**Course: Advanced Networking Project Report**

**Group Members:**

**Name :Haq Ijaz Ul ID:2820150066**

**Name: Ron Wong See ID: 2820150151**

**Name: Gary Wolphagen ID: 2820150123**

**Name: Kuanyshev Torekhan ID:2820150038**

**Topic: Capturing Router Congestion and Delay**



# INTRODUCTION

**1.1 Introduction to Project**:

**E**ND-TO-END packet delay is one of the canonical metrics in Internet Protocol (IP) networks, and is important both from the network operator and application performance points of view. For example the quality of Voice Over IP is directly dependent on delay, and network providers may have Service Level Agreements (SLAs) specifying allowable values of delay statistics across the domains they control. An important component of end-to-end delay is that due to forwarding elements, the fundamental building block of which is the delay incurred when a packet passes through a single IP router.

The motivation for the present work is a detailed knowledge and understanding of such “through-router” delays. A thorough examination of delay leads inevitably to deeper questions about congestion, and router queueing dynamics in general. We provide a comprehensive examination of these issues from three points of view: the understanding of origins, measurement, and reporting, all grounded in a unique data set taken from a router in the access network of a Tier-1 provider.

 Although there have been many studies examining delay statistics and congestion measured at the edges of the network, very few have been able to report with any degree of authority on what actually occurs at switching elements. In an analysis of single hop delay on an IP backbone network was presented. Only samples of the delays experienced by packets, on some links, were identified. A single hop delays were also got for a router. However since the router only had one input and one output link, which were of the same speed, the internal queueing was extremely limited. In this paper we work with a data set recording all IP packets traversing a Tier-1 access router over a 13 hour period. All input and output links were monitored, allowing a complete picture of congestion and in particular router delays, to be obtained.

This paper’s objective is to exploit the unique data set by reporting in detail on the magnitudes, and also the temporal structure, of delays on high capacity links with nontrivial congestion. The result is one of the most comprehensive pictures of router delay performance that we are aware of. As our analysis is based on empirical results, it is not reliant on assumptions on traffic statistics or router operations.

To define a refined model with accuracy close to the limits of time stamping precision, which is robust to many details of the architecture under reasonable loads.Packet delays and congestion are fundamentally linked, as the former occur precisely because periods of temporary resource starvation, or *microcongestion episodes*, are dealt with via buffering. Our third contribution is an investigation of the origins of such episodes, driven by the question, “What is the dominant mechanism responsible for delays?”. We use a powerful methodology of virtual or “semi-” experiments, that exploits both the availability of the detailed packet data, and the fidelity of the router model. We identify, and evaluate the contributions of, three known canonical mechanisms:

1. Reduction in link bandwidth from core to access;
2. Multiplexing of multiple input streams;
3. Burstiness of the input traffic stream(s).

 To our knowledge, such a taxonomy of link congestion has not been used previously, and our findings for this specific data set are novel and sometimes surprising. Our broader contribution however is the methodology itself, including a set of metrics which can be used to examine the origins of congestion and hence delay.

The fourth part of the paper gives an innovative solution to a nontrivial problem: how the complexities of microcongestion and associated delay behavior can be measured and summarized in a meaningful way. We explain why our approach is superior to attempting to infer delay behavior simply from utilization, an approach which is in fact fatally flawed. In addition, we show how this can be done at low computational cost, enabling a compact description of congestion behavior to form part of standard router reporting. A key advantage is that a generically rich description is reported, without the need for any traffic assumptions.

2. **SYSTEM ANALYSIS**

**2.1 INTRODUCTION:**

Using a unique monitoring experiment, we capture all packets crossing a (lightly utilized) operational access router from a Tier-1 provider, and use them to provide a detailed examination of router congestion and packet delays. The complete capture enables not just statistics as seen from outside the router, but also an accurate physical router model to be identified. This enables a comprehensive examination of congestion and delay from three points of view: the understanding of origins, measurement, and reporting. Our study defines new methodologies and metrics. In particular, the traffic reporting enables a rich description of the diversity of micro congestion behavior, without model assumptions, and at achievable computational cost.

**2.2 EXISTING SYSTEM:**

**E**ND-TO-END packet delay is one of the canonical metrics in Internet Protocol (IP) networks, and is important both from the network operator and application performance points of view. For example the quality of Voice Over IP is directly dependent on delay, and network providers may have Service Level Agreements (SLAs) specifying allowable values of delay statistics across the domains they control. An important component of end-to-end delay is that due to forwarding elements, the fundamental building block of which is the delay incurred when a packet passes through a single IP router.

