

A.27.2.7 int(∗ originalRecv)(struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t len, int nonblock, int flags, int ∗addr_len)

Stores the original TCP function pointer for receiving data from the network protocol stack

Definition at line 29 of file qotstub.h.

A.27.2.8 int(∗ originalTCPSend)(struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t size)

Stores the original TCP function pointer for sending data down the network protocol stack

Definition at line 28 of file qotstub.h.

Referenced by insertIntoStack(), and removeFromStack().

A.27.2.9 int(∗ originalUDPSend)(struct kiocb ∗iocb, struct sock ∗sk, struct msghdr ∗msg, size_t size)

Stores the original UDP function pointer for the network protocol stack

Definition at line 27 of file qotstub.h.

Referenced by insertIntoStack(), and removeFromStack().

A.27.2.10 struct semaphore qot_sem

Stores the semaphore to protect against overwriting data

Definition at line 45 of file qotstub.h.

Referenced by init_module(), qotTimerQuery(), trafficStation(), udpTrafficStation(), and writeData().

A.27.2.11 struct timer_list qot_timer

Stores the timer object to register with the Linux Kernel timer mechanism

Definition at line 36 of file qotstub.h.

Referenced by qotCatchProcWrite(), and qotTimerQuery().

A.27.2.12 int qotDelay [static]

Stores the delay time in microseconds for a network workload packet to pass through the QoT layer

Definition at line 32 of file qotstub.h.

Referenced by getDelay(), init_module(), qotCatchProcWrite(), setDelay(), trafficStation(), and udpTrafficStation().

A.27.2.13 int qotMode = 0 [static]

Stores the mode for the current experiment

Definition at line 44 of file qotstub.h.

Referenced by qotCatchProcWrite(), and qotTimerQuery().

A.27.2.14 int qotQuery [static]

Stores if the protocol is currently in the process of querying for environment information

Definition at line 35 of file qotstub.h.

Referenced by init_module(), qotCatchProcWrite(), qotEventQuery(), qotTimerQuery(), trafficStation(), and udpTrafficStation().

A.27.2.15 int qotState = 0 [static]

Stores the current state of the environment information query

Definition at line 43 of file qotstub.h.

Referenced by trafficStation(), and udpTrafficStation().

A.27.2.16 int qotTimerRange [static]

Stores the timer range for a Timer activation mechanism experiment

Definition at line 33 of file qotstub.h.

Referenced by getTimerInterval(), init_module(), qotCatchProcWrite(), qotEventQuery(), qotTimerQuery(), setTimerInterval(), trafficStation(), and udpTrafficStation().

A.27.2.17 int qotTrigger [static]

Stores the Activation trigger to be utilized, event or timer

Definition at line 34 of file qotstub.h.

Referenced by getTrigger(), init_module(), qotCatchProcRead(), qotCatchProcWrite(), qotTimerQuery(), setTrigger(), trafficStation(), and udpTrafficStation().

A.27.2.18 int(∗ tcpFunc)(char ∗name) [static]

Stores the function pointer to the TCP protocol layer IMC function

Definition at line 39 of file qotstub.h.

Referenced by qotCatchProcWrite(), and qotEventQuery().

A.27.2.19 const char TRIGGER[ ] = "trigger"

Definition at line 114 of file qotstub.h.

Referenced by qotCatchProcWrite().

A.28 ServerSocket.cpp File Reference

```
#include "ServerSocket.h"
#include "SocketException.h"
```

A.29 ServerSocket.h File Reference

```
#include "Socket.h"
```

Data Structures

- class **ServerSocket**

  Public Class.

