SYSTEM IMPLEMENTATION OF A REAL-TIME, CONTENT BASED APPLICATION ROUTER FOR A MANAGED PUBLISH-SUBSCRIBE SYSTEM

By

SUNIL MUTHUSWAMY

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2008

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of SUNIL MUTHUSWAMY find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

I would like to thank my advisor Dr Carl Hauser for his continued support and guidance which helped me stay focused on the research. I would also like to thank Dr David Bakken and Dr Jabulani Nyathi for their academic guidance and valuable inputs on my research. Further I would also like to thank all the members of the GridStat and my friends here at WSU for making my stay at WSU very enjoyable.

Finally, I would like to thank the organizations that provided the financial support for my education and research. In particular, I have received assistantship from WSU during my first semester. Further, my research has been supported by grant by National Science Foundation CNS 05-24695 (CT-CS: Trustworthy Cyber Infrastructure for the Power Grid(TCIP)).

SYSTEM IMPLEMENTATION OF A REAL-TIME, CONTENT BASED APPLICATION ROUTER FOR A MANAGED PUBLISH-SUBSCRIBE SYSTEM

Abstract

by Sunil Muthuswamy, M.S.
Washington State University
August 2008

Chair: Carl Hauser

Accurate and efficient monitoring of the electric power grid (EPG) requires that the status information from the substation and the control commands from the control center, reaches the control center and the substation respectively in real-time and secured. The latency of this secured communication flow between the substation and the control center should be low and contained. GridStat is a distributed publish-subscribe middleware framework that tries to address these current and the future quality-of-service (QoS) requirements of the communication infrastructure of the EPG. GridStat's flexible architecture allows a wider range of control applications to make use of the wealth of data available from the substation, securely and in real-time. The various components within GridStat contribute to the different QoS attributes. The component within GridStat that is primarily responsible for containing the end-to-end latency is the status router (SR). SR is an application router which routes events from the publisher to the interested subscribers at the requested rate. The current prototype of the SR in Java limits its real-time capabilities because of the lack of application control over the execution of the garbage collector within Java.

This thesis explores the various real-time mechanisms that are available and required to implement a real-time content based application router that can meet the QoS requirements of the EPG. It implements the SR in C/C++ programming language with the use of real-time features provided by the operating system (OS) such as priority scheduling, real-time threads and memory locking.

It also explores the various real-time extensions/patches that enhances the real-time capabilities of a general purpose OS such as Linux.

Such an implementation alone will not be able to meet the QoS requirements under all operating environments and conditions. Network congestion and packet losses will have a detrimental effect on the QoS. To be able to satisfy the QoS requirements under such conditions, there is a need of a scheduling algorithm that is capable of achieving end-to-end delay bounds. This thesis explores the effects of network protocol on the placement of the scheduling algorithm and also provides a system implementation of the Delay-Earliest Due Date scheduling algorithm as a queuing discipline within the kernel.

TABLE OF CONTENTS

Page

APPENDIX

LIST OF TABLES

Page

LIST OF FIGURES

Page

Dedication

This thesis is dedicated to my family and friends

CHAPTER ONE

INTRODUCTION

The electric power grid (EPG) is a large interconnection of both generation and transmission electrical subsystems to facilitate distribution of electricity to consumers and businesses. Its demand, requirement and usage makes it one of the most critical infrastructures of any nation. The integrity and reliability of its operations are therefore of prime importance, which can only be achieved by secured, real-time monitoring and control of the system. This thesis explores the different mechanisms available and required to implement a real-time application router that will enable wide-area distributed systems to satisfy the real-time control needs of the EPG.

Accurate control requires real-time data gathering, transmission, interpretation and correction. The real-time[1] dissemination of the status information from the various measuring devices at the substation like the Phasor Measurement Unit (PMU) to the control center is essential for maintaining the balance between the supply and demand of electricity through the EPG. Any control decision can be fatal if the integrity and the availability of the control information is compromised. An occasional delay in receiving information (stale data) at the control center is equally fatal as misaligned data because it represents a wrong state of the system and can lead to incorrect decision making. So the real-time requirements for the EPG not only include low average latency but also a tight upper bound on the latency to prevent occasional delays. Also, the data from any substation should be available to any control center in need, within the demanded time-frame. The present control transmission infrastructure of the US power grid provide little flexibility to adapt to these emerging communication needs. To cater to such quality of service (QoS) requirements of the EPG, a middleware framework named GridStat has been proposed in [14, 31, 30, 29], which achieves real-time, reliable and secure delivery of information. GridStat is a distributed system with different components contributing to the various QoS attributes. The publish-subscribe

[1] In this context, real-time is the ability to provide predictable and consistent throughput and latency.

model of GridStat achieves the requested QoS with the help of mechanisms such as multicasting, redundant paths, resource reservation and admission control. It also hides the heterogeneity of the diverse network from the end-users, making it deployable over a Wide Area Networks (WANs). The component within GridStat which focuses on satisfying the admitted end-to-end latency is the *status router* (SR). The purpose of the network of SR's is to provide a message bus between publishers and subscribers that is specialized for forwarding streams of periodically generated updated messages [30]. This thesis provides an implementation of the SR based on the real-time hooks provided by the operating system (OS), so that the SR meets its requirements.

Integrated Services Packet Network (ISPN)[20] identifies the components of any architecture that is required to support real-time traffic in a packet switched network such as Internet. The paper argues that all real-time applications do not require a fixed delay bound and some of them can adapt to some extent to the current networking conditions. The paper proposes four key components to any architecture required to support such real-time applications. The first component is nature of the commitment made by the network when it admits a service. Before making any commitment to any client the network should have a rough idea of the traffic that will be sent by that client. These commitments can be classified as guaranteed and predicted. In guaranteed service, if the client adheres to its traffic characterization then the service commitments should be met under all circumstances. The predicted service offers a more flexible commitment in which the client and network can adjust to the variation in the client's traffic. The second piece is the service interface comprising of the set of parameters passed between the source and the network. This interface characterizes the QoS the network will deliver and the source traffic. This information is necessary for resource allocation and admission control. The third component is the packet scheduling algorithm that should be used by the network switches to meet the service commitment. It also details out the scheduling information that must be carried in the packet headers. The final component is the algorithm used for admission control. It lists down the conditions under which a new service request should be accepted or denied. It also proposes that a portion of the total bandwidth should

be reserved for non real-time traffic.

All these components form a part of the GridStat framework at different levels. The SR concentrates at the scheduling algorithm aspect, its details and location within the system.

1.1 Motivation

Use of standard and implementation-independent protocols like CORBA (Common Object Request Broker Architecture) and UDP (User Datagram Protocol) in GridStat provides the flexibility of choosing the most suitable implementation language for the various components. The existing prototype of GridStat is in Java. Java takes the burden of judicially managing the memory off the user by repeatedly checking for unused memory references and collecting them using a *garbage collector*. The lack of end-user control over the execution of the garbage collector induces unpredictable and potentially long pauses in the run-time behavior of the SR, denying real-time status dissemination. C[2] implementations not only have better predictable run-time behaviors but also authorize fine-grain access to the system resources and, hence can cater to the real-time QoS needs of SR, thereby making it a more reasonable choice as the implementation language for SR.

The Java prototype of the SR in GridStat limits its real-time capabilities. Java applications inability to control the execution of the garbage collector makes it unsuitable for mission-critical real-time applications. The Java Virtual Machine (JVM) also prevents the generation and use of the most optimized binary for an application on a particular platform. These factors inhibits a Java applications capability to provide tight bounded latencies. This thesis explores alternate implementations, mechanisms and techniques to implement the SR and the complementing real-time support from the OS so that the SR can meet its requirements, while maintaining the SR's interoperability with other components within GridStat.

UDP has been widely accepted as the Internet protocol for multicast. The unreliable protocol of UDP working on top of the best effort services of the Internet often leads to cases of network

[2]refers to C/C++ systems throughout the thesis

congestion and packet loss. In order to meet the QoS requirements under such operating environments, a scheduling algorithm has to be in place. Scheduling algorithms like Delay-Earliest Due Date (Delay-EDD) are capable of providing tight-bounded end-to-end latencies under these circumstances. Because of the absence of a feedback loop from the UDP protocol stack to the application, any such packet scheduling algorithm has to be carefully placed in the system. This thesis proposes a system implementation of the Delay-EDD scheduling algorithm within the OS for optimal usage and performance.

1.2 Challenges

Different implementation languages have varying characteristics with respect to performance, portability, inherent security, access to system resources and management. The choice of an implementation language for an application has to be made by prioritizing the requirements and then grading the various languages based on these priorities. These fundamental challenges which lie on the implementation boundaries have to be overcome in order to successfully design, develop, evaluate and compare alternate implementations for SR. The answers for the questions below will help evaluate these differences.

1. What is the trade-offs between a SR implemented in Java and that in C in terms of security, reliability, ease of use and management?

   Systems built using Java are easier to port and manage because of the virtualization provided by the JVM (Java Virtual Machine). Conversely, systems built in C are comparatively difficult to port because of the OS specific libraries and API's (Application Programmers Interface). Security and reliability of any system largely relies on the implementation, though systems built in some implementation languages are inherently more prone to attacks than others. For example, Java keeps a track of all the application memory and reclaims any unused/unreferenced memory using the garbage collector. C/C++ applications have to explicitly allocate the required memory and deallocate it when the memory is not needed any

more. This exposes any fault in allocation/deallocation to security threats leading from the use of uninitialized memory and incorrect memory copying. It also makes C/C++ applications more vulnerable to system crashes and memory leak. Another important difference between Java and C/C++ systems is that the Java runtime provides boundary checks and raises exceptions that can be caught by the application. The C/C++ runtime does not provide any such additional checks and are prone to crashes caused by memory exceptions.

2. How to secure the SR to prevent external attacks exploiting buffer overflows, malformed packets and even to a certain extent *distributed denial of service* (DDoS)?

   Memory can be micro managed in C. This though gives C an upper hand over other languages for building scalable solutions, can also lead to buffer overflow errors if mismanaged. While it is acceptable to make assumptions about the application packet formats, cases of misaligned packets should be gracefully handled and should not lead to system failures.

   DDoS is a kind of an external attack that can paralyze any unprotected network system. It operates by flooding the system with enormous number of requests, making the system irresponsive to legitimate requests. Since the SR primarily deals with network packets, measures and actions based on firewalls and linux QoS should be taken to protect the SR from potential DDoS attacks.

3. How to meet the desired QoS requirements under all operating environments and loads?

   A lot of the devices in the power world are legacy systems and are computationally underpowered. Also in cases of emergency, the load on the communication systems can increase considerably. The SR should be designed to be able to adapt and perform even under such scenarios.

1.3 Research Contribution

The research contribution of this thesis are:

- Design and implementation of a real-time, content based application router which can cater to the real-time needs of communication infrastructure of the EPG.
- Exploring and exploiting the different types of real-time support present in the OS which can aid the application router in its real-time performance.
- Design and implementation of Delay-EDD scheduling algorithm as a kernel module to achieve the desired QoS on devices with slow network interface.
- Experimental real-time performance evaluation of the router under different load conditions.

1.4 Thesis Organization

This thesis is divided into six chapters. Chapter 1 introduces the problem space, the motivation and the impeding challenges. Chapter 2 presents other work that has been done in related field. Chapter 3 describes the problem in detail with its various requirements and how the problem can be applied to other areas. Chapter 4 divides the solution into different high-level categories based on its focus area and discusses each of them in detail. The real-time performance evaluation of the SR is done in Chapter 5. Based on the results from Chapter 5, a conclusion is presented along with future work in Chapter 6.

CHAPTER TWO

RELATED WORK

Related work can be broadly divided into two sections. The first section looks at the work that is similar to the work done in this thesis. This includes research work done on achieving end-to-end QoS and the various content based routing algorithms. The second section looks at the various QoS implementations within the Linux kernel that have been a guiding force behind the design of the SR.

2.1 Similar work

2.1.1 End-to-End QoS

Impact of upper layer adaptation on end-to-end delay management in wireless ad hoc networks [32]. This research work looks at the impact of upper layer (application and middleware layer) adaptations on end-to-end delay management for wireless ad hoc networks. Adaptation in this context is the ability of an application to dynamically change its requirements based on the underlying network conditions. The ability to adapt gives the application a better chance to thrive under dynamic network conditions such as changing network bandwidth. The adaptation happens at two levels. The application layer adaptor helps the application to tune its requirements based on end-to-end QoS measurements. The lower layer middleware adaptor constantly takes feedback from the network about its conditions and adjusts the service class of the application traffic. Changing the service class benefits from the service differentiation provided by the network layer. The factors which differentiate this work from the work done in this thesis is the underlying network assumptions and their behavior. The paper mainly focuses on achieving end-to-end QoS on ad hoc networks where bandwidth guarantees are not as consistent as in wide area networks. This thesis looks at ways to enable wide area distributed systems to achieve end-to-end QoS.

Achieving end-to-end delay bounds in a real-time status dissemination network [33]. This

work looks at scheduling algorithms capable of achieving end-to-end delay bounds for wide area systems. Scheduling algorithm form an important part of a solution delivering QoS over communication networks. This research work compares the various scheduling algorithms such as FIFO (First In First Out), RED (Random Early Detection), Delay-EDD and others based on their characteristics and capabilities in providing end-to-end delay bounds. Based on analysis and research, it concludes that Delay-EDD scheduling algorithm is capable of bounding latency to the requirement levels. It supports this conclusion by implementing Delay-EDD scheduling algorithm within GridStat and through performance evaluation. The difference between this work and this thesis is that this thesis also looks at how the network semantics affect the placement of the scheduling algorithm for optimal performance and better efficiency.

Bandwidth sensitive routing in DiffServ networks with heterogeneous bandwidth requirements [48]. Differentiated Service (DiffServ) was proposed to provide better end-to-end QoS for applications as it is capable of differentiating between traffic with different priorities. Because DiffServ scheme is not coupled with the IP routing, traffic routed by IP routers and DiffServ may flow along the same paths. This can lead to inter-class effects wherein the priority traffic has deteriorating effects on best-effort IP traffic in terms of increase in packet loss and bandwidth starvation. The work done in [48] looks at bandwidth sensitive routing schemes for DiffServ networks so that the relative congestion in all links is minimized. Though this work looks at routing schemes, its aim is to better equip the DiffServ networks in providing end-to-end QoS with minimum side-effects on other traffic. The domain of this work does is different from the work done in this thesis as it looks at routing schemes in the network protocol stack while this thesis looks at real-time mechanisms required to provide end-to-end QoS.

QoS-aware resource management for distributed multimedia applications [41]. The constraints laid down by this work on end-to-end guarantees for multimedia applications is that QoS should be achievable on a general-purpose platform and should be application-controllable. To satisfy these constraints, a QoS mediator is required which supports *QoS negotiation, admission and reservation*

in an application accessible manner. The paper proposes a loadable middleware called QualMan capable of providing these QoS negotiations and capabilities. QualMan is a potential candidate for the QoS negotiations which happen within GridStat. Even though the paper looks at mechanisms to achieve end-to-end QoS, it differs from the work done in this thesis as it explores mechanisms that are required in the middleware layer rather than on the application/kernel space.

Achieving proportional delay differentiation in wireless LAN via cross-layer scheduling [52]. Service differentiation and QoS in wireline networks are well researched and documented. The same does not hold true for the wireless networks, though some research have gone into the design of differentiated media access for the wireless LANs (Local Area Network). This paper claims that such approaches cannot completely address some QoS metrics such as queuing delays. It proposes a joint packet scheduling at the network layer and distributed coordination at the media access control (MAC) layer to achieve delay differentiation in the wireless LAN. It implements waiting time priority (WTP) scheduler at the network layer to prioritize and schedule all the packets coming out of a network interface. SR derives its inspiration to have a packet scheduler at the network layer to achieve end-to-end delay bounds from this work. Details can be found in Section 4.4.

2.1.2 Content-Based Routing

The communication subsystem of a publish-subscribe service consists of event producers and consumers. An event or a packet which has been typically looked at as a black box can also be seen as a set of well defined attribute value pairs. Such a view of an event allows the events to be filtered based on the values of these different attributes, also called as content based filtering. Content based filtering/routing has increasingly been used in publish-subscribe systems because of the similarity in their communication model. In publish-subscribe systems, the subscribers express their interest in events or a pattern of events based on a value of an attribute within the event. This gives the subscribers the flexibility to choose events on multiple dimensions. The communication model used within GridStat is that of a content based filtering at the SR. Related work has been done in

this area in terms of providing efficient event routing algorithms and implementation, matching events and filtering algorithms for content based subscription systems.

Filtering algorithms and implementations for very fast publish/subscribe systems [28]. This work looks at the broader scope and application domain of publish-subscribe systems. It notes that any tool that can manage the high load of subscriptions and events on a wide area system should be scalable and efficient. Amongst the contribution from the work is that it provides data structures that are suitable and understand the subscription language. Also the implementation of the algorithms uses specialized instructions such as the processor *prefetch* command that is capable of avoiding cache misses and thereby improving the efficiency. This benefits from the argument that the same latency overheads that lie in the boundary of main-memory and secondary device also applies to the boundary between the main-memory and processor cache, though at a smaller scale. Such a technique will benefit an algorithm which deals with large amounts of data. In GridStat and in particularly the SR such high scale data movements are not common place, but the effects of such a technique on the SR can be studied and evaluated.

Efficient event routing in content-based publish-subscribe service networks [18]. This work looks at the event routing strategies for content based publish-subscribe systems. It classifies the content based routing space into *filter-based* and *multicast-based* approach. In the filter-based approach, the filters is applied on each incoming packet on intermediate routers between the publisher and the subscriber. In multicast-based approach, high-quality multicast groups that primarily match users interests are pre-constructed and users can join groups with matching interests. The paper introduces a hybrid solution named Kyra that can benefit from the advantages of both these approaches to achieve better routing efficiency.

Matching events in a content-based subscription system [12]. In a content based system, deciding which subscription criteria matches an event is a time consuming task. The naive approach of matching each subscription for each incoming event has a run-time complexity proportional to the number of subscriptions. For applications in which the number of subscriptions are high, such

an approach can easily lead to added latencies. When events are received at a fast rate, then the matching algorithm should be capable of scaling to the incoming rate of events, failing which there can be event losses and packet queuing leading to latency overheads. The algorithm proposed in the paper is expected to have a sub-linear time complexity. This work is related to the linear rate filtering algorithm within the SR, which delivers the subscribers events at the desired subscription interval. The subscription model of the EPG may not have the need for a very high number of subscribers, but if such a need arises, the application of such sub-linear rate filtering algorithms within the SR can be explored and evaluated.

2.2 QoS in Linux kernel

2.2.1 BSD Packet Filter (BPF)

The BPF [40] minimizes the copying of a packet across the kernel/user-space boundary by filtering packets as early as possible. This conserves bandwidth as precious CPU cycles are not spent on processing unwanted packets. BPF is a kernel module which attaches itself to the network device driver. All the incoming packets are delivered to the BPF. A user configured filter then decides whether the packet should be accepted or not. If the application requires only a small subset of the incoming packets then, a significant performance gain is realized by filtering out unwanted packets in the interrupt context. The filter is conveyed to the BPF from the user application using a custom pseudo language. These filters can be based on any field in a particular offset within the packet.

The BPF can complement the functioning of the SR by filtering unwanted packets early on. Expressing the rate filtering algorithm of the SR in terms of the pseudo language of the BPF may not be an easy task, but it offers great benefits.