**2.3PROPOSED SYSTEM:**

The motivation for the present work is a detailed knowledge and understanding of such “through-router” delays. A thorough examination of delay leads inevitably to deeper questions about congestion, and router queueing dynamics in general. We provide a comprehensive examination of these issues from three points of view: the understanding of origins, measurement, and reporting, all grounded in a unique data set taken from a router in the access network of a Tier-1 provider.

The first aim of this paper is a simple one, to exploit this unique data set by reporting in detail on the magnitudes, and also the temporal structure, of delays on high capacity links with nontrivial congestion. The result is one of the most comprehensive pictures of router delay performance that we are aware of. As our analysis is based on empirical results, it is not reliant on assumptions on traffic statistics or router operations.

Our second aim is to use the completeness of the data as a tool to investigate how packet delays occur inside the router. In other words, we aim to provide a physical model capable of explaining the observed delay and congestion. Working in the context of the popular store and forward router architecture, we are able to justify the commonly held assumption that the bottleneck of such an architecture is in the output buffers, and thereby validate the fluid output queue model relied on routinely in the field of active probing. We go further to define a refined model with an accuracy close to the limits of time stamping precision, which is robust to many details of the architecture under reasonable loads.

Packet delays and congestion are fundamentally linked, as the former occur precisely because periods of temporary resource starvation, or *micro congestion episodes*, are dealt with via buffering. Our third contribution is an investigation of the origins of such episodes, driven by the question, “What is the dominant mechanism responsible for delays?” We use a powerful methodology of virtual or “semi-” experiments, that exploits both the availability of the detailed packet data, and the fidelity of the router model. We identify, and evaluate the contributions of, three known canonical mechanisms: i) reduction in link bandwidth from core to access; ii) multiplexing of multiple input streams; iii) burstiness of the input traffic stream(s).

**2.4 MODULES:**

* **CLIENT**
* **SERVER**

**MODULE DESCRIPTION:**

**SERVER:**

Server is main module in this project, server receive the data sent by client & calculate the congestion parameters for the client like delay, bandwidth, busy & status of client.

**CLIENT:**

Client sends the data to server & if client is free at server, data is successfully sent .Otherwise the data is in queue until previous data is send.

**3. SOFTWARE DESIGN**

**3.1 INTRODUCTION**:

Software design sits at the technical kernel of the software engineering process and is applied regardless of the development paradigm and area of application. Design is the first step in the development phase for any engineered product or system. The designer’s goal is to produce a model or representation of an entity that will later be built. Beginning, once system requirement have been specified and analyzed, system design is the first of the three technical activities -design, code and test that is required to build and verify software.

The importance can be stated with a single word “Quality”. Design is the place where quality is fostered in software development. Design provides us with representations of software that can assess for quality. Design is the only way that we can accurately translate a customer’s view into a finished software product or system. Software design serves as a foundation for all the software engineering steps that follow. Without a strong design we risk building an unstable system – one that will be difficult to test, one whose quality cannot be assessed until the last stage.

During design, progressive refinement of data structure, program structure, and procedural details are developed reviewed and documented. System design can be viewed from either technical or project management perspective. From the technical point of view, design is comprised of four activities architectural design, data structure design, interface design and procedural design.

**3.2 DATA FLOW DIAGRAMS:**

A graphical tool used to describe and analyze the moment of data through a system manual or automated including the process, stores of data, and delays in the system. Data Flow Diagrams are the central tool and the basis from which other components are developed. The transformation of data from input to output, through processes, may be described logically and independently of the physical components associated with the system. The DFD is also know as a data flow graph or a bubble chart.

**TYPES OF DATA FLOW DIAGRAMS**

DFDs are two types

* **PHYSICAL DFD**
* **LOGICAL DFD**

**PHYSICAL DFD**

Structured analysis states that the current system should be first understand correctly. The physical DFD is the model of the current system and is used to ensure that the current system has been clearly understood. Physical DFDs shows actual devices, departments, people etc., involved in the current system

**LOGICAL DFD**

Logical DFDs are the model of the proposed system. They clearly should show the requirements on which the new system should be built. Later during design activity this is taken as the basis for drawing the system’s structure charts.