A.30 Socket.cpp File Reference

```
#include "Socket.h"
#include "string.h"
#include <errno.h>
#include <fcntl.h>
#include <iostream>
```

A.31 Socket.h File Reference

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>
#include <unistd.h>
#include <string>
#include <arpa/inet.h>
```

Data Structures

- class **Socket**

  Public **Socket**(p. 130) *Class.*

Variables

- const int **MAXHOSTNAME** = 200
- const int **MAXCONNECTIONS** = 5
- const int **MAXRECV** = 66000

A.31.1 Variable Documentation

A.31.1.1 const int MAXCONNECTIONS = 5

Definition at line 18 of file Socket.h.

Referenced by Socket::listen().

A.31.1.2 const int MAXHOSTNAME = 200

Definition at line 17 of file Socket.h.

A.31.1.3 const int MAXRECV = 66000

Definition at line 19 of file Socket.h.

Referenced by Socket::recv().

A.32 SocketException.h File Reference

```
#include <string>
```

Data Structures

- class **SocketException**

  Public Class.

A.33 tabdialog.cpp File Reference

```
#include <QtGui>
#include "tabdialog.h"
```

A.34 tabdialog.h File Reference

Data Structures

- class **model**

  Public Model Class.

- class **TabDialog**

  Public Class.

- class **procTab**

  Public Class.

- class **configureTab**

  Public Class.

A.35 tcp.c File Reference