2.2.2 Netfilter

Netfilter is a set of hooks inside the Linux kernel that allows kernel modules to register callback functions with the network stack. A registered callback function is then called back for every

packet that traverses the respective hook within the network stack [1]. Netfilter can be used to build sophisticated QoS and policy routers and hence, provide another interesting integration point for the SR and the Linux kernel. Netfilter can also be used to build internet firewalls.

2.2.3 Traffic Control

Traffic Control (tc) consists of *shaping, scheduling, policing and dropping* network traffic. These are achieved using qdiscs (Queuing Disciplines). qdisc is a packet queue associated with an algorithm that decides when to send which packet. IP packet handling using qdisc is performed by classifying and enqueuing network packets into incoming and outgoing queues. This classification can be done based on packet header fields (source or destination IP Addresses, port numbers, etc) and payload. *Netem* is one such network emulator in the linux kernel that reproduces network dynamics by delaying, dropping, duplicating or corrupting packets [38].

The Linux Socket Filter allows a user to send/receive raw packets directly to/from the network interface bypassing the usual protocol stack. This gives the user complete control over the Ethernet header of the packet. It also gives the user access to packets which are destined to a different host. [27]

CHAPTER THREE

PROBLEM

3.1 Statement

Formalize a content based event *forwarding* system which can meet the *differential* QoS requirements of the end users by maintaining a *low average delay, and a tight upper bounded latency*.

3.2 Background

One of the main goal as mentioned in the roadmap to secure control systems in the energy sector[6] is that *Within 10 years, the power sector will help ensure that energy asset owners have the ability and commitment to perform fully automated security state monitoring of their control system networks with* real-time *remediation capability.*

Judicious monitoring and control of the EPG requires that the status information from the various measuring and sensor devices like the Intelligent Electronic Devices (IED) located in the substation reaches the remote control center uncompromised and in due time. The same network which carries the status data[1] is also responsible for securely delivering the control commands from the monitoring systems to the substation devices. The latency (time delay) requirements of the information flowing through the communication network varies with the kind of status data requested and therefore demands differential QoS. As mentioned in one of the research paper on QoS in next generation communication for control systems [31], *The latency requirements are perhaps more stringent for control commands because while the control center might be able to substitute information derived from other sensors for missing inputs, it may be quite difficult to substitute different actions when a command is not carried out*. The magnitude of impact of a wrong decision based on compromised, overdue or partial information makes these attributes the prime quantifiable parameters to evaluate any status dissemination system. For example, the Northeast Blackout

[1] includes analog values from the various devices and not necessarily just ON/OFF value

of 2003 affected an estimated 40 million people in United States and costing billions of dollars in financial losses. One of the reasons attributed for the blackout as mentioned in a research paper on overcoming challenges in software for controlling power systems [16] is that there was lacking data about the state of the grid on a large scale. The control center operators were not able to formulate appropriate control interventions on the basis of the limited information available from purely local instrumentation. Another case study was done in [51] in which *162* cases of disturbances between 1979 to 1995 as reported by the North American Electric Reliability Council (NERC) were studied to determine the cause and impact of failures. It states, *The analysis of these disturbances clearly indicates that the problems in* real-time monitoring and operating control system(37.04%), communication system, and delayed restoration(38.27%) *contribute to a very high percentage of large failures*. One of the main bottlenecks identified by the paper amongst others was inadequate exchange of real-time operating information and real-time coordination among control centers

The hard latency requirement not only varies with the kind of information requested but also across the application spectrum which uses the data. [31] quantifies it as *Hard latency requirements range from a few milliseconds for protective relaying to a few tens of milliseconds for automated dynamic stabilization applications and a few seconds for conventional control center applications*. Achieving hard real time latency requires resource management, reservation and admission control (RSVP). Theoretically and practically it is best to segregate the RSVP logic from the main event forwarder (henceforth referred as *status router*), but control commands have to be in place to communicate information related to resource availability and allocation.

A pictorial overview of the present and the proposed communication infrastructure for the EPG is given in Figure 3.1 and Figure 3.2.

3.3 Problem Derivation

The problem has been motivated and derived based on the aspects and the requirements of the EPG. Some of these aspects and requirements mentioned below make the problem very peculiar

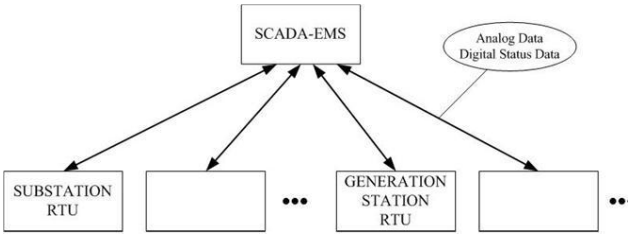

Figure 3.1: Existing Communication Infrastructure[46].

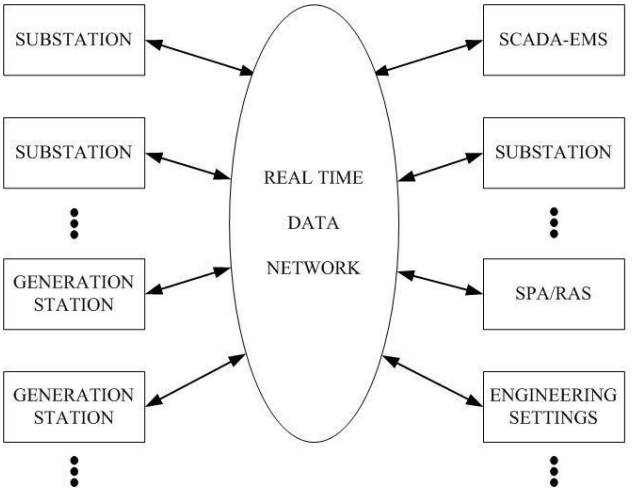

Figure 3.2: Conceptual Communication Network[46].

and particularly challenging.

3.3.1 Latency Requirements

Background:

The major performance-related requirements are latency, the delay that the communication system imposes, and rate, the number of updates which must be delivered per unit of time. Fast applications have latency requirements of about 4 ms within a substation and 8 - 12 ms external to substations. [14]

An event originating at the substation may have to go through a series of SR before reaching

the control center or vice versa. In order to limit the end-to-end latency within the permissible limits, the delay at each SR should be predictably contained within a millisecond.

Statement:

The average delay introduced by the SR for forwarding an event should not exceed one millisecond and should be highly reliable.

The various emerging needs of the communication systems of the EPG such as multicasting and differential QoS requires that the events (status and control) exchanged between the substation(s) and the control center(s) goes through a series of SR. In order to confine the end-to-end latency of these events within permissible limits, each of these SR should restrict the local delays (at each router) to the minimum. Minimizing and bounding these delays can be a challenging task because of the limitations of the operating environments and in some cases the OS on which these SR run.

For example, the stimulus to most real-time applications is external events. These events are conveyed to the OS as interrupts. On receiving the interrupt, the system has to schedule the real-time application marked to handle the event. The interrupt latency is a measure of the amount of time needed to schedule the application and the time needed by the application to take appropriate action. GPOS like Linux and Unix can be tuned to provide real-time support with a high degree of certainty.

3.3.2 QoS Requirements for varying operating environments

Background:

The next generation grid communication will have to operate on wide variety of protocols and networks such as WAN's. WAN's operate on best effort services such as IP, which are prone to delays because of network congestion and queuing delays.

Statement:

The desired QoS should be met even under network congestion or on slow networks.

Achieving QoS while there is abundance of resources is not a daunting task, but it can be when there is conflict of resources. In case of conflicts, measures have to be taken to ensure that the resources are judiciously used. Resource conflicts can be avoided under most cases by employing *resource reservation, admission control and buffering resource requests*. But under rare circumstances of emergency, there can be status data flooding, leading to *resource conflicts*.

Some of the scheduling algorithms such as Delay-EDD are capable of providing bounded end-to-end latencies when there are resource conflicts. Also some of the events may demand very high attention and may require prioritization of events. Scheduling algorithms that meet the requirements should be explored, evaluated and implemented within the system to deliver the requested QoS.

3.3.3 Control Communication Requirements

Background:

In a critical infrastructure such as the electric power grid, the communication infrastructure is much more fixed and thus static routing is a reasonable approach and is generally more efficient than dynamic routing.[14]

Statement:

The SR should provide control mechanisms for an external source to convey information such as event forwarding paths, resource availability and to establish communication channels for sending and receiving events.

The communication channels of the EPG once designed and located, rarely undergo change

in position or capacity. For example the communication systems for the EPG of North America were developed in the 1960s and have not kept pace with the developments happening in network systems. Though there has been a tremendous increase in the available status data from the substations, little has been exploited by the control center because of the conventional and aging network channels. These channels cannot cater to the increasing bandwidth requirements and are unable to protect the data from being hacked. However such a static configuration can also be exploited. As the set of status variables requested by a control center is mostly static, the routing paths hardly undergo change and hence, the routing decisions can be made static. Thus the costlier routing algorithms can be removed from the event forwarding path, bringing down the latency and reducing complexity.

Interesting work has been done in the field of Multicast Routing Heuristic [36], Low-Cost, Low-Delay Multicast Trees [37], Bandwidth Sensitive Routing [48] and QoS-Aware Resource Management [41]. These heuristics and algorithms complement the SR by providing the decision making to manage large distributed system efficiently, external to the SR and communicating these decisions to the SR over the communication channels provided by the SR. Consequently, the SR should work as part of a greater system which supports the needs of the EPG for a flexible communication system. This means that it has to communicate and inter operate with other components of the system. This involves exchanging control commands and messages. The SR should provide control communication for the following

- Receiving Event Forwarding Paths : Because of deregulation and competitive markets, a lot of private utilities have entered the power industry. This has increased the need for information sharing not only within but also across utilities. These crucial event forwading decisions are taken at a higher level than the SR. Therefore the SR should be capable of receiving such event forwarding details from external entities.

- Exchange Resource Availability: For resource reservation and admission control, an inventory of all the free resources should be available. Since the decision making related to resource allocation is done external to the SR, the SR should be able to communicate all its available resources.

- Establishing Communication Channels : External entities should be able to securely establish communication channels with the SR so that it can send/receive events from the SR.

- Provision for adding control commands : With ever increasing demands of the EPG, need for new mechanisms may arise over time. The SR should make provisions for adding new control commands.

3.3.4 Requirement to support Heterogeneity of computing resources

Background:

The communication system must function (and provide QoS guarantees) despite having to span multiple networking technologies, CPU or device architectures, programming languages, and operating systems or runtime systems. [14]

Open architecture: The communications system must be designed, developed, and deployed in a way which allows for easy interoperability across multiple vendors products. [14]

Statement:

The SR should be portable across various operating systems such as Linux, Solaris and other flavors of Unix.

Availability of many OS with different architectures and features allows the user to choose the OS that meets their requirements most closely. This though provides the user different options to choose from, also brings about the need to bridge the communication gap between them. Portability of applications and communication between these different OS's and platforms can sometimes be challenging because of the differences in their architecture. Use of standard API's

such as Berkeley sockets and POSIX within the SR will make the SR more easily portable across UNIX based systems. The ease of portability of the SR will increase its scope of deployment and acceptance.

CHAPTER FOUR

SOLUTION

System implementation of the SR alone would not satisfy all the requirements because OS without real-time support or slow network devices can lead to added latencies, hence the solution can be subdivided into the following components

1. System implementation of the Status Router
    - Component Architecture
    - High level Design
2. Exploiting and enabling operating systems real-time support
3. Equipping slow network devices with appropriate scheduling mechanisms to achieve the desired QoS.

Each of above bullets are discussed as separate sections to follow

4.1 Component Architecture of the SR

The major active and passive components within the SR are as follows

Active

- Control Communication module
- Real-Time Event Forwarding module

Passive

- Event Forwarding Table, henceforth referred to as Routing Table

The active components takes actions on receiving events while the passive components act as data holders for the active components. These components are depicted below in the Figure 4.1

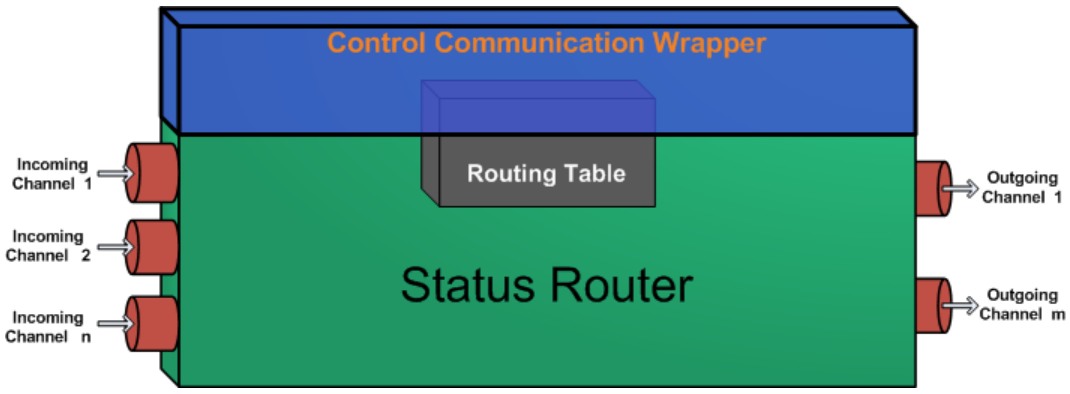

Figure 4.1: Component Architecture Diagram.

4.1.1 Control Communication Module

Purpose

The communication module serves as the command interface for the SR. It provides a communication channel for external entities to convey control information such as Event Forwarding Paths, QoS parameters and for establishing communication paths for sending and receiving events.

List of the Major Control Commands currently exposed by the SR:

1. registerPublisher: Using this command, publishers provide the SR with details such as its name and the host where it is running. If the command is acknowledged, then the SR returns the port number to which the publishers can address the events. A publisher has to register itself before it can start publishing variables. A given publisher can publish any number of variables.

2. publish: The metadata of a to be published variable is passed to the SR through this command. This metadata contains information such as variable name, type, priority and the publisher name. This metadata is used by the SR to interpret any packet that it receives from that publisher.

3. unRegisterPublisher: Removes the caller from the list of publishers known to the SR, including all of its published variables.

4. unPublish: This command can be used by a publisher to inform the SR that it wants to stop publishing a particular variable.

5. registerSubscriber: Using this command, subscribers provide the SR with details such as its name, host where it is running and the port to which the SR can forward the events to. A subscriber has to register itself before it can start subscribing to variables. A given subscriber can subscribe to any number of variables.

6. subscribe: A subscriber can use this command to subscribe to a particular variable with the desired QoS. The important QoS attributes includes subscription interval, maximum permissible latency and priority.

7. unRegisterSubscriber: All the subscriptions of the subscriber are canceled and the subscriber is removed from the list subscribers known to the SR.

8. unSubscribe: Subscription of a particular variable is canceled.

9. setupPath: The module taking the routing decisions can make use of this command to populate the Routing Table of the SR.

10. terminatePath: Removes routing entries from the routing table.

11. connectLink: Establishes the link between two SR's. This command can be used to build a cloud of SR's in the specified layout.

12. disconnectLink: Terminates the link between two SR's.

Protocols for Control Communication

Control communication on a distributed system can be well established using a middleware. Various middlewares such as CORBA, DCE (Distributed Computing Environment), COM (Component Object Model), RPC (Remote Procedure Call), Java's RMI (Remote Method Invocation), Microsoft's .NET provide different level of abstraction to the application developer. The choice between them as the communication protocol for the SR has to be made based on their usability, flexibility, scalability and availability. CORBA was chosen because it provides a lot of transparency and options for the implementation languages, CPU architectures, OS, network protocols and is readily available as an open source middleware.

Extending/Adding new Control Commands

All the control commands are represented using IDL (Interface Definition Language) in CORBA. IDL segregates the implementation from the interface and hence is easily extendable. New command interfaces can be added to the IDL file, which when fed to any CORBA implementation generates easy to attach stubs for clients and servers.

4.1.2 Real-Time Event Forwarding module

Purpose

The core component within the SR is the *event forwarding* module because it has to cater to all the QoS requirements. Events arrive at the SR as network packets. On receiving an event, the forwarding module has to look-up the routing table to find the destination(s) of the event. Then the event is placed on the outgoing communication links connecting the SR to these destination(s). All this processing has to be done in real-time, keeping the latency bounded. The forwarding module has to increase its scheduling priority to make sure that other processes do not preempt it while the event is being processed. This can be done by using the real-time support provided by the OS.

Managing Communication Channels, the Threading Model

A thread is a single, sequential flow of control within a process. Threads are a way for a program to fork itself into two or more simultaneously running tasks. A thread is contained within a process[1] and different threads within a process share the same resources. Switching between threads is typically faster than switching between processes.

Events arriving at the SR from different entities have no correlation between them and can be routed independently of one another. Several threading models can be designed to handle these events. Two of them are discussed below:

- **Single-Threaded**: In the single-threaded model, there is a single thread within the SR which reads the first available packet from the ports of all the incoming channels which are ready to be read. The *select* system call facilitates this feature by returning a set of ready to be read ports. After reading the packet, the thread looks up the routing table to find the routing entry corresponding with the packet. The routing entry has information of the outgoing channel(s) on which the packet should be forwarded to. After sending the packets to these channel(s) the thread moves onto reading the next available packet from the set of ready ports. Once the set is exhausted, the *select* call is made to get a new set of ready ports. If there is no port ready for reading, the thread sleeps on the *select* system call waiting for a packet to arrive. A pictorial representation of a single-threaded communication model is shown below in Figure 4.2

- **Multi-Threaded**: In the multi-threaded model, there is a dedicated thread for each incoming channel. Each of the threads sleep on the *read* system call waiting for a packet to arrive on the port attached to the corresponding channel. The first thread to receive a packet is woken up and scheduled to run. When the thread runs, it reads the packet from the port, and looks up the routing table to find a corresponding routing entry. After sending the packet

[1] A running instance of a program

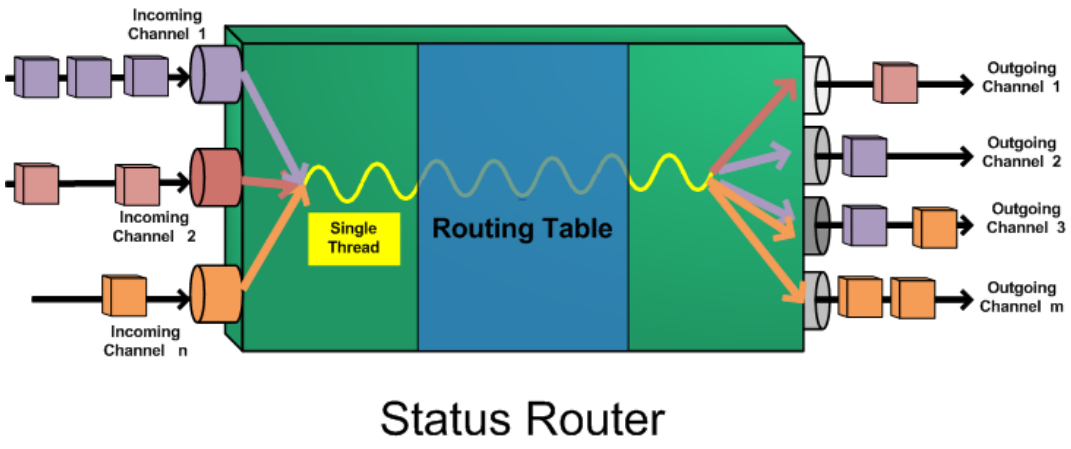

Figure 4.2: Single Threaded Model of SR.

on the associated outgoing link(s), the thread goes back to sleep on the *read* system call waiting for the next packet to arrive. In the mean time if a packet had arrived on any other incoming channel, then the thread attached to that channel would have been scheduled to run. When the first thread goes back to sleep on the *read* system call, the other thread is executed. A pictorial representation of a multi-threaded communication model is shown below in Figure 4.3. On a multiprocessor machine these threads can be executed independently and simultaneously.