**The Basic Notation used to create a DFD’s are as follows:**

**Dataflow:**Data move in a specific direction from an origin to a

Destination.

**Process:**People, procedures, or devices that use or produce

(Transform) Data. The physical component is not identified.

**Source:**External sources or destination of data, which may be

People, programs, organizations or other entities.

**Data Store:**Here data are stored or referenced by a process in the System

**Data flow diagram**

Client

Network

Router

**3.3 UML DIAGRAMS:**

The Unified Modeling Language allows the software engineer to express an analysis model using the modeling notation that is governed by a set of syntactic semantic and pragmatic rules.

**Introduction to UML:**

The Unified Modelling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct and document the artifacts of a software-intensive system.

A UML system is represented using five different views that describe the system from distinctly different perspective. Each view is defined by a set of diagram, which is as follows.

* + **User Model View**

1. This view represents the system from the users perspective.
2. The analysis representation describes a usage scenario from the end-users perspective.
   * **Structural model view**
3. In this model the data and functionality are arrived from inside the system.
4. This model view models the static structures.
   * **Behavioral Model View**

It represents the dynamic of behavioral as parts of the system, depicting the interactions of collection between various structural elements described in the user model and structural model view.

* + **Implementation Model View**

In this the structural and behavioral as parts of the system are represented as they are to be built.

* **Environmental Model View**

In this the structural and behavioral aspects of the environment in which the system is to be implemented are represented.

 UML is specifically constructed through two different domains they are:

* + - UML Analysis modeling, this focuses on the user model and structural model views of the system.
    - UML design modeling, which focuses on the behavioral modeling, implementation modeling and environmental model views.

Use case Diagrams represent the functionality of the system from a user’s point of view. Use cases are used during requirements elicitation and analysis to represent the functionality of the system. Use cases focus on the behavior of the system from external point of view.

**3.4 USE CASE DIAGRAM:**

**USE CASE:**

A Use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Also use case diagrams can be applied to illustrate the static use case view of a system. They are especially important in organizing and modeling the behaviors of a system.

**ACTOR:**

An actor represents a coherent set of roles that users of a system play when interacting with the use cases of the system. An actor participates in use cases to accomplish an overall purpose. An actor can represent the role of a human, a device, or any other systems.

**Use Case Diagram**:

****

**3.5 SEQUENCE DIAGRAM:**

A Sequence diagram is an interaction diagram that emphasizes the time orderingof messages. A Sequence diagram shows a set of objects and the messages sent andreceived by those objects.

The objects are typically named or anonymous instances of classes, but may also represent instances of other things such as collaboration, components and nodes. These are used to illustrate the dynamic view of the system. It is isomorphic, which means taking one of them.

****

**3.6 CLASS DIAGRAM:**

A class diagram is used to show the existence of the classes and their relationship in the logical design of the system. The icon we use to represent a class in a class diagram has a shape that of an amorphous blob also called a cloud. This icon represents an abstraction with some crispy defined boundaries. The dashed lines that form the outline of this icon indicate that clients generally operate only upon instances of a class, but not the class itself. The name of the class is required and is replaces inside the blob.

**Class Diagram:**



**3.7 COLLABORATION DIAGRAM:**

A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

**Collaboration Diagram:**



**3.8 ACTIVITY DIAGRAM:**

The activities that occur within a use case or within an objects behavior typically occur in a sequence .an activity diagram is designed to be simplified look at what happens during an operations or a process.

Each activity is represented by a rounded rectangle the processing within an activity goes to compilation and then an automatic transmission to the next activity occurs. An arrow represents the transition from one activity to the next. The activity diagram has a starting point represented by a filled in circle, and an endpoint represented by a bull’s eye. An activity diagram describes a system in terms of activities. Activities are the state that represents the execution of a set of operations. These are similar to flow chart diagram and dataflow.

**Activity diagram for client: Activity diagram for server:**

****

**3.9 STATE CHART DIAGRAM:**

At any given time, on object is in a particular state. The Uml state diagram captures    this bit of reality. The symbol at the top of the figure represents the figure represents the start state and symbol at the bottom represents the end state. State chart diagrams describe the behavior of an individual object as a number of states and transitions between these states. a state chart represents a particular set of values for an object. The sequence diagram focuses on the messages exchanged between objects, the state chart diagrams focuses on the transition between states.