```
#include <linux/timer.h>
#include <linux/proc_fs.h>
```

Data Structures

- struct **packetData**

  Private Data Structure.

Defines

- #define **TIMER** 0
- #define **EVENT** 1
- #define **NAMESIZE** 512
- #define **THRESHOLD** 25

Functions

- static void **tcpQueryData** (unsigned long input)

  Private Function.

- int **tcpCatchProcRead** (char ∗buf, char ∗∗start, off_t offset, int len, int ∗unused, void ∗data)

  Public Function.

- int **tcpQueryRead** (char ∗name)

  Public Function.

- int **tcpCatchProcWrite** (struct file ∗file, const char __user ∗buffer, unsigned long count, void ∗data)

  Public Function.

- static int **tcpEnvironmentUpdate** (struct sk_buff ∗skb)

Variables

- static struct timer_list **tcp_timer**
- static int(∗ **tcp_func**)(char ∗name, int value)
- static struct proc_dir_entry ∗ **tcp_test_entry**

- static int **tcpTimerInterval** = 1000
- static int **tcpTimerRange** = 100
- static int **tcpQuery** = 0
- static int **tcpTrigger** = 0
- static char **tcpQueryName** [NAMESIZE]
- static int **tcpRegistered** = 0
- static int **debug** = 0
- static int **tcpState** = 0
- semaphore **tcp_sem**
- **packetData tcpData**

A.35.1 Define Documentation

A.35.1.1 #define EVENT 1

EVENT defines the Event Activation Mechanism

Definition at line 309 of file tcp.c.

A.35.1.2 #define NAMESIZE 512

NAMESIZE defines the maximim length of a name in a name-value pair

Definition at line 310 of file tcp.c.

A.35.1.3 #define THRESHOLD 25

THRESHOLD defines the change in environment status before a Information Query is initiated with an Event Activation Mechanism

Definition at line 311 of file tcp.c.

A.35.1.4 #define TIMER 0

TIMER defines the Timer Activation Mechanism

Definition at line 308 of file tcp.c.

A.35.2 Function Documentation

A.35.2.1 int tcpCatchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *unused*, void * *data*)

Public Function.

This function catches anything trying to read from /proc/brandxtcp. This function is called to handle the output of information to user space.

Parameters:

buf character pointer, The buffer where the data is to be inserted

start double character pointer, If you don't want to use the buffer allocated by the kernel

len int, Current position in the file

unused int, Size of the buffer in the first argument

data void pointer, For future use

Definition at line 412 of file tcp.c.

References tcpQuery, tcpRegistered, tcpTimerInterval, tcpTimerRange, and tcpTrigger.

A.35.2.2 int tcpCatchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*)

Public Function.

This function catches anything trying to write to /proc/brandxtcp. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

Set tcpQuery flag to false. Query will stop on next cycle

Definition at line 464 of file tcp.c.

References debug, NAMESIZE, tcp_timer, tcpQuery, tcpQueryData(), tcpQueryName, tcpQueryRead(), tcpRegistered, tcpTimerInterval, tcpTimerRange, and tcpTrigger.

A.35.2.3 static int tcpEnvironmentUpdate (struct sk_buff * *skb*) [static]

/brief Public Function

This function updates the internal cache stored in the local protocol layer. This is utilized in decreasing response time with a Synchronized Pull architecture.

Definition at line 646 of file tcp.c.

References debug, packetData::dropped, packetData::packets, packetData::rxBytes, packetData::rxErrors, tcpData, packetData::txBytes, and packetData::txErrors.

A.35.2.4 static void tcpQueryData (unsigned long *input*) [static]

Private Function.

Timer activated information update mechanism. This function is utilized during information updates based on a Timer activation mechanism. Special care must be taken to remove the registered tcp_timer value before this function is removed or is will cause a segmenation fault.

Make a call to brand X to retrieve data

Definition at line 356 of file tcp.c.

References debug, tcp_sem, tcp_timer, tcpQuery, tcpQueryName, tcpTimer-Interval, tcpTrigger, and TIMER.

Referenced by tcpCatchProcWrite().

A.35.2.5 int tcpQueryRead (char * *name*)

Public Function.

Intermodule Communication function, calls the registered callback function with updated environment information. This is utilized during the Synchronous Pull architecture.

Make a call to brand X to retrieve data

Definition at line 434 of file tcp.c.

References debug, and tcpQueryName.

Referenced by tcpCatchProcWrite().

A.35.3 Variable Documentation

A.35.3.1 int debug = 0 [static]

Stores the current DEBUG level

Definition at line 322 of file tcp.c.

A.35.3.2 int(* tcp_func)(char *name, int value) [static]

Stores the pointer to the function launched with the timer object

Definition at line 314 of file tcp.c.

A.35.3.3 struct semaphore tcp_sem

Stores the semaphore for the local layer race condition control

Definition at line 324 of file tcp.c.

Referenced by tcpQueryData().

A.35.3.4 struct proc_dir_entry* tcp_test_entry [static]

Stores the entry for the proc file system in the Linux Kernel

Definition at line 315 of file tcp.c.

A.35.3.5 struct timer_list tcp_timer [static]

Stores the Linux kernel timer object utilized with a Timer Activation Mechanism

Definition at line 313 of file tcp.c.

Referenced by tcpCatchProcWrite(), and tcpQueryData().

A.35.3.6 struct packetData tcpData

Referenced by readFromBillboard(), tcpEnvironmentUpdate(), and writeOnBillboard().

A.35.3.7 int tcpQuery = 0 [static]

Stores if the protocol layer is actively performing information queries

Definition at line 318 of file tcp.c.

Referenced by tcpCatchProcRead(), tcpCatchProcWrite(), and tcpQueryData().

A.35.3.8 char tcpQueryName[NAMESIZE] [static]

Stores the names of the different architectures utilized in the experiments

Definition at line 320 of file tcp.c.

Referenced by tcpCatchProcWrite(), tcpQueryData(), and tcpQueryRead().

A.35.3.9 int tcpRegistered = 0 [static]

Stores if the protocol layer IMC functions have been registered with the Linux Kernel

Definition at line 321 of file tcp.c.

Referenced by tcpCatchProcRead(), and tcpCatchProcWrite().

A.35.3.10 int tcpState = 0 [static]

Stores the current state of the information query

Definition at line 323 of file tcp.c.

A.35.3.11 int tcpTimerInterval = 1000 [static]

Defines the timer interval for Timer driven information querys

Definition at line 316 of file tcp.c.

Referenced by tcpCatchProcRead(), tcpCatchProcWrite(), and tcpQueryData().

A.35.3.12 int tcpTimerRange = 100 [static]

Defines the timer range for Event driven information querys

Definition at line 317 of file tcp.c.

Referenced by tcpCatchProcRead(), and tcpCatchProcWrite().

A.35.3.13 int tcpTrigger = 0 [static]

Defines the activation mechanims utilized in the current experiment

Definition at line 319 of file tcp.c.

Referenced by tcpCatchProcRead(), tcpCatchProcWrite(), and tcpQueryData().

A.36 udp.c File Reference