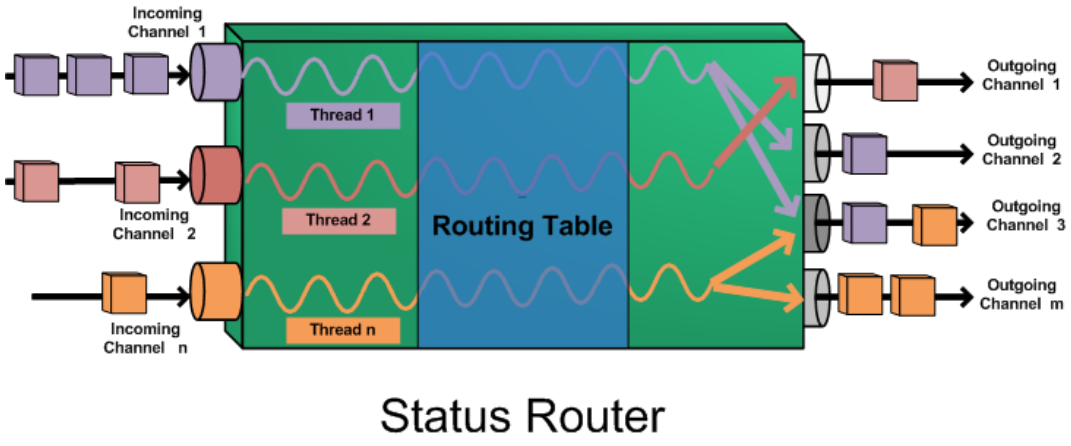

Figure 4.3: Multi-Threaded Model of SR

Real-Time threads in POSIX

To keep the latency to a minimum, once a thread starts executing, it should not be preempted by any other process or thread. For this, the priority of the thread has to be elevated to real-time. On a single-processor system, only one process's code is executing at a time. The scheduler decides which process should have control of the CPU. *The scheduler chooses which process should execute based on priority, therefore the highest priority process will be the one that is executing* [2]. A higher priority process cannot be preempted by a lower priority process. When an event occurs, if the thread marked to handle the event has a higher priority than the running thread, then the running thread is preempted and the higher priority thread is made the running thread. This reduces the latency in handling an event.

Thread Priority

All processes are given an initial priority, either implicitly by the OS or by the user. There are three real-time priority ranges in Linux as shown in the Table 4.1.

	Priority Level	
Range (Low to High)	Minimum	Maximum
Nonprivileged user	SCHED_PRIO_USER_MIN	SCHED_PRIO_USER_MAX
System	SCHED_PRIO_SYSTEM_MIN	SCHED_PRIO_SYSTEM_MAX
Realtime	SCHED_PRIO_RT_MIN	SCHED_PRIO_RT_MAX

Table 4.1: Priority Ranges in UNIX

The priority of the routing threads should be real-time to avoid being preempted by any other process or thread. Priority of a thread can be set using the *sched_setparam* function.

Scheduling Policy

The scheduling policy determines the order in which the various threads should execute. There are two fixed priority scheduling policies to schedule the various real-time threads on a system. *SCHED_FIFO*: First-in first-out scheduling and *SCHED_RR*: Round-robin scheduling. The first-in first-out scheduling policy (SCHED_FIFO) gives maximum control to the application. Here

each thread[2] runs to completion or until it voluntarily yields or is preempted by a higher-priority thread. In the round-robin scheduling policy (SCHED_RR), every running thread is allocated a quantum. When it finishes its quantum, it goes to the end of the thread list of its priority. The *sched_setscheduler* function can be used to set the scheduling policy. The various thread priorities, scheduling policies and the functions associated with them are explained in detail in [2].

A snippet of the C code to create threads with real-time priorities in Linux (as used in the SR), with SCHED_FIFO as the scheduling policy is shown below.

```
struct sched_param param;
//Set the priority to Max
param.sched_priority=sched_get_priority_max(SCHED_FIFO);
//Initialize the thread attribute 'attr' with default values
pthread_attr_init(&attr);
//Set the scheduling parameter inheritance state attribute
//in the specified attribute object
pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);
//Set the scheduling parameter
pthread_attr_setschedparam(&attr, &param);
//Set the scheduling policy
pthread_attr_setschedpolicy(&attr,SCHED_FIFO);
//Create thread with the given attributes
ret = pthread_create(&m_thread, &attr, threadFunction, NULL);
```

Multi-Threaded model was chosen as the threading model for the SR for the following reasons:

- The different real-time threads can execute in parallel on a multiprocessor machine giving better performance and reduced latency. Routing table is the only shared resource between these threads. Use of read-locks to access the routing table makes the different threads more concurrent.

- The different threads can be assigned different real-time priorities based on the priority of the event being handled by them. This helps in providing differential QoS because the thread with higher priority will be always chosen to be scheduled prior to a lower priority thread.

[2]Threads and Process can be interchangeably used in this context

- In a single-threaded application, if a thread blocks on a system call, the whole process gets blocked. On the other hand, if there are multiple threads, even if one of them blocks, the CPU can schedule the other ready threads, thereby increasing throughput and decreasing latency.

- In a multi-threaded SR, as the various threads in the SR would be sleeping, waiting for an event to occur, CPU is not consumed and, hence a multi-threaded SR can be considered as light as a single-threaded SR.

Independent sending thread is overkill

A different architecture would suggest having a separate sending thread to optimize on the exceptionally[3] slow *send* system call. In this design, the receiving thread(s) reads from the incoming channel(s) and deposits the packet along with the outgoing link(s) information, in a buffer. It then wakes up the sending thread, subsequently going back to sleep waiting for a packet to arrive. The sending thread removes a packet from the buffer and sends it on the corresponding outgoing link(s). Once all the packets in the buffer have been exhausted the sending thread then goes back to sleep waiting to be woken up by a receiving thread. By detaching sending from receiving into separate threads, the receiving thread does not have to wait on the *send* call and can move onto processing the next packet. A buffer between the receiving and the sending thread can provide margin for the congestion to ease out. For devices with slow network interface, the buffer also bridges the gap between a fast receiving thread and a slow sending thread without affecting the latency. Network congestion or slow network does not always warrant a slow *send* system call. The underlying network protocol also plays a major role. For example, if the network protocol is UDP which does not guarantee any kind of reliability, then the packets which cannot be handled by the network are just dropped. The protocol does not take any feedback from the network and hence does not affect the *send* system call. On the other hand, a network protocol such as TCP (Transmission Control

[3] under scenarios of network congestion or devices with slow network interface

Protocol) which provides reliable, in-order delivery of packets, does not accept more packets from the application than that can be handled by the network. The feedback is provided to the application through the *send*[4] system call by blocking the call to the extent required. Because UDP is the network protocol used within the SR, a separate sending thread will be redundant for reasons stated above.

4.1.3 Routing Table

Purpose

Routing table maps variable ID[5] to routing entries. Routing entries contains a list of all the subscribers that have subscribed to that variable. Subscribers subscribe to variables at a particular subscription interval, also present in the routing entry. For example, a subscriber S1 is interested in receiving variable V1 updates only once every 1 second while another subscriber S2 would like to receive V1 updates every 100ms. The event forwarding thread, on receiving a variable, looks up the routing table for the mapped routing entry. Using a rate filtering algorithm, the thread then shortlists the subscribers whose subscription intervals are met. The variable is then forwarded on the communication paths leading to these shortlisted subscribers.

4.2 High Level Desigh of the SR

The design of the Java SR [29] gave due consideration to modularity, multi-threading and use of buffers to avoid frequent memory requests. The design of the C SR borrows most of the modules from the Java SR, but uses a different threading model for reasons stated in Section 4.1.2. The Java SR uses an independent thread for each incoming and outgoing link and buffers to pass events between these threads. The C SR associates a thread only to an incoming channel to avoid unnecessary thread context switching. Also to keep the design of the SR compatible with its initial Java prototype, the SR is differentiated as edge SR and SR. edge SR, in addition to the routing

[4]asynchronous *send* does not block but can be complicated to handle

[5]Each variable published by a publisher is given a unique ID.

functionality provided by the SR, provides means for end-entities to establish connection and publish/subscribe to events. For detailed discussion on the differences between edge SR and SR, refer Section 3.2.4 of [29]. The design of the (edge) SR in terms of the major classes/components and the interaction between them is presented in Figure 4.4.

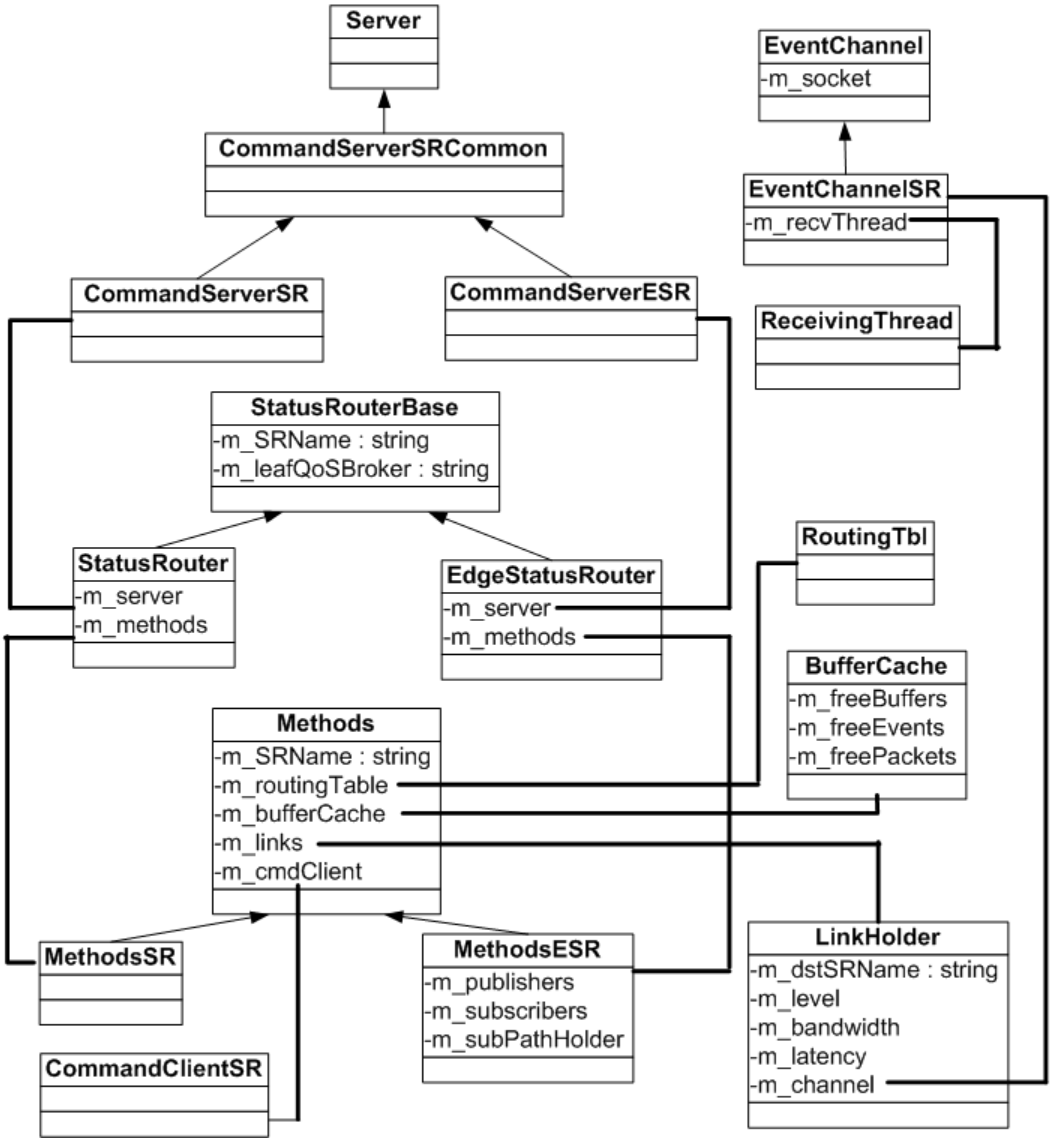

Figure 4.4: Class Diagram of the C SR

4.2.1 The Command Module

CommandServerESR and *CommandServerSR* handle all the incoming commands for the edge SR and SR respectively. The list of major commands supported by *CommandServerESR* is listed in Section 4.1.1 while only commands 9,10,11 and 12 in the list is supported by *CommandServerSR*. These commands can be used by external entities to control the mechanisms of the (edge) SR. The commands are delegated to the *MethodsESR* and the *MethodsSR* class for appropriate action. These classes also keep track of all the connected components such as publishers, subscribers and other linked SR's, by pinging them at regular intervals.

4.2.2 The Main Module

EdgeStatusRouter and *StatusRouter* are the main classes for edge SR and SR respectively. They instantiate all the other necessary classes and link them together. For example, they initialize the CORBA system for control communication, start the Command(E)SR server to listen to control commands and initiate a thread which waits for user input for termination of the (edge) SR. The common methods between these classes are implemented within the *StatusRouterBase* class.

4.2.3 Routing Table

All the information required to route an incoming event to the corresponding outgoing channel(s) is contained within the *RoutingTbl* class. Routing table is discussed in detail in Section 4.1.3.

4.2.4 Buffer Cache

A global pool of buffers is maintained by the *Methods* class. Every time a packet arrives at the incoming channel, a free buffer is taken from the *BufferCache* and alloted to the packet. Once the packet has been sent, the buffer is returned to the *BufferCache* for further allocation. This encourages reuse of memory and prevents new memory request for each incoming packet, thereby reducing the overall latency.

4.2.5 EventChannelSR

This class acts as the communication channel or the message bus for the events. It provides means for other entities to connect to it and send/receive events. There is a receiving thread attached to each connected (incoming) link. The *EventChannelSR* class derives from the *EventChannel* class which abstracts all the system specific *socket* details.

4.2.6 Path of the event within the SR

1. The receiving thread requests a free buffer from the *BufferCache*.
2. Then it waits indefinitely on the incoming channel for any events to be read into the free buffer.
3. Any incoming event is passed on to the routing table (*RoutingTbl*) for further action.
4. The routing table extracts various parameters such as the time event was created and variable ID from the packet.
5. The routing table then decides whether the event has to be flooded or not.
6. If not, then the packet is filtered using a rate filtering algorithm and is sent on the outgoing links leading to the subscribers for which the time interval requirements are met.
7. After sending the event, the buffer is returned to the *BufferCache*.

A user space application relies on the OS to provide it the necessary mechanisms to achieve its goals. The following section explains how the SR uses the OS real-time extensions/features to meet its latency requirements.

4.3 Exploiting And Enabling Operating Systems Real-Time Support

4.3.1 Desirable Operating System Properties for Real-Time applications

A real-time OS should be able to schedule events so that they happen in a deterministic amount of time. This OS properties which contribute to the deterministic behavior are *kernel preemption, scheduler latency, scheduler's run-time complexity and priority scheduling*. Responsiveness of the OS to events is critical to real-time systems. The definition of real-time used in defining the scope of the POSIX 1003.1b standard (realtime extensions) is:

"Real-time in operating systems: the ability of the operating system to provide a required level of service in a bounded response time." [4]

Kernel Preemption

Kernel preemption is the ability to preempt the kernel to run a higher priority (ready) task. If the kernel is not preemptible and a real-time task is runnable, then the kernel will run to completion before scheduling the task. This can introduce indeterministic response time in the real-time task. For the SR, kernel preemption is an important factor to be considered because on arrival of an event, the SR should be scheduled to run without any delay. If the kernel is non-preemptible and is performing a time consuming task (for eg: Disk I/O), then the event will only be delivered to the SR after the kernel has finished its task. This considerably increases the end-to-end latency of the event.

Scheduler Latency

Scheduler latency is the delay between the occurrence of an interrupt and the running of the process that services the interrupt. *In the context of the kernel, it is the time between a wakeup (the stimulus) signaling that an event has occurred and the kernel scheduler getting an opportunity to schedule the thread that is waiting for the wakeup to occur (the response)* [50]. In the context of the SR this is the time delay after a packet arrives on the host machine where the SR is running and before the SR receives the packet. Preemptible kernel and scheduler latency are closely related concepts

because the degree to which the kernel is preemptible directly affects the scheduler latency. A fully preemptible kernel will have a significant lower and more predictable scheduler latency than a non-preemtible kernel.

Scheduler's Run-Time Complexity

An important goal of the scheduler is to allocate CPU time slices efficiently between the various ready-to-run tasks. It has to minimize response times for critical real-time tasks while maximizing overall CPU utilization. Run time complexity of the scheduler is a measure of the time taken by the scheduler to schedule a task. If it is of the order of O(n), then the time taken to schedule a task is directly proportional to the number of active tasks n. A much better algorithm having a complexity of O(1) should be able to schedule the tasks in a constant time.

Priority Scheduling

Priority scheduling is the ability of the scheduler to run a higher priority task (if required, by preempting a lower priority task) without being interrupted by any other lower priority task. A preemptive kernel plays a major role here as it can guarantee that a higher priority task can quickly preempt a lower priority task irrespective of whether the lower priority task is running in user or kernel mode [2], as seen in Figure 4.6. In the case of a non-preemptible kernel, once a user enters the kernel by issuing a system call, all context switches are disabled until the system call is completed, as seen in Figure 4.5. Therefore a lower priority task in the midst of a system call, can delay the execution of a higher priority real-time task.

4.3.2 OS Evaluation

The rational for choosing General Purpose OS (GPOS) as the platform for hosting the SR has already been provided in the Section 3.3.4. There are several freely and commercially available GPOS with varying real-time characteristics, like Linux, Sun Solaris and Windows. Table 4.2 compares the three OS with respect to their availability and real-time response, compiled using [49, 4].