**4.SYSTEM DEVELOPMENT ENVIRONMENT**

**ABOUT JAVA:**

**Java** is a programming language originally developed by Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any supported hardware/operating-system platform. One should be able to write a program once, compile it once, and run it anywhere.

This is achieved by compiling the Java language code, not to machine code but to Java bytecode – instructions analogous to machine code but intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a JRE installed on their own machine, or in a Web browser.

Standardized libraries provide a generic way to access host specific features such as graphics, threading and networking. In some JVM versions, bytecode can be compiled to native code, either before or during program execution, resulting in faster execution.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would, and Java suffered a reputation for poor performance. This gap has been narrowed by a number of optimization techniques introduced in the more recent JVM implementations.

One such technique, known as (just-in-time compilation) JIT, translates Java bytecode into native code the first time that code is executed, then caches it. This result in a program that starts and executes faster than pure interpreted code can, at the cost of introducing occasional compilation overhead during execution. More sophisticated VMs also use dynamic recompilation, in which the VM analyzes the behavior of the running program and selectively recompiles and optimizes parts of the program. Dynamic recompilation can achieve optimizations superior to static compilation because the dynamic compiler can base optimizations on knowledge about the runtime environment and the set of loaded classes, and can identify *hot spots* - parts of the program, often inner loops, that take up the most execution time. JIT compilation and dynamic recompilation allow Java programs to approach the speed of native code without losing portability.

Another technique, commonly known as *static compilation*, or ahead-of-time (AOT) compilation, is to compile directly into native code like a more traditional compiler. Static Java compilers translate the Java source or bytecode to native object code. This achieves good performance compared to interpretation, at the expense of portability; the output of these compilers can only be run on a single architecture. AOT could give Java something like performance, yet it is still not portable since there are no compiler directives, and all the pointers are indirect with no way to micro manage garbage collection.

Java's performance has improved substantially since the early versions, and performance of JIT compilers relative to native compilers has in some tests been shown to be quite similar. The performance of the compilers does not necessarily indicate the performance of the compiled code; only careful testing can reveal the true performance issues in any system.

One of the unique advantages of the concept of a runtime engine is that errors (exceptions) should not 'crash' the system. Moreover, in runtime engine environments such as Java there exist tools that attach to the runtime engine and every time that an exception of interest occurs they record debugging information that existed in memory at the time the exception was thrown (stack and heap values). These Automated Exception Handling tools provide 'root-cause' information for exceptions in Java programs that run in production, testing or development environments.

**4.1 Features of java**:

* **Distributed**

Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP Java applications can open and access across the Net via URLs with the same ease as when accessing local file system.

We have found the networking capabilities of Java to be both strong and easy to use. Anyone who has tries to do Internet programming using another language will revel. How simple Java makes onerous tasks will like opening a socket connection.

1. **Robust**

Java is intended for writing programs that must be readable in a Variety ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic checking, and eliminating situations that are error prone. The single biggest difference between Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.

The Java compiler detects many problems that in other languages would only show up at runtime. As for the second point, anyone who has spent hours chasing a memory leak cost by a printer bug will be very happy with this feature of Java.

Java gives you the best of both worlds. You need not pointers for everyday constructs like string and arrays. You have the power of pointers if you need it, for example, for like lists. And you have always-complete safety, since you can never access a bad pointer or make memory allocation errors.

* **Secure**

Java is intended to be used in networked/distributed environment toward that end; a lot of emphasis has been placed on security. Java enables the contraction of virus-free, temper-free systems.

Here is a sample of what Java’s security features are supposed to keep a Java programming from doing:

1. Overrunning the runtime stack.

2. Corrupting memory outside its own process space.

3. Reading or writing local files when invoked through a security- Conscious class loaders like Web browser.

* **Architecture Neutral**

The compiler generates an architecture neutral object file format- the compiled code is executable on many processors, given the presence of Java runtime system...The Java compiler does this by generating byte code instructions which have nothing to do with a particularcomputer architecture. Rather they ere designed to be both easy to any machine and easily translated into native machine code on the fly.

Twenty years ago, the UCSD Pascal system did the same thing in a commercial product and, even before that, Nicholas Worth’s original implementation of Pascal used the same approach. By using bytecode, performance takes major hit. The designers of Java did an excellent job developing a byte code instruction set those workers well on today’s most common computer architectures. And the codes have been designed to translate easily into actual machine instructions.