```
#include <linux/timer.h>
#include <linux/proc_fs.h>
#include <linux/random.h>
```

Data Structures

- struct **packetData**

  Private Data Structure.

Defines

- #define **TIMER** 0
- #define **EVENT** 1
- #define **NAMESIZE** 512
- #define **THRESHOLD** 25

Functions

- static void **udpQueryData** (unsigned long input)

  Private Function.

- int **udpCatchProcRead** (char ∗buf, char ∗∗start, off_t offset, int len, int ∗unused, void ∗data)

  Public Function.

- static int **linkQueryRead** (char ∗name)

  Public Function.

- int **udpCatchProcWrite** (struct file ∗file, const char __user ∗buffer, unsigned long count, void ∗data)

  Public Function.

- static int **udpEnvironmentUpdate** (struct sock ∗sk)

Variables

- static struct timer_list **udp_timer**
- static int(∗ **udp_func**)(char ∗name, int value)
- static struct proc_dir_entry ∗ **udp_test_entry**
- static int **udpTimerInterval** = 1000
- static int **udpTimerRange** = 100
- static int **udpQuery** = 0
- static int **udpTrigger** = 0
- static char **udpQueryName** [NAMESIZE]
- static int **udpRegistered** = 0
- static int **debug** = 0
- static int **udpState** = 0
- semaphore **udp_sem**
- **packetData udpData**

A.36.1 Define Documentation

A.36.1.1 #define EVENT 1

EVENT defines the Event Activation Mechanism

Definition at line 141 of file udp.c.

A.36.1.2 #define NAMESIZE 512

NAMESIZE defines the maximim length of a name in a name-value pair

Definition at line 142 of file udp.c.

A.36.1.3 #define THRESHOLD 25

THRESHOLD defines the change in environment status before a Information Query is initiated with an Event Activation Mechanism

Definition at line 143 of file udp.c.

A.36.1.4 #define TIMER 0

TIMER defines the Timer Activation Mechanism

Definition at line 140 of file udp.c.

A.36.2 Function Documentation

A.36.2.1 static int linkQueryRead (char * *name*) [static]

Public Function.

Intermodule Communication function, calls the registered callback function with updated environment information. This is utilized during the Synchronous Pull architecture.

Definition at line 268 of file udp.c.

References debug, udp_func, and udpQueryName.

A.36.2.2 int udpCatchProcRead (char * *buf*, char ** *start*, off_t *offset*, int *len*, int * *unused*, void * *data*)

Public Function.

This function catches anything trying to read from /proc/brandxudp. This function is called to handle the output of information to user space.

Parameters:

buf character pointer, The buffer where the data is to be inserted

start double character pointer, If you don't want to use the buffer allocated by the kernel

len int, Current position in the file

unused int, Size of the buffer in the first argument

data void pointer, For future use

Definition at line 247 of file udp.c.

References udpQuery, udpRegistered, udpTimerInterval, udpTimerRange, and udpTrigger.

A.36.2.3 int udpCatchProcWrite (struct file * *file*, const char __user * *buffer*, unsigned long *count*, void * *data*)

Public Function.

This function catches anything trying to write to /proc/brandxudp. This function is called to handle the input of information from user space.

Parameters:

file a file pointer, To the file to be read

buffer a character buffer, To put the data into (in the user segment)

count unsigned long, The length of the buffer

data void pointer, Offset in the file to store the data at

Definition at line 299 of file udp.c.

References debug, linkQueryRead(), NAMESIZE, udp_timer, udpQuery, udpQueryData(), udpQueryName, udpRegistered, udpTimerInterval, udpTimerRange, and udpTrigger.

A.36.2.4 static int udpEnvironmentUpdate (struct sock * *sk*) [static]

/brief Public Function

This function updates the internal cache stored in the local protocol layer. This is utilized in decreasing response time with a Synchronized Pull architecture.

Definition at line 475 of file udp.c.

References debug, packetData::dropped, packetData::packets, packetData::rxBytes, packetData::rxErrors, packetData::txBytes, packetData::txErrors, and udpData.

A.36.2.5 static void udpQueryData (unsigned long *input*) [static]

Private Function.