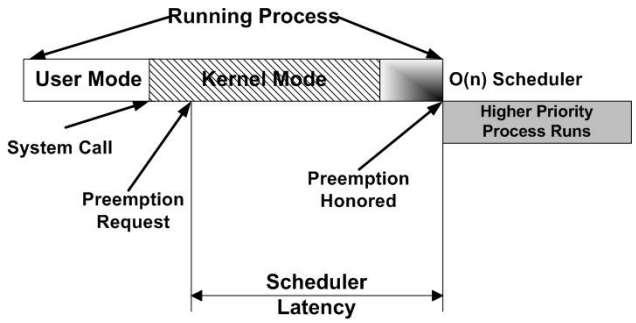

Figure 4.5: Non-Preemptible Kernel with O(n) Scheduler: Higher Scheduler Latency

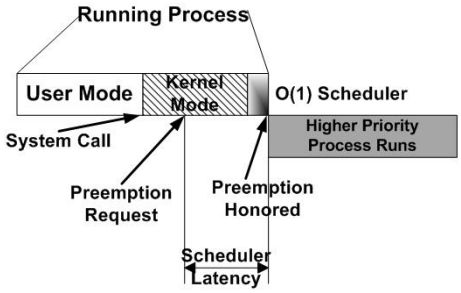

Figure 4.6: Preemptible Kernel with O(1) Scheduler: Lower Scheduler Latency

Linux and Solaris have very similar API model, mostly complying to either Portable Operating System Interface (POSIX eg: threads) or Berkeley Software Distribution (BSD eg: Sockets), while Windows comes with custom API's for accessing system resources. Therefore a C application is very easily portable across Linux and Solaris. As Linux is one of the more readily available and

	Characteristics				
Operating System	Soft RT	Hard RT	Availability	Extendable	Priority Scheduling
Windows(Vista)	Y	N	Commercial	N	Y
Linux 2.6.18.3+	Y	N Y(RT Patch)	Free	Y	Y
Solaris 8.0+	Y	Y	Free	Y	Y

Table 4.2: OS Comparison

extendable OS with patches available to support hard and soft real-time applications, it was chosen as the platform to implement the SR.

4.3.3 Real-Time System Tools in Linux

Linux is equipped with several real-time extensions which the application can exploit to suit its real-time needs. The ones relevant and used in the SR are discussed below:

- Real-Time Threads: Real-time threads can be created in Linux using the standard POSIX thread API *pthread_create* with the help of the parameter *pthread_attr_t*. This parameter contains all the essential information such as the scheduling policy to be used for the thread and the thread priority. These values are used by the scheduler to schedule the real-time threads amongst other running tasks in the system. The various scheduling policy and thread priority available is discussed in detail in Section 4.1.2. The SR uses these attributes to create real-time receiving threads to handle incoming events.

- Memory Lock: Memory management ensures that a process has enough memory to continue execution. The memory address space visible to the application is usually virtual and is transparently translated to actual physical memory address by the memory management. This feature allows the application to have a memory space only bounded by the addressing space of the system. This is achieved by swapping not so recently used memory pages from the faster main memory to the capacious but slower secondary disks. But it comes with a cost, the swapping of pages between main memory and secondary disks is done using I/O (input/output) which is time consuming and can be critical for real-time applications. A time critical process should be locked into memory. UNIX provides memory locking functions which allow locking the entire process at the time of the function call and throughout the life of the application, or selectively lock and unlock as needed. The following memory locking/unlocking functions as shown in Table 4.3 are available on UNIX Platforms[2]

Function	Description
mlock	Locks a specified region of the process's address space
mlockall	Locks all of the process's address space
munlock	Unlocks a specified region of the process's address space
munlockall	Unlocks all of the process's address space

Table 4.3: Memory Locking/Unlocking Functions

The SR employs the *mlockall* system call to lock all of its pages in main memory before processing any events. The effects of locking a data memory region ("buffer") using *mlock* is shown in the Figure 4.7, taken from [2]

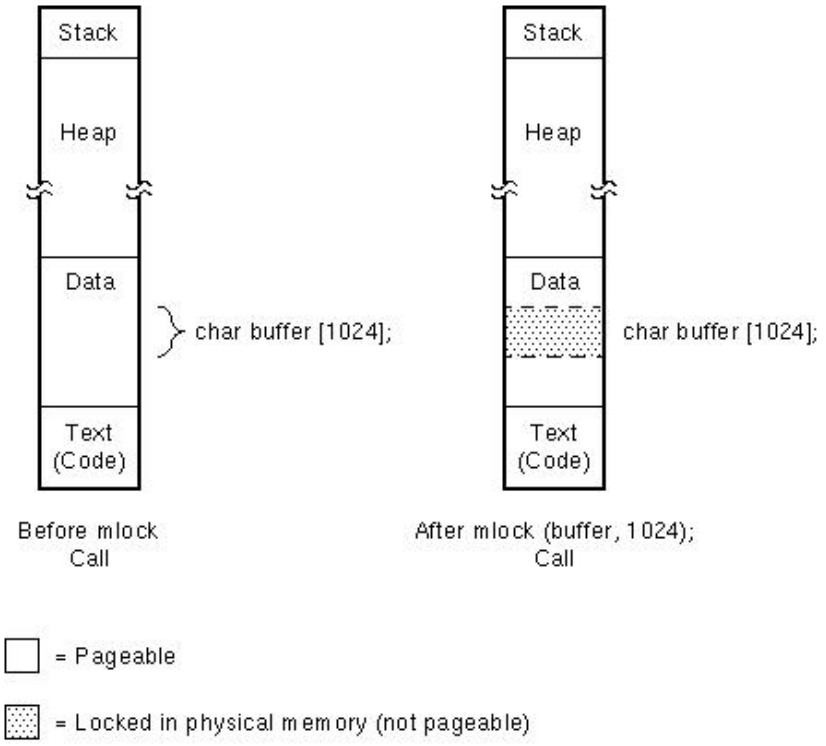

Figure 4.7: Locking memory with mlock.

- Real-Time Patches[6]: Linux has two essential real-time patches. They are

[6]Refer Appendix A on HOWTO apply these patches

- Preemption Patch: Makes kernel more preemptible by creating opportunities for the scheduler to be run more often thereby reducing the time between occurrence of an event and running of the scheduler. It also helps prioritizing tasks because if a higher priority task becomes runnable, then the kernel can be preempted and the task can run.
- Low-Latency Patch: This patch focuses on introducing explicit preemption points within parts of kernel which have long execution times. This involves techniques such as lock breaking, in which a spin lock is dropped, the scheduler is invoked and then the spin lock is reacquired. Also places where large data structures are iterated are inspected to check if the loop has crossed the allowed threshold and if so the scheduler is appropriately invoked if possible.

  A comparison of the scheduler latency on vanilla Linux 2.4.17 and Linux 2.4.17 with Low Latency and Preemption patch as taken from [50], is given in Figure 4.8. The Low Latency and the Preemption patch have merged into a single real-time patch for Linux kernel 2.6 and above. The results given in the figure below are also applicable to any patched Linux kernel above 2.4.17.

The impact of using real-time threads and Memory Locking on the performance of the SR is illustrated in the Experiment 5.2.2.

The above mechanisms alone may not be sufficient to achieve desire QoS on devices with slow network interfaces under exceptional operating environments. The following section explains how scheduling algorithms can be used to circumvent such issues.

Scheduler Latency in milliseconds	
Vanilla Linux Kernel 2.4.17	Linux Kernel 2.4.17 + Low Latency Patch + Preemption Patch
Maximum: **232.7ms**	Maximum: **1.3ms**
Mean: 0.0883ms	Mean: 0.052ms
Standard Deviation: 2.119ms	Standard Deviation: .0168ms
92.84442% of samples < 0.1ms 97.08432% of samples < 0.2ms 99.73050% of samples < 0.5ms **99.84382% of samples < 0.7ms** 99.94038% of samples < 1ms 99.97922% of samples < 5ms 99.98096% of samples < 10ms 99.98590% of samples < 50ms 99.98828% of samples < 100ms 100.00000% of samples < 232.7ms	**98.31268% of samples < 0.1ms** 99.76028% of samples < 0.2ms 99.99648% of samples < 0.5ms **99.99978% of samples < 0.7ms** 99.99992% of samples < 1ms 100.00000% of samples < 1.3ms

Figure 4.8: Scheduler Latency Comparison.

4.4 Equipping Slow Networks with Appropriate Scheduling Mechanisms

4.4.1 Why is there a need of a Scheduling Algorithm?

A crisis or contingency in the EPG could trigger a series of events leading to additional demand of status variables from the substations, and a flurry of control commands from the control center. This increased load can lead to network congestion and resource conflicts even on a managed subsystem. This will challenge the devices with slow network interfaces to provide the admitted QoS, because of limited resource availability. Appropriate mechanisms should be in place to cope with such situations. Because of limited network bandwidth, excess[7] packets transmitted over unreliable protocol (eg: UDP) as in GridStat, will be lost. Buffers have to be in place to queue such packets. But buffers have an undermining effect on QoS specifically on end-to-end delays, if

[7] more than what can be handled by the network

not managed through appropriate scheduling algorithm. Consequently there is a need for buffers, regulated by a scheduling algorithm that can provide the desired QoS.

4.4.2 Which Scheduling Algorithm?

Much research has been done on the various packet scheduling algorithms and their capabilities in bounding maximum delay. Joel Helkey in his masters thesis [33] proposes *Delay-EDD* as the scheduling discipline which can meet the end-to-end delay bound requirements of a real-time status dissemination network such as GridStat. [33] implements Delay-EDD within GridStat and carries out experiments to prove the same. Based on the experiments it concluded that *"simple scheduling algorithms in the routers, like First In First Out or priority are not adequate, because they do not protect well behaving flows from different sources of variability inside the network"*.

In Delay-EDD each destination *i* declares its performance requirements in terms of the end-to-end delays D_i. D_i is then broken down into local delay bounds $d_{i,j}$ at each router node *j*. The local bounds are computed so that, if the node *j* can assure that no packet on channel *i* is delayed locally beyond its local bound $d_{i,j}$ then the end-to-end delay bound D_i can be met. Scheduling is done based on deadline. The router sets a packets deadline to the time the packet should have been sent had it been received according to the traffic contract [39]. Let $D_{i,j}^k$ be the deadline assigned to the packet *k* for the flow *i* by the router *j*, then:

$$D_{i,j}^k = max\{a_{i,j}^k + d_{i,j}, D_{i,j}^{k-1} + X_{min}^i\}$$

where

- $a_{i,j}^k$ is the arrival time of the packet *k* of flow *i* at router *j*.
- $d_{i,j}$ is the local delay bound for flow *i* at the router *j*.
- X_{min}^i is the minimum inter-arrival time for connection's *i* packets, as established during the connection-establishment phase.

4.4.3 Where should it be located?

A network packet has to go through many layers/modules in the OS after leaving the application and before reaching the network interface, as shown in Figure 4.9. Details are available in [26]. The semantics of the protocol governs the course of the packet under different operational environments. For example, UDP a unreliable internet protocol does not make any guarantees on packet delivery and packets can go missing (network congestion, destination host down) without notice while TCP provides reliable, in-order delivery of packets.

Under heavy network traffic load, the rate at which the application pushes the packets to the OS increases while the rate at which the network interface sends the packets to the physical layer remains the same. If the network interface has spare bandwidth, it can adjust to the increase in demand, but slow network interfaces have limited bandwidth and fail to accommodate. This leads to queuing of packets in the *ring buffer* and the *queuing discipline (qdisc)*. On overflowing, if there is no *qdisc* implemented, for unreliable protocols, packets are dropped and no feedback is given to the application. Therefore applications by themselves cannot adjust to the increased traffic. In order to reduce packet losses and to achieve bounded end-to-end delays, priority based Delay-EDD should be implemented as a *qdisc* with a configurable buffer size. *qdisc* is a scheduler and a major building block on which all of Linux Traffic Control [38] is built. It is implemented as a kernel module and is attached to a particular network interface and filters/schedules all the packets for that interface. Similar work has been done in [52] which implements WTP (Weighting Time Priority) scheduling algorithm as a queuing discipline for wireless LANs (Local Area Network) to achieve proportional delay differentiation.

4.4.4 Challenges in Implementing Delay-EDD Scheduling Algorithm as a Queuing Discipline

Scheduling algorithms such as First In First Out (FIFO), WTP and many others do not need any prior information about the flow and delay bounds to schedule the packets. They use the arrival time of each packet to schedule them accordingly. Conversely, as seen from Section 4.4.2, the

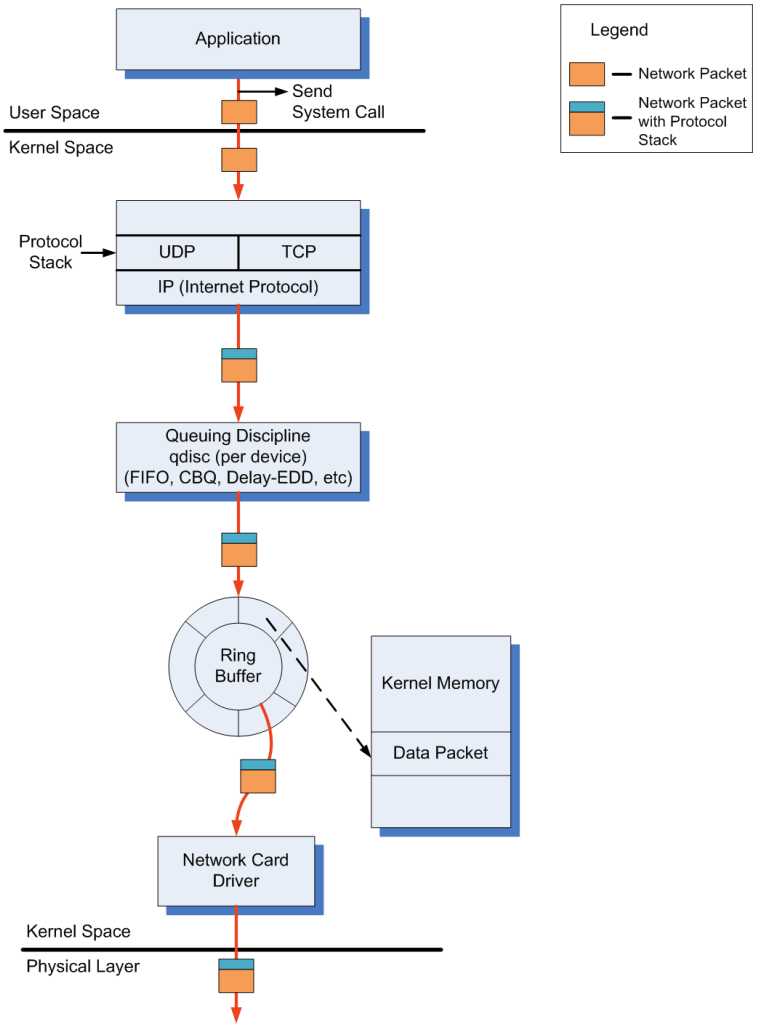

Figure 4.9: Transmission of a Packet

Delay-EDD algorithm requires the values for the local delay bounds and the minimum inter-arrival time of each flow to calculate the deadline for each packet. This information is provided to the router application by the publishers and the subscribers during connection establishment. Implementing Delay-EDD as a qdisc, which is a Linux kernel Module (LKM) restricts its ability to communicate with user applications. Since the router is implemented as a user space application and Delay-EDD as a LKM, some means for communication between them has to be established across the user-kernel space boundary created by the OS.

A system call is one such means for passing information between the user application and the kernel. The Linux kernel permits the addition of new system calls. The kernel can also be hacked so that a LKM can intercept any system call. This is required because Delay-EDD will be implemented as a qdisc, which are essentially LKM's. The steps for adding and intercepting system calls is given in detail in Appendix two.

4.4.5 System Call Interface/Algorithm for Delay-EDD as a qdisc

The pseudo code below describes the system call interface and the algorithm used to implement Delay-EDD scheduling algorithm as a queuing discipline within the Linux kernel.

```
/*
This function initializes the Delay-EDD Kernel Module. Various Queues
and hash maps are initialized. The original enqueue and dequeue function
pointers are overwritten with Delay-EDD's enqueue and dequeue function
pointers.
        offsetVarID: Offset of VariableID field within the network packet.
        offsetArrivalTime: Offset of ArrivalTime field within the network
                packet. All concerned incoming packets are timestamped.
        queueLength: The Maximum Length of the Queue that will hold the
                pending network packets to be sent.
        Returns SUCCESS if Delay-EDD successfully initialized,
                ERROR otherwise.
*/
int initModule(int offsetVarId, int offsetArrivalTime, int queueLength){
  Initialize Flow Queue (flow_q). This Queue will contain the details about
    each registered flow or destination.
  Initialize Variable ID Hash Map (variableID_h). This hash map will map
    registered variable ID's to its Expected Deadline.
  Initialize Packet Queue (pktQ) with maximum length as 'queueLength'.
    This Queue will contain the network packets sorted on their deadlines.
  Hook the Delay-EDD's network routines (enqueue, dequeue, requeue and drop)
        with the linux kernel.
  Intercept the system calls added for Delay-EDD(registerVariable,
        unRegisterVariable, registerFlow and unRegisterFlow).
  return SUCCESS
}

/*
This function does the necessary cleanup of the Delay-EDD kernel module.
All allocated memory for Queues and Hash Map is freed. Original Network
routines are restored.
*/
cleanup(){
```

```
  Restore original network routines.
  Free all allocated memory for Variable ID Hash Map, Flow Queue
        and Packet Queue.
}

/*
Registers a variable ID along with the Maximum Local Delay(delay) and
Minimum Inter-Arrival Time(XMin) with the Delay-EDD Module.
        varId: Variable ID of the variable to register.
        delay: Maximum Allowed Local Delay for the variable.
        XMin: Minimum Inter Arrival Time for the variable.
        Returns SUCCESS if variable successfully reigstered,
                ERROR otherwise.
*/
int registerVariable(long varId, long delay, long XMin){
  Search varId within the Variable ID Hash Map
  If varId present in the map
    return ERROR
  else
    Create an instance of ExpectedDeadline using varId, delay and XMin
    Map it to varId in the ExD Hash Map
  return SUCCESS
}

/*
Unregisters the variable with ID varId from the Variable ID Hash Map.
        varID: ID of the variable to unregister.
        Returns SUCCESS if variable successfully unregistered,
                ERROR otherwise.
*/
void unRegisterVariable(long varId){
  Remove varId and any mapped values from the Variable ID
        Hash Map, if present
}

/*
Registers a flow/destination with Delay-EDD Module. Since all network
packetsgo through the Delay-EDD Module, there should be means to
differentiate between concerned and other general packets. Only network
packets destined for registered port and destination IP address are
looked at by the Delay-EDD Module.
        port: Port Number of the Destination
        destIPAddress: IP Address of the Destination
        Returns SUCCESS if flow successfully reigstered and ERROR
                if any error occurs.
*/
int registerFlow(unsigned short port, unsigned long destIPAddress){
  Check if the destIPAddress is present in the Flow Queue
  If present return ERROR
  else add destIPAddress to the Flow Queue
  return SUCCESS
```

```
}

/*
Unregisters an already registered Flow.
        port: Registered Port Number of the Destination
        destIPAddress: Registered IP Address of the Destination
        Returns SUCCESS if flow successfully unregistered, ERROR otherwise.
*/
void unRegsiterFlow(unsigned short port, unsigned long destIPAddress){
  Remove destIPAddress from the Flow Queue, if present
}

/*
This routine is called by the kernel whenever it receives a network
packet from the application.
        Packet: Network Packet sent by the application
        Returns SUCCESS if the packet was successfully queueed,
                ERROR otherwise.
*/
int enqueue(Packet){
  Check if the destination Address of the packet matches with any of the
  registered Flows in the Flow Queue.
  If not then call original Enqueue(Packet)
  else
    varID=extract variableID from the Packet using offsetVarId
    arrTime=extract arrival time from the Packet using offsetArrivalTime
    Search for varID in the Variable ID Hash Map
    If not found
      call original Enqueue(Packet)
    else
      Get the ExpectedDeadline object mapped to varID in the Variable
        ID Hash Map
      Calculate new Expected Deadline
      Add the Packet to the Packet Queue sorted on the Expected Deadline
        value.
  return SUCCESS
}

/*
This routine is called by the kernel whenever it is ready to send the
next network packet. The function should return the pointer to the
network packet that has to be sent.
        Returns the packet that has to sent over the network.
*/
Packet dequeue(){
  Return the first packet from the Packet Queue if any present.
  else return original dequeue()
}
```

CHAPTER FIVE

EXPERIMENTAL RESULTS

Experiments were conducted to show and compare the performance (latency) of the C SR against the Java SR, within the GridStat framework. Publishers and subscribers from the GridStat project were ported from Java to C to remove any possible latency overhead at the publishers/subscribers and accurately measure end-to-end latency. A load scalability and a maximum throughput test was performed to document the behavior of the SR under load conditions. A hop scalability test determined the latency trend with a chain of SR's. Also experiments were carried out with different configurations/features of the SR and the OS to show their effect on the local delay at the SR. Experiments were also carried out to evaluate the performance of the Delay-EDD scheduling algorithm implemented as a queuing discipline. Analysis of the results is done after every experiment and possible enhancements/bottlenecks are pointed out. The remainder of this chapter is organized as follows. Section 5.1 describes the various hardware and software settings used for the experiment and the experiment procedure. Section 5.2 describes and analyzes the conducted experiments.