* **Portable**

Unlike C and C++, they are no "implementation dependent" aspects of the specifications. The sizes of the primitive’s data types are specified, as is the behavior of arithmetic on them.

For example, an int in Java is always a 32-bit integer. In C/C++, int can mean a 16-bit integer, a 32-bit integer, or any size the compiler vendor likes. The only restriction is that it must have at least as many bytes int and cannot have more bytes than a long int.

The libraries that are a part of the system define portable interfaces. For example, there is an abstract window class and implementations of it UNIX, Windows, and the Macintosh.

* **Interpreted**

The Java interpreters can execute Java byte codes directly on any machine to which the interpreter has been ported. Since linking is a more incremental and lightweight process, the development process can be much more rapid and explanatory.

One problem is that the JDK is fairly slow at compiling your source code to the bytecode that will, ultimately, be interpreted in the current version.

* **High Performance**

While the performance of interpreted bytecode is usually more than

adequate, there are situations higher performance is required. The bytecode can be translated on fly into machine code for the particular CPU the application is running on.

Native code compilers for Java are not yet generally available. Instead there are just-in-time (JIT) compilers. These work by compiling the byte codes into native code once, caching the results, and then calling them again, if needed. This speeds up code once, catching the results, and calling them again, if needed. This speed up the loop tremendously since once has to do the interpretation only once. Although still slightly slower than a true native code compiler, just-in-time compilers can give you a 10-or even 20-fold speedup for some programs and will almost always be significantly faster than the Java Interpreter.

**⚫ Multithreaded**

In a number of ways, Java is more dynamic language than C or C++. It was designed to adapt to an evolving environment. Libraries can freely add new methods and instance variables without any effect on their clients.... In Java, finding out run time type information is straightforward.

This is an important feature in those situations where code needs to be added to a running program. A prime example is code that is downloaded from the Internet to run in browser.

**4.2 SWINGS:**

Swing is a platform-independent, *Model-View-Controller* GUI framework for Java. It follows a single-threaded programming model, and possesses the following traits:

**Platform independence**

Swing is platform independent both in terms of its expression (Java) and its implementation (non-native universal rendering of widgets).

**Extensibility**

Swing is a highly partitioned architecture, which allows for the "plugging" of various custom implementations of specified framework interfaces: Users can provide their own custom implementation(s) of these components to override the default implementations. In general, Swing users can extend the framework by extending existing (framework) classes and/or providing alternative implementations of core components.

**Component-oriented**

Swing is a component-based framework. The distinction between objects and components is a fairly subtle point: concisely, a component is a well-behaved object with a known/specified characteristic pattern of behaviour. Swing objects asynchronously fire events, have "bound" properties, and respond to a well-known set of commands (specific to the component.) Specifically, Swing components are Java Beans components, compliant with the Java Beans Component Architecture specifications.

**Customizable**

Given the programmatic rendering model of the Swing framework, fine control over the details of rendering of a component is possible in Swing. As a general pattern, the visual representation of a Swing component is a composition of a standard set of elements, such as a "border", "inset", decorations, etc. Typically, users will programmatically customize a standard Swing component (such as a JTable) by assigning specific Borders, Colors, Backgrounds, opacities, etc., as the properties of that component. The core component will then use these property (settings) to determine the appropriate renderers to use in painting its various aspects. However, it is also completely possible to create unique GUI controls with highly customized visual representation.

**Configurable**

Swing's heavy reliance on runtime mechanisms and indirect composition patterns allows it to respond at runtime to fundamental changes in its settings. For example, a Swing-based application can change its look and feel at runtime. Further, users can provide their own look and feel implementation, which allows for uniform changes in the look and feel of existing Swing applications without any programmatic change to the application code.

**Lightweight UI**

Swing's configurability is a result of a choice not to use the native host OS's GUI controls for displaying itself. Swing "paints" its controls programmatically through the use of Java 2D APIs, rather than calling into a native user interface toolkit. Thus, a Swing component does not have a corresponding native OS GUI component, and is free to render itself in any way that is possible with the underlying graphics APIs.

However, at its core every Swing component relies on an AWT container, since (Swing's) JComponent extends (AWT's) Container. This allows Swing to plug into the host OS's GUI management framework, including the crucial device/screen mappings and user interactions, such as key presses or mouse movements. Swing simply "transposes" its own (OS agnostic) semantics over the underlying (OS specific) components. So, for example, every Swing component paints its rendition on the graphic device in response to a call to component.paint(), which is defined in (AWT) Container. But unlike AWT components, which delegated the painting to their OS-native "heavyweight" widget, Swing components are responsible for their own rendering.