Timer activated information update mechanism. This function is utilized during information updates based on a Timer activation mechanism. Special care must be taken to remove the registered tcp_timer value before this function is removed or is will cause a segmenation fault.

Definition at line 187 of file udp.c.

References debug, TIMER, udp_func, udp_sem, udp_timer, udpQuery, udpQueryName, udpTimerInterval, and udpTrigger.

Referenced by udpCatchProcWrite().

A.36.3 Variable Documentation

A.36.3.1 int debug = 0 [static]

Stores the current DEBUG level

Definition at line 154 of file udp.c.

A.36.3.2 int(* udp_func)(char *name, int value) [static]

Stores the pointer to the function launched with the timer object

Definition at line 146 of file udp.c.

Referenced by linkQueryRead(), and udpQueryData().

A.36.3.3 struct semaphore udp_sem

Stores the semaphore for the local layer race condition control

Definition at line 156 of file udp.c.

Referenced by udpQueryData().

A.36.3.4 struct proc_dir_entry* udp_test_entry [static]

Stores the entry for the proc file system in the Linux Kernel

Definition at line 147 of file udp.c.

A.36.3.5 struct timer_list udp_timer [static]

Stores the Linux kernel timer object utilized with a Timer Activation Mechanism

Definition at line 145 of file udp.c.

Referenced by udpCatchProcWrite(), and udpQueryData().

A.36.3.6 struct packetData udpData

Referenced by readFromBillboard(), udpEnvironmentUpdate(), and writeOnBillboard().

A.36.3.7 int udpQuery = 0 [static]

Stores if the protocol layer is actively performing information queries

Definition at line 150 of file udp.c.

Referenced by udpCatchProcRead(), udpCatchProcWrite(), and udpQueryData().

A.36.3.8 char udpQueryName[NAMESIZE] [static]

Stores the names of the different architectures utilized in the experiments

Definition at line 152 of file udp.c.

Referenced by linkQueryRead(), udpCatchProcWrite(), and udpQueryData().

A.36.3.9 int udpRegistered = 0 [static]

Stores if the protocol layer IMC functions have been registered with the Linux Kernel

Definition at line 153 of file udp.c.

Referenced by udpCatchProcRead(), and udpCatchProcWrite().

A.36.3.10 int udpState = 0 [static]

Stores the current state of the information query

Definition at line 155 of file udp.c.

A.36.3.11 int udpTimerInterval = 1000 [static]

Defines the timer interval for Timer driven information querys

Definition at line 148 of file udp.c.

Referenced by udpCatchProcRead(), udpCatchProcWrite(), and udpQueryData().

A.36.3.12 int udpTimerRange = 100 [static]

Defines the timer range for Event driven information querys

Definition at line 149 of file udp.c.

Referenced by udpCatchProcRead(), and udpCatchProcWrite().

A.36.3.13 int udpTrigger = 0 [static]

Defines the activation mechanims utilized in the current experiment

Definition at line 151 of file udp.c.

Referenced by udpCatchProcRead(), udpCatchProcWrite(), and udpQueryData().

Bibliography

[1] H. Balakrishnan, “Challenges to reliable data transport over heterogeneous wireless networks.” Ph.D. dissertation, University of California at Berkeley, 1998.

[2] R. W. Woodings, S. B. Barnes, and C. D. Knutson, “Quality of transport (qot) protocol specification v1.0,” Mobile Computing Laboratory, Brigham Young University, Provo, UT, Provo, Utah, August 2002.

[3] G. Montenegro and S. Drach, “System isolation and network fast fail capability in Solaris,” *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pp. 67–78, April 1995.

[4] J. Postel, “Internet control message protocol,” RFC 792, DARPA Internet Program Protocol Specification, September 1981.

[5] P. Sudame and B. R. Badrinath, “On providing support for protocol adaptation in mobile wireless networks,” *Mobile Networks and Applications*, vol. 6, no. 1, pp. 43–55, 2001.

[6] G. Wu, Y. Bai, J. Lai, and A. Ogielski, “Interactions between tcp and rlp in wireless internet,” IEEE GlobeCom‘99, Rio de Janeiro, pp. 661–666, December 1999.

[7] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, “The anatomy of a context-aware application,” in *Mobile Computing and Networking*, 1999, pp. 59–68. [Online]. Available: citeseer.ist.psu.edu/article/harter02anatomy.html