5.1 Experiment Setting

5.1.1 Hardware/Software Specification

Each machine used for the experiment had a single processor Intel(R) Pentium(R) 4 CPU 1.50GHz with Hyper-Threading, 250KB Cache or equivalent, 500 MB RAM and a 100Mb/s Network Card.

Linux 2.6.24 SMP kernel was used for all experiments. GridStat 3.0 Source and Binaries were used and compiled using Java 1.6.0. The C SR was compiled using gcc 4.1.2 without any optimization flags. For CORBA, MICO version 2.3.11 was used.

5.1.2 Experiment Setup

- Publishers and subscribers run on the same machine for accurate latency calculations.

- Reference system is publishers/subscribers across which the latency is measured. Load system is publishers/subscribers which help in generating load.
- Publishing interval is 10ms for reference system and 1ms for load system unless stated otherwise
- Subscribing interval is 10ms for reference system and 1ms for load system unless stated otherwise
- Experiments involving Java SR are allowed to run at least 20 minutes to initialize the JVM, prior to taking any samples.
- For each experiment millions of sample points are taken. The graphs may show only a small portion of the sample points because of limited plotting capabilities.
- All latency values are in micro seconds (μS) unless stated otherwise.
- In the topology diagram of an experiment, each color represents a separate host machine.
- The experiment setup is inspired from [29] to have a comparable results.
- Only one event of size 24 bytes is bundled with each packet. Hence, the throughput mentioned as packets/ms is equivalent to events/ms.
- Some of the experiments involve *load variables*. The publisher publish the load variables in a loop with some lag between each load variable. After publishing all the load variables, the publisher sleeps for the publishing interval and then the process is continued.

5.2 Experiments

5.2.1 Base Experiment

This experiment is a base setup with 1 publisher running on Host1, publishing a variable every 10ms, 1 SR on Host2 and 1 subscriber on Host1 subscribing every 10ms. The topology of the

experiment is given in Figure 5.1. The experiment aims at measuring the lower bound end-to-end latency and to show that C SRs have much lower latency overhead than their Java counterparts.

Figure 5.1: Topology for the Base Experiment

Graphical and Tabular representation of the end-to-end latency is given in the Figures 5.2 and 5.3. End-to-End latency is the delay between the time an event is published by the publisher and the time that event arrives at the subscriber.

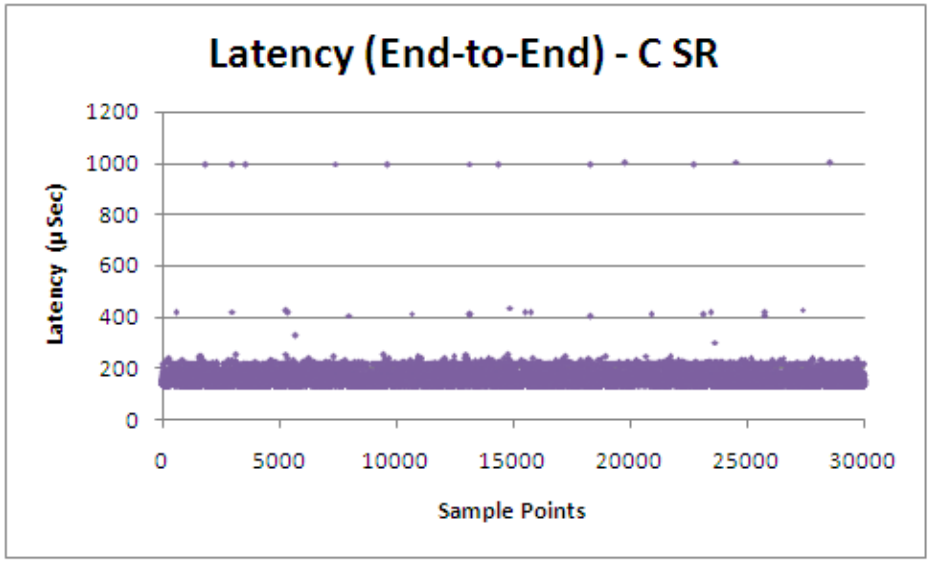

Figure 5.2: End-To-End Latency - C SR

Analysis of Results

From the Figure 5.7 it can be clearly seen that both the average and maximum end-eo-end latency of the C SR is much lower than that of the Java SR. While all the latencies values were less than a millisecond for the C SR, the maximum latency for Java SR was recorded at around 12 ms. From

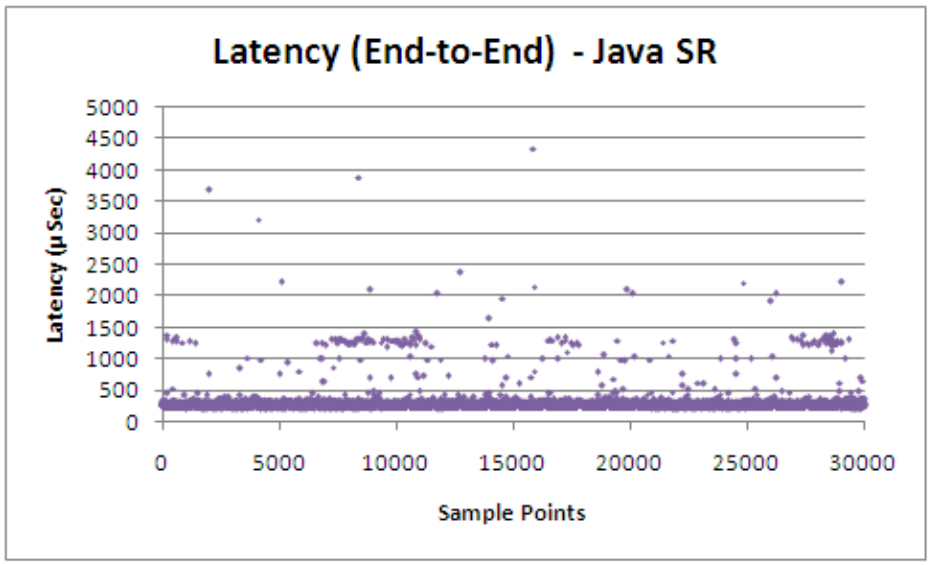

Figure 5.3: End-To-End Latency - Java SR

the Figures 5.2, it can also be seen that a signification portion of the values lie below the 500μS mark for the C SR, but the same does not hold true for the Java SR. For a detailed comparison of latency distribution refer to Figure 5.4.

As it can be observed from the Latency Distribution Table (Figure 5.4),the majority of the

Range(µS)	# Samples	Percentage
101-200	241260	96.504
201-400	8495	3.398
401-500	145	0.058
701-1000	84	0.033
1001-1100	16	0.006
Total	250000	
	C SR	

Range(µS)	# Samples	Percentage
201-300	239136	95.6544
301-400	9091	3.6364
401-500	364	0.1456
501-1000	340	0.1360
1001-2000	961	0.3844
2001-3000	67	0.0268
3000-8000	40	0.0160
11001-12000	1	0.0004
Total	250000	
	Java SR	

Figure 5.4: Latency Distribution Table

values for the C SR is within the range 100-200μS and 200-300μS for Java SR. On further calculation, it can be calculated that the percentage of samples below the 500μS mark is *99.96%* for C

SR and *99.436%* for Java SR, a difference of *0.524%*.

A mere analysis of the end-to-end latency will not be sufficient to compare the performances of the C and the Java SR because of the involvement of other factors such as operating system overhead and network delays. Figures below will help compare the behavior of the C and the Java SR in isolation. Figure 5.5 and 5.6 show the local latency for the C and the Java SR respectively. Local latency at the SR is the delay between the time a packet is picked up by the SR after the *recv* socket call completes and the time that packet is released by the SR just before the *send* socket call.

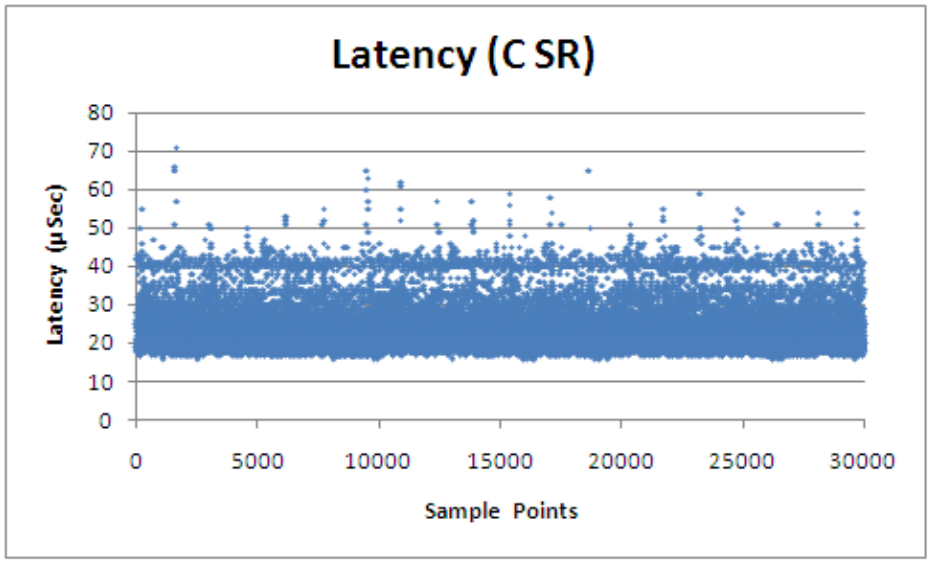

Figure 5.5: Local Delay at the C SR

Local delay values from Figure 5.7 show that there is an *85%* improvement in the average performance of the C SR over the Java SR. Also the *maximum* local delay at the C SR *(*71μS) which is much less than the *minimum* local delay at the Java SR *(*107μS). It can be also be seen that the Java SR is unable to bound the maximum local delays to the *sub millisecond* range.

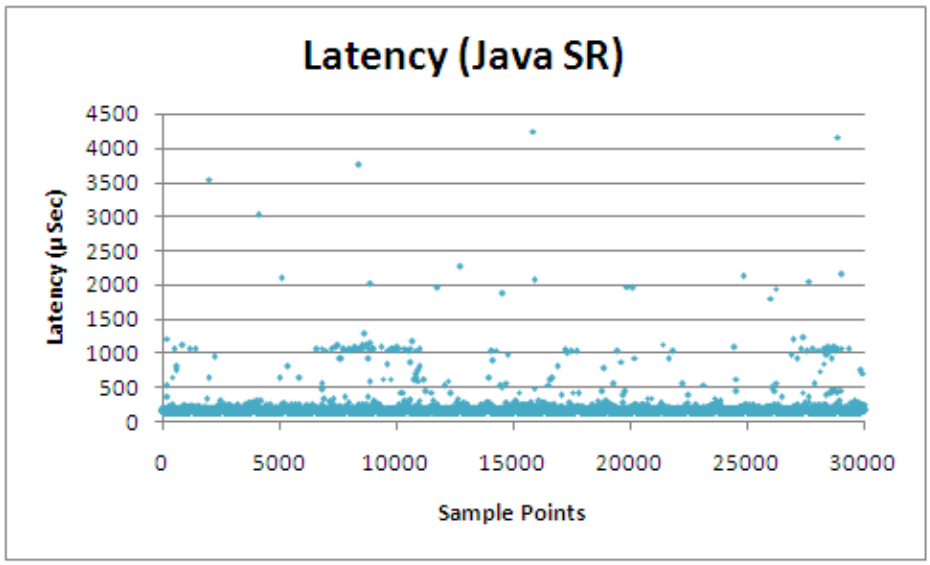

Figure 5.6: Local Delay at the Java SR

	Latency (µSec)			
SR	Average	Min	Max	Std. Dev. (σ)
C SR	152	128	**1042**	25.640
Java SR	260	202	**11742**	94.076
		End-To-End Latency		

	Latency (µSec)			
SR	Average	Min	Max	Std. Dev. (σ)
C SR	24	16	**71**	6.046
Java SR	158	107	**11617**	83.472
		Local Delay at SR		
	Number of Samples 250k			

Figure 5.7: Local Latency Comparison - C Vs Java SR

End-to-End latency values and corresponding local delays at the SR were plotted for 500 sample points to better understand the system behavior. The graphs for the same can be seen in the Figures 5.8 and 5.9.

Local delays at the C SR contributed only $\approx 15\%$ to the overall latency while for the Java SR, more than half ($\approx 60\%$) of the average end-to-end latency came from the local delays in the

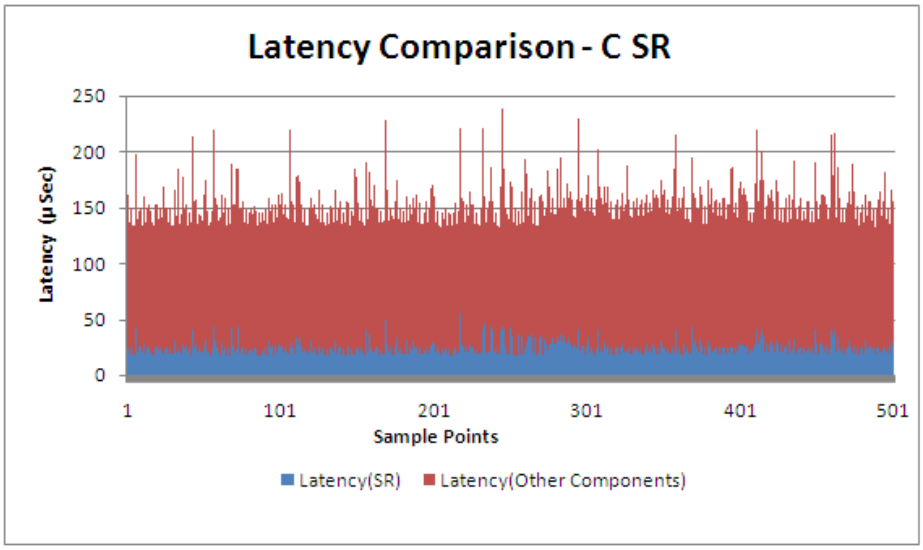

Figure 5.8: Local Vs End-To-End Latency Comparison - C SR

Java SR. There is a noticeable difference ($\approx$125μS) between the end-to-end latency values and the corresponding local delays at the SR (for both C and Java SR). A possible explanation for this is given below.

End-to-End latency is a collective sum of the delays caused by the OS (sending the packet from the publisher application to the physical link and the physical link to the SR or the subscriber application), network delays (time taken for the packet to reach from one endpoint to another endpoint), queuing delays at the SR and the delay within the SR (forwarding the event). The publishing interval for the experiment is 10ms, while the time taken by the SR to forward any event is in the order of μS with few exceptions. Therefore the *receiving thread* of the SR will always be waiting for the event to arrive even before the event is published. Consequently there will be no queuing of events on the incoming channel of the SR and hence queuing delays can be ruled out. Since a high speed Ethernet connects the hosts where the publisher/subscriber and the SR are running, delays because of network components can also be ignored. The only unaccounted component contributing to the end-to-end delay is the OS overhead which seems to be the most probable cause for the added latency. A support for this reasoning also comes from the Figure 4.8.

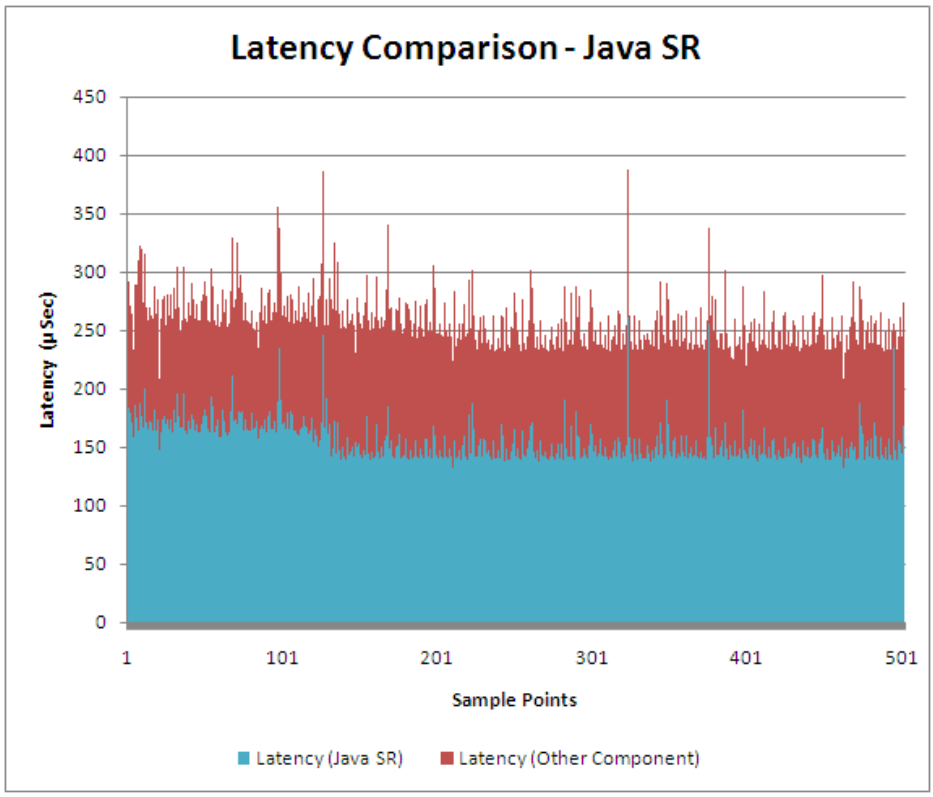

Figure 5.9: Local Vs End-To-End Latency Comparison - Java SR

The average interrupt latency for a Linux kernel 2.4.17[1] with the real-time patch is $52\mu S$. In this particular experiment, there are two network interrupts of interest. The first one is generated when the packet reaches the host running the SR from the publisher and the second one when the packet reaches the host running the subscriber from the SR, requesting the OS to schedule the SR and the subscriber application respectively. These two interrupts have an average latency overhead of $104\mu S$ (=52*2) which comes close to the magic value of 125μS.

Possible enhancements to reduce the latency overhead due to the OS are suggested in the *Future Work* under Section 6.2.2.