This transposition and decoupling is not merely visual, and extends to Swing's management and application of its own OS-independent semantics for events fired within its component containment hierarchies. Generally speaking, the Swing Architecture delegates the task of mapping the various flavors of OS GUI semantics onto a simple, but generalized, pattern to the AWT container. Building on that generalized platform, it establishes its own rich and complex GUI semantics in the form of the JComponent model. A review of the source of Container.java and JComponent.java classes is recommended for further insights into the nature of the interface between Swing's lightweight components and AWT's heavyweight widgets.

**Loosely-Coupled/MVC**

The Swing library makes heavy use of the Model/View/Controller software design pattern, which conceptually decouples the data being viewed from the user interface controls through which it is viewed. Because of this, most Swing components have associated *models* (which are specified in terms of Java interfaces), and the programmer can use various default implementations or provide their own. The framework provides default implementations of model interfaces for all of its concrete components.

Typically, Swing component model objects are responsible for providing a concise interface defining events fired, and accessible properties for the (conceptual) data model for use by the associated JComponent. Given that the overall MVC pattern is a loosely-coupled collaborative object relationship pattern, the model provides the programmatic means for attaching event listeners to the data model object. Typically, these events are model centric (ex: a "row inserted" event in a table model) and are mapped by the JComponent specialization into a meaningful event for the GUI component.

For example, the JTable has a model called TableModel that describes an interface for how a table would access tabular data. A default implementation of this operates on a two-dimensional array.

The view component of a Swing JComponent is the object used to graphically "represent" the conceptual GUI control. A distinction of Swing, as a GUI framework, is in its reliance on programmatically-rendered GUI controls (as opposed to the use of the native host OS's GUI controls). This distinction is a source of complications when mixing AWT controls, which use native controls, with Swing controls in a GUI.

Finally, in terms of visual composition and management, Swing favors relative layouts (which specify the positional relationships between components) as opposed to absolute layouts (which specify the exact location and size of components). This bias towards "fluid"' visual ordering is due to its origins in the applet operating environment that framed the design and development of the original Java GUI toolkit. (Conceptually, this view of the layout management is quite similar to that which informs the rendering of HTML content in browsers, and addresses the same set of concerns that motivated the former.)

### Look and feel

Swing allows one to specialize the look and feel of widgets, by modifying the default (via runtime parameters), deriving from an existing one, by creating one from scratch, or, beginning with **J2SE 5.0**, by using the skinnable synth Look and Feel (see Synth Look and Feel), which is configured with an XML property file. The look and feel can be changed at runtime, and early demonstrations of Swing frequently provided a way to do this.

### Relationship to AWT

### Since early versions of Java, a portion of the Abstract Window Toolkit (AWT) has provided platform-independent APIs for user interface components. In AWT, each component is rendered and controlled by a native peer component specific to the underlying windowing system.

By contrast, Swing components are often described as *lightweight* because they do not require allocation of native resources in the operating system's windowing toolkit. The AWT components are referred to as *heavyweight components*.

Much of the Swing API is generally a complementary extension of the AWT rather than a direct replacement. In fact, every Swing lightweight interface ultimately exists within an AWT heavyweight component because all of the top-level components in Swing (JApplet, JDialog, JFrame, and JWindow) extend an AWT top-level container. However, the use of both lightweight and heavyweight components within the same window is generally discouraged due to Z-order incompatibilities.

The core rendering functionality used by Swing to draw its lightweight components is provided by Java 2D, another part of JFC.

## 4.3 NETWORKING:

### TCP/IP stack

The TCP/IP stack is shorter than the OSI one:



TCP is a connection-oriented protocol; UDP (User Datagram Protocol) is a connectionless protocol.

### IP datagram’s

The IP layer provides a connectionless and unreliable delivery system. It considers each datagram independently of the others. Any association between datagram must be supplied by the higher layers. The IP layer supplies a checksum that includes its own header. The header includes the source and destination addresses. The IP layer handles routing through an Internet. It is also responsible for breaking up large datagram into smaller ones for transmission and reassembling them at the other end.