[8] K. Chen, S. H. Shah, and K. Nahrstedt, “Cross-layer design for data accessibility in mobile ad hoc networks,” *Wireless Personal Communications*, vol. 21, no. 1, pp. 49–76, April 2002.

[9] D. D. Clark and D. L. Tennenhouse, “Architectural considerations for a new generation of protocols,” in *SIGCOMM ’90: Proceedings of the ACM symposium on Communications architectures & protocols*. New York, NY, USA: ACM Press, 1990, pp. 200–208.

[10] M. A. A.-R. Qi Wang, “Cross-layer signalling for next-generation wireless systems,” in *Proc. IEEE Wireless Communications and Networking Conference 2003 (WCNC’03)*, vol. 2, New Orleans, Louisiana, USA, March 2003, pp. 1084–1089.

[11] V. T. Raisinghani and S. Iyer, “Cross-layer design optimizations in wireless protocol stacks,” *Elsevier Computer Communications*, 2003. [Online]. Available: citeseer.ist.psu.edu/658477.html

[12] Y. Koucheryavy, D. Moltchanov, J. Harju, and G. Giambene, “Cross-layer black-box approach to performance evaluation of next generation mobile networks,” *IEEE International*, 2004.

[13] V. T. Raisinghani, A. K. Singh, and S. Iyer, “Improving tcp performance over mobile wireless environments using cross layer feedback,” *IEEE International Conference on Personal Wireless Communications*, New Delhi, India, 2002.

[14] Y. Fang and A. B. McDonald, “Cross-layer performance effects of path coupling in wireless ad hoc networks: Power and throughput implications of ieee 802.11 mac,” *IEEE International*, vol. 21, pp. 281–290, 2002.

[15] H. R. Duffin, C. D. Knutson, and M. A. Goodrich, “Prioritized soft constraint satisfaction: A qualitative method for dynamic transport selection in heterogeneous wireless

environments," in *Proceedings of the IEEE Wireless Communication and Networking Conference (WCNC 2004)*, Atlanta, Georgia, March 2004, pp. 2527–2532.

[16] S. B. Barnes, R. W. Woodings, and C. D. Knutson, "Transport discovery in wireless multi-transport environments," in *Proceedings of the IEEE Wireless Communications and Networking Conference*. New Orleans, Louisiana: IEEE, March 2003, pp. 1328–1333.

[17] C. D. Knutson, R. W. Woodings, S. B. Barnes, H. R. Duffin, and J. M. Brown, "Dynamic autonomous transport selection in heterogeneous wireless environments," in *Proceedings of the IEEE Wireless Communication and Networking Conference (WCNC 2004)*. Atlanta, Georgia: IEEE, March 2004, pp. 689–694.

[18] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[19] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring distributed systems," *ACM Transactions on Computer Systems*, pp. 151–150, May 1987. [Online]. Available: citeseer.ist.psu.edu/article/mansouri-samani93monitoring.html

[20] M. Mansouri-Samani and M. Sloman, "Monitoring distributed systems," Imerial College, Dept. of Computing, pp. 20–30, April 1993.

[21] M. L. Sichitiu, "Cross-layer scheduling for power efficiency in wireless sensor networks." Hong Kong, China: IEEE INFOCOM 2004, March 2004, pp. 1740–1750. [Online]. Available: citeseer.ist.psu.edu/article/sichitiu04crosslayer.html

[22] V. Kawadia and P. R. Kumar, "A cautionary perspective on cross layer design," IEEE Wireless Communications, vol. 12, no. 1, pp. 3–11, February 2005.

[23] A. J. Goldsmith and S. B. Wicker, "Design challenges for energy-constrained ad hoc wireless networks," *IEEE Wireless Communications*, vol. vol. 9, no. 4, pp. 8–27, 2002.

[24] S. Toumpis and A. J. Goldsmith, “Performance, optimization, and cross-layer design of media access protocols for wireless ad hoc networks,” in *Proceedings of the International Conference Communications*, Anchorage, AK, May 2003, pp. 2234 – 2240.

[25] I. Akyildiz, Y. Altunbasak, F. Fekri, and R. Sivakumar, “Adaptnet: An adaptive protocol suite for the next-generation wireless internet,” *IEEE Communications Magazine*, pp. 128–136, March 2004.

[26] T. M. Steinfatt, “The alpha percentage and experimentwise error rates in communication research,” in *Human Communication Research*, vol. 5, no. 4, June 1979, pp. 366 – 374.

[27] *Specifications of the Bluetooth System*, Vol 1, v1.2 ed., Bluetooth Special Interest Group, November 2003.