[1] the latency patch results are also applicable to Linux kernel 2.6.24, which was used for the experiments

5.2.2 Experiment to compare OS real-time capabilities

Section 4.3.3 explains the various real-time support/extensions in the Linux OS such as the *real-time priority threads* and *memory lock* to avoid paging. This experiment tries to document the effect of these features on the end-to-end latency. The topology of the experiment is the same as that of the Experiment 5.2.1. The experiment is run under three different configurations of the C SR

- Without using any Real-Time Threads or Memory Lock in the SR - "No RT Thread + No Mem Lock"
- Using Real-Time Threads but no Memory Lock - "RT Thread + No Mem Lock"
- Real-Time Threads and Memory Lock - "RT Thread + Mem Lock"

Analysis of Results

The effect of these different features on the local delay at the C SR is shown in the Figures 5.10, 5.11 and 5.12.

It can be clearly seen that real-time threads and memory locks have a beneficial impact on the performance of the SR. The maximum latency drops down from $4257\mu S$ on no real-time support to $266\mu S$ on using real-time threads to $63\mu S$ on using both real-time threads and memory locks. Also there is an $\approx 35\%$ decrease in average delay as you move from no real-time support to having all the real-time features. Reasoning for this can be found in Section 4.3.3. To summarize, usual (non real-time) threads are at the mercy of the scheduler to be scheduled when all other higher priority tasks are finished. Because of the undeterministic execution times of the higher priority tasks, the average latency increases. This also makes it difficult to limit the maximum latency, hence the peak is observed at $\approx 4ms$. When real-time threads are used, the scheduler has to give them priority over lower priority tasks. Since real-time threads are scheduled without any delay on reception of

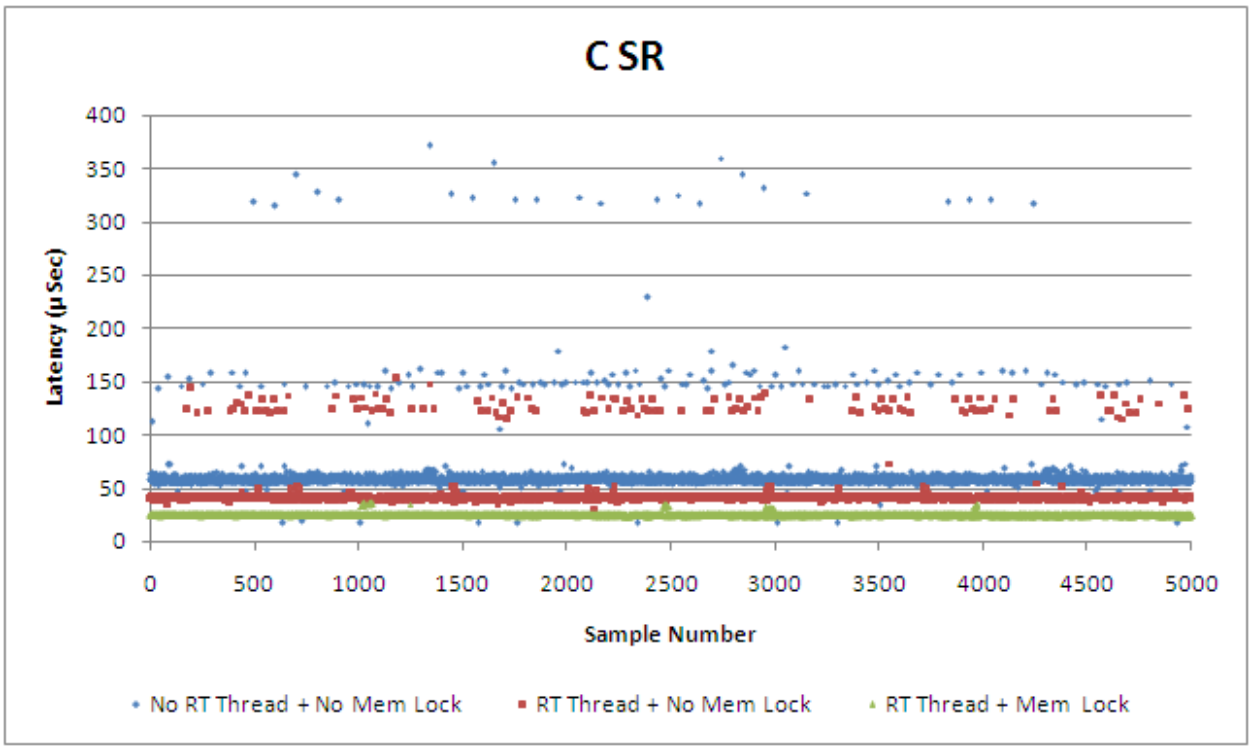

Figure 5.10: Local Latency Comparison - C SR with Different Configuration

an event, the average and maximum ($266\mu S$) latency drops down.

When real-time threads are used without any memory locks, there is a jitter in the recorded latency values, *97.317%* within *1-50*μS and *2.607%* within the range *101-150*μS. Memory lock helps containing this jitter by preventing paging and hence restricting *99.2%* of the values between *21-30*μS.

5.2.3 Load Scalability Experiment

The purpose of this experiment is to observe and analyze the behavior of the SR under varying load conditions. The topology of the experiment is depicted in Figure 5.13. *Pub0-Sub0* and *Pub2-Sub2* form the reference system across which the end-to-end latency is measured. *Pub1-Sub1* and *Pub3-Sub3* constitute the load system each providing half of the total load.

Range(µS)	# Samples	Percentage
11-20	1	0.003
21-30	29761	99.203
31-40	234	0.780
41-50	3	0.010
61-70	1	0.003
Total	30000	
RT Thread + Mem Lock		

Range(µS)	# Samples	Percentage
1-50	29195	97.317
51-100	18	0.060
101-150	782	2.607
151-200	3	0.010
201-250	1	0.003
251-300	1	0.003
Total	30000	
RT Thread + No Mem Lock		

Range(µS)	# Samples	Percentage
1-50	164	0.547
51-100	29015	96.717
101-150	420	1.400
151-200	256	0.853
201-250	4	0.013
251-300	2	0.007
301-350	115	0.383
351-400	18	0.060
1801-1850	2	0.007
4101-4150	2	0.007
4151-4200	2	0.007
Total	30000	
No RT Thread + No Mem Lock		

Figure 5.11: Local Latency Distribution Table - C SR with Different Configuration

	Latency (µSec)			
C SR Configuration	Average	Min	Max	Std. Dev. (σ)
RT Thread + Mem Lock	24	19	63	1.054
RT Thread + No Mem Lock	42	29	266	14.173
No RT Thread + No Mem Lock	62	18	4157	54.314
Number of Samples 30k				

Figure 5.12: Summarizing Latency Comparison - C SR with Different Configuration

Analysis of Results

Different parameters such as end-to-end latency, throughput and system load are used to analyze and quantify the different aspects of the system. Average, min, max and standard deviation of end-to-end latency for the C SR under varying degrees of load is given in the Figure 5.14. End-To-End latency distribution is given by the Figure 5.15

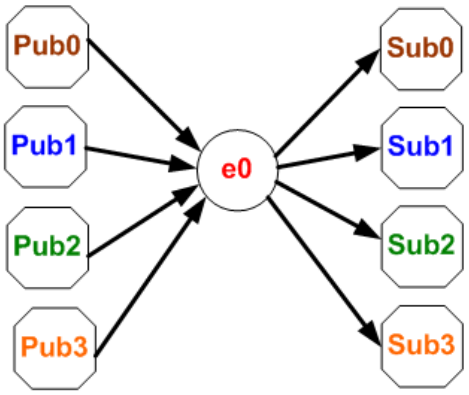

Figure 5.13: Topology for the Load Scalability Experiment

	Latency (µSec)			
Load Var.	Average	Min	Max	Std. Dev. (σ)
0	166.21	135	1030	34.38
60	166.16	130	1272	38.00
120	165.82	134	1305	37.79
180	167.32	132	1355	38.85
Number of Samples 250k				

Figure 5.14: Summarizing Latency Values for Reference Variables - C SR

End-To-End Latency

As it can be observed from the Figure 5.14, there is little or no effect of increasing load on the end-to-end latency values. There is a very slight increase in average (1.11 μS, from *166.21* to *167.32*), max (325 μS, from *1,030* to *1,355*) and standard deviation (4.37 μS, from *34.38* to *38.85*), from no load to maximum load. This can be reasoned out by looking at Figure 5.16 which captures the system load in terms of the percentage of CPU used. The system load increases from $\approx$*1%* under no load to $\approx$*17%* under full load. Still a significant($\approx$*83%*) percentage of the CPU is unutilized, which can handle considerable load before melting down.

A more detailed analysis of the standard deviation can be done through the latency distribution

	Load Variables			
Latency (µSec)	0	60	120	180
0-100	0.0000	0.0000	0.0000	0.0000
101-200	91.1184	88.1868	88.3190	86.7220
201-300	8.2884	10.7444	10.6169	12.1152
301-400	0.2304	0.7204	0.7113	0.8424
401-500	0.3172	0.2972	0.3036	0.2700
501-600	0.0048	0.0060	0.0072	0.0056
601-700	0.0000	0.0000	0.0020	0.0012
701-800	0.0000	0.0000	0.0008	0.0016
801-900	0.0000	0.0004	0.0004	0.0008
901-1000	0.0212	0.0328	0.0296	0.0304
1001-1100	0.0196	0.0104	0.0084	0.0088
1101-1200	0.0000	0.0000	0.0000	0.0004
1201-1300	0.0000	0.0012	0.0000	0.0012
1301-1400	0.0000	0.0000	0.0008	0.0004
Number of Samples 250k				

Figure 5.15: End-To-End Latency Distribution for Reference Variables - C SR

	CPU Load	
Load Var.	C SR	Java SR
0	0.7-1.3	0.1-0.5
60	2.0-5.0	28-35
120	8.0-12.0	52-58
180	15-17	59-65

Figure 5.16: CPU Load

given by the Figure 5.15. The end-to-end latency values are a little more wide spread under maximum load (180 load variables) than the concentrated distribution under no load. *99.41%* of the values are contained within the range of 100-300μS under no load, in comparison to a marginally lesser *98.84%* on maximum load. A different experiment to define the throughput of the SR is conducted and explained in the *Maximum Throughput/Load Experiment* under section 5.2.6.

5.2.4 Hop Scalability Experiment

In this particular experiment, the ability of the SR to scale linearly with additional SR in the forwarding path and with varying load is tested. The expected behavior on adding SR's along the path of an event from a publisher to a subscriber is that the end-to-end latency should show a linear

increase in proportion to the number of SR's added. As also seen from the *load scalability experiment* that additional load has little or no impact on the performance on the SR, similar behavior is expected from this experiment. The topology of the experiment is show in Figure 5.17. *Pub0-Sub0* form the reference system across which the end-to-end latency is measured. *Pub1-Sub1* and *Pub2-Sub2* constitute the load system and each of them provide half of the total load.

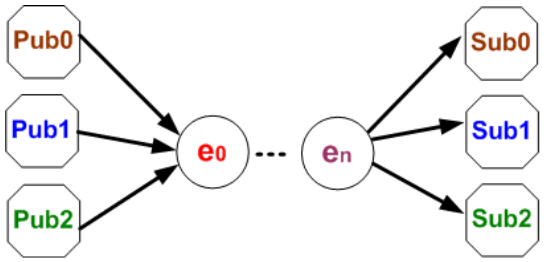

Figure 5.17: Topology for the Hop Scalability Experiment

Analysis of Results

A summary of the latency values for C SR is given in Figure 5.18 and for Java SR in Figure 5.19. The end-to-end latency distribution for C SR is given in Figure 5.20 and for Java SR in Figure 5.21. Finally the system load in terms of the % of CPU used is given in Figure 5.22..

As seen from the Figure 5.18 and in line with the expectations, the latency grows linearly with the number of SR and that increasing load has inconsequential impact on the average latency. The increase in average increase with each additional SR is about *90-100*μS. Also the maximum observed latency is *1,488*μS which is only marginally above the general average (of maximum) of $\approx$*1,300*μS. Surprisingly the maximum latency does not show any trend as we move from 1SR to 4SR and 0 to 120 load variables. On the other hand, the standard deviation and the average show a monotonic increase in value with additional number of SR in the forwarding path and increasing load. The smallest (*25.64*μS) standard deviation is observed with 1SR under no load and the largest (*65.92*μS) with 4SRs and 120 load variables. This is an expected behavior because as the

	C SR							
	Latency (µSec)							
	0 Load Variables				40 Load Variables			
Latency (µ Sec) =>	Average	Min	Max	Std. Dev. (σ)	Average	Min	Max	Std. Dev. (σ)
1 SR	152.16	128	1042	25.64	173.15	134	1331	43.93
2 SR	251.72	208	1108	36.21	261.58	210	1374	48.54
3 SR	340.28	289	1039	38.88	353.62	291	1358	57.83
4 SR	448.42	376	1488	42.31	447.45	369	1307	61.07
	80 Load Variables				120 Load Variables			
Latency (µ Sec) =>	Average	Min	Max	Std. Dev. (σ)	Average	Min	Max	Std. Dev. (σ)
1 SR	168.68	136	1344	37.68	178.05	137	1368	48.06
2 SR	270.34	204	1349	52.84	269.42	215	1005	51.06
3 SR	358.09	292	1158	61.45	366.89	295	1306	64.76
4 SR	460.32	377	1388	62.63	475.37	383	1392	65.92
	Number of Samples 250k							

Figure 5.18: Summarizing Latency Values for Reference Variables - C SR

	Java SR			
	Latency (µSec)			
	4ESR			
Latency (µ Sec) =>	Average	Min	Max	Std. Dev. (σ)
0 Load Variables	890.92	758	49122	788.24
120 Load Variables	3138.65	816	141703	5094.83
	Number of Samples 250k			

Figure 5.19: Summarizing Latency Values for Reference Variables - Java SR

load is increased packets start arriving while the previous packets are being processing. Other forms of processing within the kernel such as the Interrupt Service Routine (ISR) servicing the newer packets also start contributing to the latency of the previous packets. This adds to the spread in the latency values with increasing load.

A comparison of the results from the Java and the C SR under no load conditions for 4SR (Figure 5.18 and 5.19) shows that the average latency of the C SR is approximately half of that of the Java SR and the maximum latency in C SR is $\approx$*1.5 ms* while that for Java SR it is $\approx$*50 ms*.

Received (%)								
	0 Load Var				40 Load Var			
Latency(µ S	1ESR	2ESR	3ESR	4ESR	1 ESR	2 ESR	3 ESR	4 ESR
0-100	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
101-200	96.5040	0.0000	0.0000	0.0000	85.0692	0.0000	0.0000	0.0000
201-300	3.3972	91.7568	4.7144	0.0000	13.2808	86.3020	1.4436	0.0000
301-400	0.0008	7.7608	87.5572	9.8286	1.1700	11.7616	82.5872	19.1408
401-500	0.0580	0.4332	7.5344	79.8724	0.3500	1.5684	13.6180	66.5200
501-600	0.0000	0.0076	0.1496	9.9500	0.0368	0.1792	1.8896	12.2860
601-700	0.0000	0.0000	0.0052	0.3019	0.0268	0.0692	0.2344	1.4900
701-800	0.0004	0.0000	0.0004	0.0073	0.0180	0.0488	0.0744	0.2388
801-900	0.0000	0.0004	0.0000	0.0006	0.0072	0.0200	0.0628	0.1044
901-1000	0.0332	0.0224	0.0208	0.0079	0.0216	0.0280	0.0572	0.1208
1001-1100	0.0064	0.0184	0.0180	0.0304	0.0172	0.0216	0.0272	0.0656
1101-1200	0.0000	0.0004	0.0000	0.0006	0.0012	0.0004	0.0052	0.0240
1201-1300	0.0000	0.0000	0.0000	0.0000	0.0008	0.0000	0.0000	0.0092
1301-1400	0.0004	0.0000	0.0000	0.0000	0.0004	0.0008	0.0004	0.0004
1401-1500	0.0000	0.0000	0.0000	0.0006	0.0000	0.0000	0.0000	0.0000
	80 Load Var				120 Load Var			
Latency(µ S	1ESR	2ESR	3ESR	4ESR	1 ESR	2 ESR	3 ESR	4 ESR
0-100	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
101-200	89.1080	0.0000	0.0000	0.0000	81.4631	0.0000	0.0000	0.0000
201-300	9.9588	82.1040	1.6232	0.0000	16.1665	82.4248	0.1812	0.0000
301-400	0.5712	14.9472	79.9632	8.8612	1.7224	15.0504	76.5536	1.7428
401-500	0.2460	2.4264	15.3904	71.9120	0.5064	2.0784	19.2060	71.9549
501-600	0.0332	0.3180	2.4548	16.4732	0.0412	0.2392	3.4084	22.0239
601-700	0.0308	0.0744	0.3328	2.1188	0.0312	0.0836	0.3924	3.4592
701-800	0.0076	0.0560	0.0712	0.3076	0.0180	0.0596	0.0852	0.4848
801-900	0.0028	0.0260	0.0684	0.0940	0.0052	0.0240	0.0620	0.0912
901-1000	0.0148	0.0284	0.0656	0.1060	0.0212	0.0352	0.0628	0.1148
1001-1100	0.0260	0.0188	0.0256	0.0924	0.0244	0.0048	0.0424	0.0796
1101-1200	0.0004	0.0004	0.0048	0.0232	0.0000	0.0000	0.0044	0.0368
1201-1300	0.0000	0.0000	0.0000	0.0084	0.0000	0.0000	0.0012	0.0096
1301-1400	0.0004	0.0004	0.0000	0.0032	0.0004	0.0000	0.0004	0.0024
1401-1500	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Number of Samples 250k

Figure 5.20: End-To-End Latency Distribution for the Hop Scalability Experiment: C SR

The effect of increasing load on the Java SR is shown in the Figure 5.19. From the figure it can be easily inferred that load has a detrimental effect on the behavior of the Java SR. While the average

	Received (%)	
Latency(μ Sec)	4ESR - 0 Load	4ESR - 120 Load
701-800	21.5364	0.0000
801-900	67.8048	1.2516
901-1000	7.5232	2.4148
1001-1100	1.1612	3.7476
1101-1200	0.2868	4.1580
1201-1300	0.2348	4.8096
1301-1400	0.1220	4.7292
1401-1500	0.0696	5.3400
1501-1600	0.0428	5.1152
1601-1700	0.0360	5.6664
1701-1800	0.0304	5.9168
1801-1900	0.0372	5.0872
1901-2000	0.0252	5.0820
2001-2100	0.0240	4.8636
2101-2200	0.0272	4.5468
2201-2300	0.0612	3.9384
2301-2400	0.1076	3.3900
2401-2500	0.1364	2.9912
2501-2600	0.0488	2.4156
2601-2700	0.0540	1.9216
2701-2800	0.0620	1.5844
2801-2900	0.0600	1.2528
2901-3000	0.0628	1.0108
3001-3500	0.0936	4.2380
3501-4000	0.0444	2.5440
4001-4500	0.0336	1.9896
4501-5000	0.0228	1.3396
5001 +	0.2512	8.6552
Number of Samples	250k	

Figure 5.21: End-To-End Latency Distribution for the Hop Scalability Experiment: Java SR increases from $\approx 900\mu$S to $\approx 3{,}000\mu$S, the maximum shoots to an unacceptable $\approx 150{,}000\mu$S. Similar experiment which was carried out in [29] on the Java SR observed an average latency of *1.999 ms* with 4 SR and 120 load variables [2]. Here the observed average latency for the Java SR for 4 SR and 120 load variables is *3.138 ms*. The differences in the two values could be because of the

[2] refer Table 8.12 in [29]

	CPU Load	
Load Var.	C SR	Java SR
0	0.7-1.3	3.0-5.0
60	2.0-5.0	30-35
120	8.0-12.0	40-46
180	15-17	55-60

Figure 5.22: CPU Load for the Hop Scalability Experiment

differences in the hardware specifications of the machines used in these experiment setups. Performance of the SR has a high dependency on the computing power of the host machine. Under such circumstances a better performance of the Java SR in [29] than here is understandable because of the use of more powerful machines for the experiments in [29].