### UDP

UDP is also connectionless and unreliable. What it adds to IP is a checksum for the contents of the datagram and port numbers. These are used to give a client/server model - see later.

### TCP

TCP supplies logic to give a reliable connection-oriented protocol above IP. It provides a virtual circuit that two processes can use to communicate.

### Internet addresses

In order to use a service, you must be able to find it. The Internet uses an address scheme for machines so that they can be located. The address is a 32 bit integer which gives the IP address. This encodes a network ID and more addressing. The network ID falls into various classes according to the size of the network address.

### Network address

Class A uses 8 bits for the network address with 24 bits left over for other addressing. Class B uses 16 bit network addressing. Class C uses 24 bit network addressing and class D uses all 32.

### Subnet address

Internally, the UNIX network is divided into sub networks. Building 11 is currently on one sub network and uses 10-bit addressing, allowing 1024 different hosts.

### Host address

8 bits are finally used for host addresses within our subnet. This places a limit of 256 machines that can be on the subnet.

### Total address



The 32 bit address is usually written as 4 integers separated by dots.

### Port addresses

A service exists on a host, and is identified by its port. This is a 16 bit number. To send a message to a server, you send it to the port for that service of the host that it is running on. This is not location transparency! Certain of these ports are "well known".

### Sockets

A socket is a data structure maintained by the system to handle network connections. A socket is created using the call socket. It returns an integer that is like a file descriptor. In fact, under Windows, this handle can be used with Read File and Write File functions.

#include <sys/types.h>

#include <sys/socket.h>

int socket(int family, int type, int protocol);

Here "family" will be AF\_INET for IP communications, protocol will be zero, and typewill depend on whether TCP or UDP is used. Two processes wishing to communicate over a network create a socket each. These are similar to two ends of a pipe - but the actual pipe does not yet exist.

**5. CONCLUSION**

 We discovered in detail router congestion and packet delay, based on a unique data set where all IP packets crossing a router were measured. We presented authoritative experimental results about packet delays in the code above, but also practical inside the router to deliver a physical model of router process which can very precisely infer, not just packet delay, but the entire structure of micro congestion (busy periods). We then used semi-experiments to inquire into the causes of micro congestion based on three canonical mechanisms: bandwidth reduction, link multiplexing, and burstiness.

We defined new corresponding metrics, and used them to show that, for this data, the classical nonlinear effect of the multiplexing of flows was primarily responsible for the observed delay behavior. Finally, we examined the difficult problem of compactly summarizing the richness of micro congestion behavior, beginning by explaining why an approach based on utilization is fundamentally flawed. By exploiting an observed commonality in shape of large busy periods, we were able to capture much of the busy period diversity using a simple joint distribution of busy period amplitude and duration. We showed how this metric captures important information about the delay process without any model based preconceptions, and gave a sketch for on-line algorithms for its collection and export.

**6. BIBLIOGRAPHY**

[1] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, F. Tobagi, and C. Diot, “Analysis of measured single-hop delay from an operational backbone network,” in *Proc. IEEE INFOCOM*, New York, Jun. 2002, pp. 535–544.

[2] Waikato Applied Network Dynamics. [Online]. Available: <http://www.wand.cs.waikato.ac.nz/wand/wits/>

[3] N. Hohn, D. Veitch, K. Papagiannaki, and C. Diot, “Bridging router performance and queueing theory,” in *Proc. ACM Sigmetrics*, New York, Jun. 12–16, 2004, pp. 355–366.

[4] K. Papagiannaki, D. Veitch, and N. Hohn, “Origins of microcongestion in an access router,” in *Proc. Passive and Active Measurement Workshop(PAM2004)*, Juan-Les-Pins, France, Apr. 19–20, 2004, Lecture Notes in Computer Science (LNCS), pp. 126–136, Springer.

[5] N. McKeown, “SLIP: A scheduling algorithm for input-queued

switches,” *IEEE/ACM Trans. Networking*, vol. 7, no. 2, pp. 188–201,Apr. 1999.

[6] W. Simpson, “PPP in HDLC-like framing,” RFC 1662 [Online]. Available: <http://www.ietf.org/rfc/rfc1662>

[7] Endace Measurement Systems.[Online]. Available: <http://www.endace.com/>

[8] S. Donnelly, “High precision timing in passive measurements of data n

**Sites Referred:**

<http://java.sun.com>

<http://www.sourcefordgde.com>

<http://www.networkcomputing.com/>