The latency distribution Figures of 5.20 and 5.21 are also in line with the expectations. As the load increases, the latency values get more evenly divided in comparison to the highly concentrated values under no load. For C SR, the highest percentage of values within any range under no load is *96.5040%* (1SR, 0 load, 101-200μS) but under full load the highest is much lesser at *81.4631%* (1SR, 120 load, 101-200μS). Java SR shows similar behavior with the exception that the curve under full load becomes very flat and the latency values are almost evenly divided from *1ms - 3ms*. The highest percentage of values within any range for Java SR with no load is *67.8048%* (4SR, 0 load, 801-900μS) in comparison to *5.9168%* (4SR, 120 load, 1701-1800μS) under full load.

5.2.5 Multicast Mechanism Experiment

The purpose of this experiment is to test the effect of multicast on the performance of the SR. The experiment topology is similar to the *hop scalability experiment* except that each of the SR in this experiment has a fan out of 2 instead of 1. The topology of the experiment is shown in Figure 5.23. *Pub0* and *Sub0* form the reference system across which the end-to-end latency is measured. *Pub1* and *Pub2* are the load publishers, each providing half of the total load. Four load subscribers (Sub1 to Sub4), each subscribe to both *Pub1's* and *Pub2's* load variables.

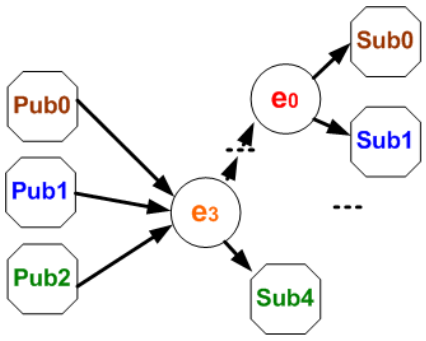

Figure 5.23: Topology for the Multicast Mechanism Experiment

Analysis of Results

A summary of the different attributes of the latency values is presented in the Figure 5.24. End-To-End latency distribution and system load in terms of % of CPU used is given in Figure 5.25.

C SR

	Latency (µ Sec)			
	4ESR			
Latency (µ Sec) =>	Average	Min	Max	Std. Dev. (σ)
0 Load Variables	448.48	376	1128	42.45
120 Load Variables	648.13	391	1971	159.91

Java SR

	Latency (µ Sec)			
	4ESR			
Latency (µ Sec) =>	Average	Min	Max	Std. Dev. (σ)
0 Load Variables	890.92	758	49122	788.24
120 Load Variables	21715.25	922	1382078	97224.13
	Number of Samples 250k			

Figure 5.24: Summarizing Latency Values for the Multicast Mechanism Experiment

For C SR, the average latency only increases by $\approx 200\mu$S even on substantial increase in load. Conversely, the maximum latency shows a much steeper increase of ≈ 1ms from ≈ 1ms to ≈ 2ms. The standard deviation shows analogous trend as the maximum latency.

Java SR

	Received (%)	
	4 SR	
Latency (µ Sec)	0 Load	120 Load
701-800	21.7364	0.0000
801-900	66.6024	0.0000
901-1000	8.5256	1.2768
1001-1100	1.1612	0.7920
1101-1200	0.2868	0.4528
1201-1300	0.2348	0.7256
1301-1400	0.1220	1.3136
1401-1500	0.0696	2.0000
1501-1600	0.0428	2.8200
1601-1700	0.0360	3.5000
1701-1800	0.0304	4.1072
1801-1900	0.0372	4.7792
1901-2000	0.0252	4.9960
2001-2100	0.0240	5.3944
2101-2200	0.0272	5.7272
2201-2300	0.0612	5.6144
2301-2400	0.1076	5.3760
2401-2500	0.1364	4.8864
2501-2600	0.0488	4.6800
2601-2700	0.0540	4.3200
2701-2800	0.0620	3.8632
2801-2900	0.0600	3.2864
2901-3000	0.0628	2.9120
3001-3100	0.0320	2.6464
3101-3200	0.0240	2.2064
3201-3300	0.0128	1.8816
3301-3400	0.0104	1.4344
3401-3500	0.0144	1.2448
3501-4000	0.0444	3.6424
4001-4500	0.0336	1.6608
4501-5000	0.0228	0.9408
5001 +	0.2512	11.5192
Number of Samples	250k	

C SR

	Received (%)	
	4 SR	
Latency (µ Sec)	0 Load	120 Load
301-400	9.9028	0.1985
401-500	78.7404	17.4713
501-600	10.9776	24.4853
601-700	0.3296	23.3217
701-800	0.0080	19.5183
801-900	0.0008	9.9060
901-1000	0.0096	2.9864
1001-1100	0.0304	0.8971
1101-1200	0.0008	0.4289
1201-1300	0.0000	0.2833
1301-1400	0.0000	0.1736
1401-1500	0.0000	0.1320
1501-1600	0.0000	0.0888
1601-1700	0.0000	0.0544
1701-1800	0.0000	0.0216
1801-1900	0.0000	0.0240
1901-2000	0.0000	0.0088
Number of Samples	250k	

	CPU Load	
Load Var.	C SR	Java SR
0	0.7-1.3	3.0-5.0
120	25-30	75-80

Figure 5.25: End-To-End Latency Distribution for the Multicast Mechanism Experiment

The performance of the Java SR degrades drastically while moving from no load to full load. The maximum latency crosses the *seconds* boundary, touching $\approx$*1.4 Seconds* and the standard deviation is about *5* times the average latency of $\approx$*21 ms*. From Table 8.21 in [29], it can bee seen

that the average latency for 4 SR drops from *1.366 ms* for 20 load variables to *6.476 ms* for 120 load variables. Though the average latency of ≈*6 ms* as seen in [29] is much less than the average latency of ≈*21 ms* for 4 SR, 120 load variables seen here, similar degradation in the performance of the Java SR can be seen in both the experiments. Reasons for the exceptional behavior of the Java SR is debated under *System Load Analysis* in Section 8.5.1 of [29].

5.2.6 Maximum Throughput/Load Experiment

The purpose of this experiment is to push the SR to its limit in order to observe the maximum throughput that the SR can provide or the maximum load that it can sustain. The experiment gives an insight into the performance boundaries of the SR. The effect of adding a publisher/subscriber to a SR in terms of average and maximum latency and CPU Load is observed. The topology of the experiment is shown in Figure 5.26.

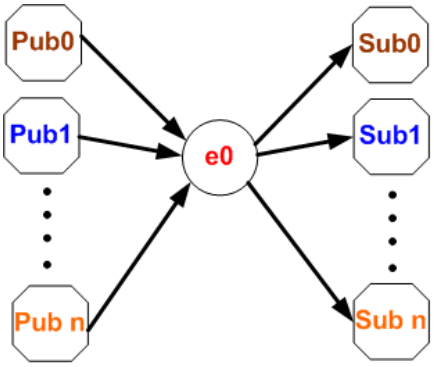

Figure 5.26: Topology for the Maximum Throughput/Load Experiment

Only Pub0-Sub0 form the reference system across which the end-to-end latency is measured. All other publishers/subscribers form a part of the load system. The publishing and subscribing interval for both reference and load systems is 1ms. Therefore each publisher/subscriber contribute to 1000 events in every second. After adding a publisher/subscriber, the system is allowed to run for some time and latency and CPU Load values are measured. This is carried on till the CPU load on the machine on which the SR is running becomes very high and the CPU becomes irresponsive

to other activities. For example, if the figures for 1000 forwards/second were to be taken from the experiment, then only the reference system will be used without any load pub/sub. In general, for taking values for n*1000 forwards/second, there will be (n-1) load pub/subs and the reference pub/sub. The reference pub/sub run on a single host with no load pub/sub. The load pub/sub were divided into two host machines. Such a setup was chosen because of the limited availability of machines with patched Linux kernel.

Analysis of Results

The average and maximum latency trend on increasing load can be observed in the Figure 5.28. The CPU load as observed on the host where the SR was running is represented in the Figure 5.29. The effect of incremental load on the C and Java SR in terms of the latency and the CPU load is summarized in Figure 5.27.

As seen from the table and the figures, both the latency and CPU load show a linear growth with each additional publisher/subscriber. The system is able to handle around 10k forwards/second without much degradation in performance. The average latency increases by $\approx$50%(158μS to 236μS) while moving from 1k forwards/sec to 8k forwards/sec. Once the CPU load crosses the midway mark of 50%, the latency values show a much steeper slope. The system load displays a much more linear growth. The CPU load increases by $\approx$4% with the addition of each publisher/subscriber. The values and figures shown in the table are indicative of the performance of the SR on this test machine and may vary with the CPU capabilities.

The Java SR is only capable of handling around 6k forwards/seconds, beyond which on increasing load, packets were lost. From the data collected at the subscriber with 7-11 load pub/sub, it could be seen that the Java SR was able to only forward packets from 5-6 publishers within a millisecond and packets from other pub/subs were dropped. Because of this reason, the latency values for the Java SR in the Figure 5.27 are shown only till 6000 forwards/second. An important difference to note in the throughput figures cited in the various experiments within [29] and

C SR

Forwards in k/Sec	Latency (µS) Average	Max	CPU %
1	158.89	790	3.2
2	173.28	813	6.7
3	178.28	792	11.3
4	186.49	1106	15.4
5	197.87	1153	19.8
6	207.39	1200	24.5
7	223.21	1212	28.8
8	236.34	1193	33.7
9	259.18	1196	38.3
10	283.52	1303	41.2
11	328.73	1573	47.6
12	367.26	1890	51.7
13	392.15	2200	55.1
14	410.34	2183	60.8
15	453.23	2217	65.7
16	476.72	2274	69.2
Number of Samples 250k			

Java SR

Forwards in k/Sec	Latency (µS) Average	Max	CPU %
1	303.75	27205	22
2	345.01	35947	28
3	387.62	42150	36
4	401.34	45301	44
5	425.34	45890	50
6	440.14	47391	60
Number of Samples 250k			

Figure 5.27: Summary Table and System Load

here is that in [29], the throughput is presented as events/ms and here the throughput is given in forwards/sec or packets/sec. The difference between the two is that a packet can contain one or more events. So in an experiment setup with n events packed within a packet, a throughput of x packets/sec is equivalent to x*n events/sec. For the experiments in [29], each packet contained 4 variables[3] or events and for the experiments conducted for this thesis, each packet contains only one variable or event.

[3] refer to the Section 8.1 in [29]

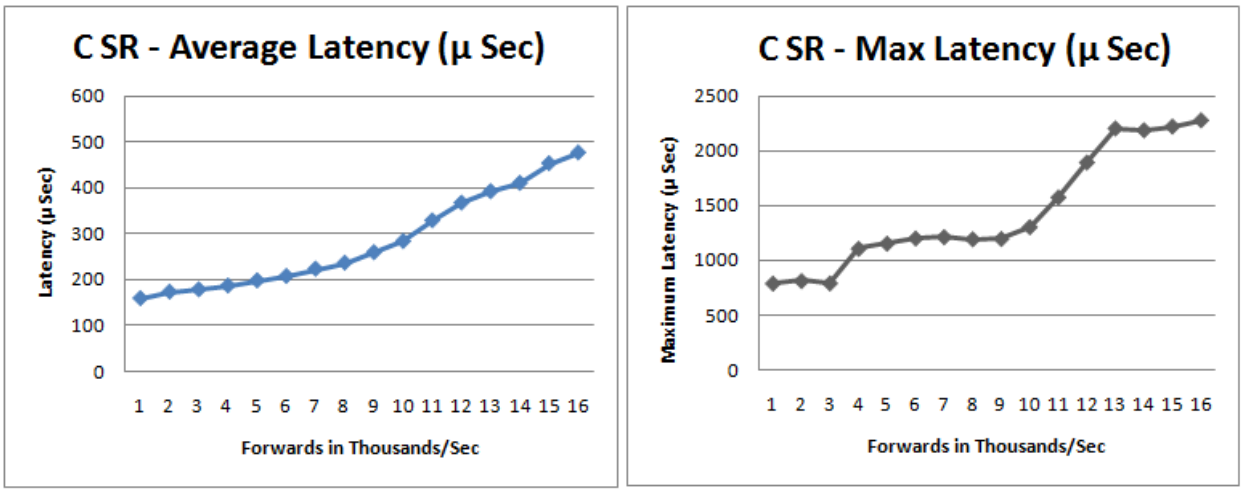

Figure 5.28: Average and Maximum Latency Graphs - C SR

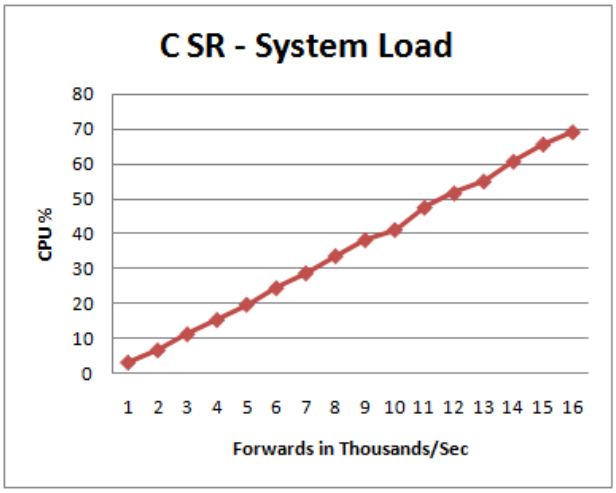

Figure 5.29: CPU Load - C SR

5.2.7 Scheduling Mechanism Experiment

Delay EDD and FIFO scheduling algorithm were implemented as kernel modules. The experiment below evaluates these two algorithms on the end-to-end latency performance. The experiment

setup involves 12 load flows and a reference flows. The load flows provide the load for the experiment and end-to-end latency is measured across the reference flows. The load flows are publishing a variable every 10ms with the end-to-end delay bound of 20ms while the reference flow publishes a variable every 1ms with the end-to-end delay bound of 5ms.

Analysis of Results

A summary of the latency values for the reference flow is given below in Figure 5.30. From the table it can be easily seen that the Delay-EDD is able to contain the latency for the reference flow well below the end-to-end delay bound of 5ms. On the other hand, the maximum latency observed for FIFO is ≈39 ms, which is well beyond the permissible limit of 5ms.

	Latency (µS)			
Scheduling Discipline	Min	Max	Average	Std. Dev.
Delay EDD	191	2449	741	249
FIFO	4767	39056	5263	266
Number of Samples 250k				

Figure 5.30: Summarizing Latency Values for Reference Flow

CHAPTER SIX

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The system implementation and other mechanisms presented in this thesis addresses the need for a real-time content based application router. Application routers are presently widely used within distributed systems such as overlay networks. [14] proposes the need for a wide area distributed middleware system named GridStat to cater to the emerging needs of the communication infrastructure of the electric power grid (EPG). GridStat provides a QoS managed publish-subscribe system to cater to the differential QoS requirements of the end-user such as EPG Applications. The present real-time routing within GridStat lacked optimal implementation and failed to provide bounded end-to-end delays.

This thesis explores the current limitations of the Java application router in GridStat. It discusses the reason behind these limitations and provides a rational for choosing a more suitable implementation language (C/C++) for implementing a real-time application router. These rationals are then backed up by enumerating the requirements from a real-time application router from the perspective of the communication infrastructure of the EPG. To meet these requirements, a system implementation of the router is presented. This system implementation exploits the various real-time support from the operating system such as priority scheduling and real-time threads. The thesis then argues that mere system implementation of the router will not satisfy all the requirements as other operating constraints can induces run time overheads. Alternative mechanisms are explored to overcome such limiting constraints. One of them is to extend the operating system through real-time patches to provide maximum real-time support. Another mechanism that has to be in place to deal with network congestion and slow network devices is appropriate scheduling algorithm for the network, at the appropriate location. Based on the work done by [33], it is clear

that Delay-EDD is a scheduling algorithm which can achieve the bounded delay requirements of GridStat and EPG in general. A scheduling algorithm can only deliver optimal results if placed at the most optimal location on a network stack. The thesis then argues that the most optimal location for Delay-EDD scheduling algorithm for a distributed system using UDP as a communication protocol is below the network stack and above the physical layer. Such an implementation would require communication across the user-kernel space boundary. This is achieved through a set of additional system calls that were added to the Linux kernel. These mechanisms and methodology used to achieve the QoS within the router can be easily applied to the specific QoS needs of similar real-time applications.

The experiments on the SR evaluated its ability to scale under high load and with respect to the number of SR's between the publishers and the subscribers. The multicast experiment evaluated the ability of the SR to efficiently deliver the same status event to different subscribers with minimum impact on the latency. From the results obtained from the experimental evaluation of the SR it can be concluded that the C/C++ SR has the capability and the capacity to achieve low average delay and a tight upper bound latency even under heavy load and in a multicast deployment. The results from the Delay-EDD scheduling algorithm experiment shows that Delay-EDD scheduler complements the SR and enables the SR to achieve the desired QoS under the operating boundaries of heavy loads, network congestion and on slow network devices.

6.2 Future Work

6.2.1 Port the SR to other OS such as Solaris and RTOS

Presence of numerous OS in the market today provides users the flexibility to choose an OS that meets most of their requirements. Implementing the SR across multiple OS and hardware platforms will increase its deployment scope.

The present implementation of the SR uses POSIX and BSD Socket API's to communicate with the OS. UNIX based OS have widely accepted these API's as a standard. This makes the SR

easily portable across OS such as Solaris and Berkeley Unix, with minimal efforts. On the other hand, RTOS have very specific architectures pertaining to the application needs. Porting the SR to these RTOS will require more efforts and OS specific changes.

6.2.2 Event Forwarding algorithm as a Kernel Module

Modules residing with the kernel have closer access to the system resources including the CPU (Central Processing Unit), than a user application. Kernel also provides hooks for modules to attach themselves at various accessible points. One such beneficial location is on top of the network Interrupt Service Routine (ISR) and below the network protocol stack. Network ISR is the first packet managing code that is invoked on reception of a network packet.

The closest a module can get to a network packet while staying network card vendor independent is by placing itself on top of the ISR and below the kernel network protocol stack. The network ISR will invoke the attached kernel module before passing the packet on to the kernel. The kernel module can choose to drop the packet based on some business logic and save further CPU cycles on the network packet. In the context of the SR, this business logic is represented by the event forwarding code or the routing algorithm. Having the routing algorithm implemented as a kernel module will bring down the latencies significantly.

Business logic usually reside within an application and tend to be dynamic for a distributed system. Hence, a communication channel using system calls has to be established between the user application or the SR in this case and the kernel module. But system calls are inflexible and were not designed to convey business logic. Also a faulty kernel module can easily crash the whole system and hence, applications as kernel modules are strongly discouraged. But the performance gains of having the event forwarding code as a kernel module are significant enough to justify at least a feasibility study. A good reference to start with will be [40].

6.2.3 Security Aspects of the SR

The present implementation of the SR takes measures to prevent exploitable buffer overflow vulnerabilities and unpredictable behaviors due to malformed packets. But this does not provide end-to-end security for the SR. It is still susceptible to numerous network attacks such as SYN flooding, smurfing, distributed denial-of-service, identity spoofing and routing attacks [13]. While some of these attacks can be easily prevented using firewalls, others might require a more complicated encrypted based solution or an intrusion detection system. Security solutions because of their intrinsic nature tend to have an impact on the throughput and latency. End-To-End security solution for the SR is a challenging problem because the solution has to ensure that the latency requirements are least affected.

APPENDIX

APPENDIX ONE

HOWTO: APPLYING REAL-TIME PATCH TO LINUX

This appendix lists down the steps required to apply the real-time patch to the Linux kernel. Steps to configure the kernel and apply the patch[35]

1. Install any latest version of Linux such as Ubuntu 7.10+, CentOS 5+. Lets say it is CentOS 5.0

2. Download the latest version of stable Linux kernel (2.6.18.3 or above) from `http://www.kernel.org/pub/linux/kernel/` and the corresponding RT Patch from `http://www.kernel.org/pub/linux/kernel/projects/rt/older/` to the directory /usr/src. Make sure the kernel version and the patch version matches. Lets say the kernel version is *2.6.24*, then the RT Patch should be 2.6.24-rt1

3. Extract the source

   - cd /usr/src
   - tar xjf linux-2.6.24.tar.bz2

4. Make a soft link

   - ln -s linux-2.6.24 linux

5. cd /usr/src/linux

6. Clean up previous stuff

   - make clean
   - make mrproper
   - cp /boot/config-‘uname -r‘ ./.config

7. Applying RT Patch

    - Dry-Run, *bzip2 -dc /usr/src/patch-2.6.24-rt1.bz2 | patch -p1 –dry-run*
    - If the Dry-Run completes without any error, apply the patch,
      bzip2 -dc /usr/src/patch-2.6.24-rt1.bz2 | patch -p1

8. Configuration

    - make menuconfig
    - Go into Load an Alternate Configuration File
    - Press OK
    - Enable realtime-preempt patch
        - Go into Processor type and feature menu
        - Go into Preemptible Kernel
        - Select Complete Preemption (Real-Time)
        - You will be back at the Processor Type and features menu (make sure the Preempt The Big Kernel Lock) is now selected
    - Exit and save the configuration

9. Build

    - make all
    - make modules_install
    - make install

10. The new kernel should appear in /boot/grub/menu.lst. Make sure the default (default = < Position of new kernel image in the menu.lst file >) is set to the new kernel.

11. reboot

12. After rebooting you can check if it is the new kernel by running uname -r It should display something like 2.6.24

The actual steps for building the kernel might vary with the flavor of Linux being used, but the steps for enabling the Realtime-preemption patch should remain the same. For compiling kernel on individual flavors refer

- Ubuntu: `http://www.howtoforge.com/kernel_compilation_ubuntu`
- CentOS: `http://www.howtoforge.com/kernel_compilation_centos`

The Realtime-preemptible and the Low Latency Patch has been merged as a single RT Patch post Linux Kernel 2.6. Also the O(1) scheduler has been incorporated as into the Linux Kernel 2.5. So if you choose to install a kernel version prior to 2.6, then these patches can be downloaded from
`http://www.kernel.org/pub/linux/kernel/people/rml/preempt-kernel/`.
For applying these patches refer `http://lowlatency.linuxaudio.org/`

APPENDIX TWO

HOWTO: ADD AND INTERCEPT SYSTEM CALLS IN LINUX

System call is a mechanism through which a user space application requests the service of the OS. Every system call is given a unique number within the OS. A system call is essentially an interrupt (INT 0x80h), which switches the hardware operating mode from normal user to super user. The super user mode in which the Linux kernel operates, provides direct access to the system hardware and also to a higher set of assembly instructions which are not available to the user space applications running in normal user mode. In INT 0x80h, the system call number is passed using the EAX register while the arguments of the system call are passed through other (EBX, ECX, etc.) registers. The kernel executes the system call on the applications behalf and returns the result to the application through a register. This appendix lists down the steps to implement a new system call in Linux 2.6. It also enumerates the implementation details to intercept the newly added system call through a Linux Kernel Module.

B.1 Steps to Add a System Call in Linux

Lets assume that the system call to be added has a prototype "*long registerFlow(unsigned short port, unsigned long addr)*". Following files have to be modified/created to support this new system call. Refer [19] for further details.

Kernel Files to be modified

- Untar any Linux 2.6.xx source in the directory /usr/src/.
- /usr/src/linux/arch/i386/kernel/syscall_table.S: This file contains the names of all the system calls. Add "*.long sys_registerFlow*" at the end of the list of all the system calls.
- /usr/src/linux/include/asm-i386/unistd.h: This file attaches a number to each system call.

  - Add "*#define __NR_mycall <Last_System_Call_Num + 1>*" at the end of the list. For example if the last system call defined was *#define __NR_move_pages 317*, the add *#define __NR_registerFlow 318*
  - Increment the "NR_syscalls" by 1. So, if NR_syscalls is defined as *#define NR_syscalls 318*, then change it to *#define NR_syscalls 319*.

- /usr/src/linux/include/linux/syscalls.h: This file contains the prototype of the system calls. Add *asmlinkage long sys_registerFlow(unsigned short port, unsigned long addr);* at the end of the file.

- /usr/src/linux/Makefile: Search for regular expression *core-y.**+=* in the file. Add *registerFlow/* to *core-y*. The registerFlow directory will contain the implementation of the system call.

Kernel Files to be created

- Create a new directory named */usr/src/linux/registerFlow*.

- Create a source file named *registerFlow.c* in the directory */usr/src/linux/registerFlow*. This file will contain the implementation of the *registerFlow* system call. This file should look like

```
#include <linux/linkage.h>
asmlinkage int sys_registerFlow(unsigned short port, unsigned long addr){
  //Add your implementation here
  return 0;
}
```

- Create a file named *Makefile* in the directory */usr/src/linux/registerFlow*. The Makefile will have only one line *obj-y := registerFlow.o*.

User Space Files to be created to test the new system call

- Create a header file called *testRegisterFlow.h*. This header file should be included by any program wishing to call the *registerFlow* system call. The file looks like

```
int registerFlow(unsigned short port, unsigned long addr){
  int     retval = -1;
  //EAX register contains the system call number
  asm("   movl    $318, %eax");
  //EBX has the first argument
  asm("   movl    %0, %%ebx " : : "m" (port) );
  //ECX has the second argument
  asm("   movl    %0, %%ecx " : : "m" (addr) );
  //Change to super user mode
  asm("   int     $0x80          ");
  //EAX has the return value.
  asm("   movl    %%eax, %0 " : "=m" (retval) );
  return  retval;
}
```

- Create a C file named *testRegisterFlow.c*. This file will invoke the *registerFlow* system call. The file looks like

```
#include <linux/linkage.h>
asmlinkage int sys_registerFlow(unsigned short port, unsigned long addr){
  //Add your implementation here
  return 0;
}
```

Steps to test the new system call

- Compile and install the new kernel. Details are available in Appendix ONE.
- Compile and execute the *testRegisterFlow.c* file.
- If you put any *printk* inside the *registerFlow* system call, it should show up on *dmesg* | *tail*.

B.2 Intercepting System Call through Linux Kernel Module (LKM)

The Linux kernel maintains a table of pointers to the functions implementing the various system calls called as the **System Call Table** (sys_call_table). Prior to Linux 2.5 kernels, a LKM could

easily access the sys_call_table structure by declaring it as an extern variable. Post Linux kernel 2.5 the sys_call_table structure is no longer exported and is kept in a read-only memory within kernel space. So any attempt to modify the sys_call_table structure within any LKM will lead to segmentation fault. The first step in intercepting a system call through a LKM is to export the memory location where the sys_call_table structure is stored. This can be done by either using the objdump utility or by looking for the symbol 'sys_call_table' in System.map file. After exporting the structure, the sys_call_table structure should be made modifiable by setting the page frame entry containing the structure 'writable'. Once the structure is writable, the process for intercepting a system call is the same as in any Linux kernel. The function pointer in the sys_call_table pointing to the system call has to be replaced with the pointer of the function intercepting the system call in *init_module*. Later during the *cleanup_module* the original function pointer has to be restored in the sys_call_table. Command for finding the memory location of the sys_call_table. Refer [21, 3, 22, 34] for further details.

```
[bash]$ grep sys_call_table /boot/System.map
c044fd00 D sys_call_table

                  or

[bash]$ cd /usr/src/linux
[bash]$ objdump -t vmlinux | grep sys_call_table
c044fd00 D sys_call_table
```

The code snippet below for the LKM uses the memory location from above to intercept the obsolete *break* system call [22].

```
//----------------------------------------------------------------------
//      newcall.c
//
//      This module dynamically installs a new system-call, without
//      the need to modify and recompile the kernel's source files.
//      It is based on 'patching' an obsolete entry in the kernel's
//      'sys_call_table[]' array.  The address of that array can be
//      discovered, either in the 'System.map' file for the current
//      kernel (normally located in the '/boot' directory), or else
```

```
//      in the uncompressed kernel binary ELF file ('vmlinux') left
//      behind in the '/usr/src/linux' directory after compilation.
//      Some recent kernel-versions place the 'sys_call_table[]' in
//      a 'read-only' page-frame, so we employ a 'workaround' here.
//
//      NOTE: Written and tested using Linux kernel version 2.4.17.
//
//      programmer: ALLAN CRUSE
//      written on: 22 JUN 2004
//      revised on: 01 DEC 2007 -- for Linux kernel version 2.6.22.5
//---------------------------------------------------------------------

#include <linux/module.h>       // for init_module()
#include <asm/uaccess.h>        // for copy_to/from_user()
#include <asm/unistd.h>         // for __NR_break
#include <asm/io.h>             // for phys_to_virt()

unsigned long *sys_call_table;
unsigned long save_old_syscall_entry;
unsigned long save_old_pgtable_entry;
unsigned int _cr4, _cr3;
unsigned int *pgdir, dindex;
unsigned int *pgtbl, pindex;

asmlinkage long my_syscall( int __user * num ){
        int     val;
        if ( copy_from_user( &val, num, sizeof( int ) ) ) return -EFAULT;
        ++val;
        if ( copy_to_user( num, &val, sizeof( int ) ) ) return -EFAULT;
        return  0;  // SUCCESS
}

static void __exit my_exit( void ){
        sys_call_table[ __NR_break ] = save_old_syscall_entry;
        pgtbl[ pindex ] = save_old_pgtable_entry;
}

static int __init my_init( void ){
        /*obtain sys_call_table from hardcoded value
        we found in System.map*/
    *(long *)&sys_call_table=0xc044fd00;
        printk( "(sys_call_table[] at %08X) \n", (int)sys_call_table );

        // get current values from control-registers CR3 and CR4
        asm(" mov %%cr4, %%eax \n mov %%eax, _cr4 " ::: "ax" );
        asm(" mov %%cr3, %%eax \n mov %%eax, _cr3 " ::: "ax" );

        // confirm that processor is using the legacy paging mechanism
        if ( (_cr4 >> 5) & 1 )
        {
                printk( " processor is using Page-Address Extensions \n");
```

```
                return  -ENOSYS;
        }

        // extract paging-table indices from 'sys_call_table[]' address
        dindex = ((int)sys_call_table >> 22) & 0x3FF;   // pgdir-index
        pindex = ((int)sys_call_table >> 12) & 0x3FF;   // pgtbl-index

        // setup pointers to the page-directory and page-table frames
        pgdir = phys_to_virt( _cr3 & ~0xFFF );
        pgtbl = phys_to_virt( pgdir[ dindex ] & ~0xFFF );

        // preserve page-table entry for the 'sys_call_table[]' frame
        save_old_pgtable_entry = pgtbl[ pindex ];

        printk("\nInstalling new function for system-call %d\n",__NR_break);
        pgtbl[ pindex ] |= 2;   // make sure that page-frame is 'writable'
        save_old_syscall_entry = sys_call_table[ __NR_break ];
        sys_call_table[ __NR_break ] = (unsigned long)my_syscall;
        return  0;  // SUCCESS
}

module_init(my_init);
module_exit(my_exit);
MODULE_LICENSE("GPL");
//-----------------------------------------------------------------
```

A Makefile to compile the LKM has to be created. It contains only one line 'obj-m += newcall.o'. To compile the LKM use the command 'make -C /usr/src/linux-‘uname -r‘ SUBDIRS=$PWD modules'. To insert and remove the LKM use the commands 'insmod newcall.ko' and 'rmmod newcall.ko' respectively. After inserting the LKM, use the code snippet below [22] to test whether the *break* system call was successfully intercepted. On each successful call, the value passed to the *break* system call should increment by 1.

```
//-----------------------------------------------------------------
//      try17.cpp
//
//      This program makes a direct call to the Linux kernel with a
//      request to execute the obsolete system-call number 17.  The
//      kernel no longer implements that particular kernel service,
//      so we will see no changes to our function's argument-value.
//
//      We will use this program to demonstrate that we can replace
//      that unimplemented system-call with our own kernel routine.
```

```
//
//              compile using:  $ g++ try17.cpp -o try17
//              execute using:  $ ./try17
//
//      programmer: ALLAN CRUSE
//      written on: 24 JUN 2004
//-----------------------------------------------------------------

#include <stdio.h>      // for printf()

int exec_syscall_17( int *num )
{
        int     retval = -1;
        asm("   movl    $17, %eax       ");
        asm("   movl    %0, %%ebx " : : "m" (num) );
        asm("   int     $0x80           ");
        asm("   movl    %%eax, %0 " : "=m" (retval) );
        return  retval;
}


int main( int argc, char **argv )
{
        int     number = 15;
        printf( "\nDemonstrating system-call 17 ... \n" );
        for (int i = 0; i <= 5; i++)
        {
                printf( "\n#%d:  number=%d ", i, number );
                exec_syscall_17( &number );
        }
        printf( "\n\n" );
}
//-----------------------------------------------------------------
```

APPENDIX THREE

SYSTEM REQUIREMENTS

This appendix will list down the software requirements to run/compile the different components of GridStat such as the LeafQoSBroker, publisher, status router and the subscriber. It also presents the command line usage for the C publisher, subscriber and the (edge) SR and sample steps to setup and carry out a simple experiment.

- Java: C SR depends on the GridStat 3.0 LeafQoSBroker to supply the routing paths. GridStat 3.0 is written in Java and requires jdk1.5+, which can be downloaded from `http://java.sun.com/javase/downloads/index_jdk5.jsp`. For latest jdk can goto `http://java.sun.com/javase/downloads/index.jsp`.

- CORBA: C SR, Publisher and Subscriber has been tested with both MICO[1] 2.3.11(recommended) and Java ORB. MICO can be downloaded from `http://www.mico.org/down.html` while Java ORB is an integral part of jdk. Documentation for MICO installation is present within the MICO tar ball (<Installation Directory>/doc/doc.ps).

- C++ Compiler: C SR, publisher and subscriber were compiled using c++ (GCC) 3.4.6 with the *pthread* and the *rt* library (comes with the default installation of c++). The c++ compiler comes with the default installation of Linux or can be installed as a separate package.

Command Line Parameters/Usage

- **SR**

  EdgeStatusRouter.out [OPTIONS] or StatusRouter.out [OPTIONS]

  Options are:

  -p <Starting Port Number>: Optional. Ports to which publishers/subscribers can connect to

[1] an open source CORBA implementation

are allocated starting at this value. Default value is *10000*.

-n <Name of the Status Router>: Optional. Name of the Status Router as identified by other entities. Default value is *c1.e0*.

-l <Name of the LeafQoSBroker>: Optional. Name of the LeafQoSBroker as registered with the CORBA Naming Service. Default value is *c1*.

<Arguments to initialize the CORBA Naming Service>: Mandatory. These options vary with the CORBA implementation being used.

For MICO it is

-ORBInitRef NameService=corbaloc::<Host Name where the Naming Service is running>:<Naming Service Port Number>/NameService .

Example: *-ORBInitRef NameService=corbaloc::gridstat05.eecs.wsu.edu:9000/NameService*

- **Naming Service: MICO**

  nsd -ORBIIOPAddr <NSD-address>

  <NSD-address> is Base Server Address in format inet:<hostname>:<port>

  Example: *nsd -ORBIIOPAddr inet:gridstat05.eecs.wsu.edu:9000*

- **Naming Service Administration: MICO**

  nsadmin -ORBNamingAddr <NSD-address>

  <NSD-address> is Base Server Address in format inet:<hostname>:<port>

  Example: *nsadmin -ORBNamingAddr inet:gridstat05.eecs.wsu.edu:9000*

- **Publisher**

  Publisher.out [OPTIONS]

  Options are:

  -p <Starting Port Number>: Optional. Port Number through which the events will be published. Default value is *10000*.

-n <Name of the Publisher>: Optional. Name of the Publisher as identified by other entities. Default value is *c1.pub0*.

-e <Name of the Status Router>: Optional. Name of the Edge Status Router to Connect to. Default value is *c1.e0*.

<Arguments to initialize the CORBA Naming Service>: Mandatory. These options vary with the CORBA implementation being used.

For MICO it is

-ORBInitRef NameService=corbaloc::<Host Name where the Naming Service is running>:<Naming Service Port Number>/NameService .

Example: *-ORBInitRef NameService=corbaloc::gridstat05.eecs.wsu.edu:9000/NameService*

- **Subscriber**

  Subscriber.out [OPTIONS]

  Options are:

  -p <Starting Port Number>: Optional. Ports Number on which the Subscriber will receive events. Default Value is *10000*.

  -n <Name of the Subscriber>: Optional. Name of the Subscriber as identified by other entities. Default value is *c1.sub0*.

  -e <Name of the Status Router>: Optional. Name of the Edge Status Router to Connect to. Default value is *c1.e0*.

  <Arguments to initialize the CORBA Naming Service>: Mandatory. These options vary with the CORBA implementation being used.

  For MICO it is

  -ORBInitRef NameService=corbaloc::<Host Name where the Naming Service is running>:<Naming Service Port Number>/NameService .

  Example: *-ORBInitRef NameService=corbaloc::gridstat05.eecs.wsu.edu:9000/NameService*

Steps to Setup a Simple Publisher-SR-Subscriber System:

Assuming that all the components will be running on a single host (gridstat05).

1. Start the CORBA Naming Service: *nsd -ORBIIOPAddr inet:gridstat05:9000*

2. Start LeafQoSBroker: *java -jar leafQoSBrokerSimpleGUI.jar*

3. Start EdgeStatusRouter: *./EdgeStatusRouter.out -p 10000 -n c1.e0 -l c1 -ORBInitRef Name-Service=corbaloc::gridstat05:9000/NameService*

4. Start Publisher: *./Publisher.out -p 11000 -n c1.pub0 -e c1.e0 -ORBInitRef NameService=corbaloc::gridstat05:9000/NameService*

   Connect to the Edge Status Router and publish a variable *var* every 100ms.

5. Start Subscriber: *./Subscriber.out -p 12000 -n c1.sub0 -e c1.e0 -ORBInitRef NameService=corbaloc::gridstat05:9000/NameService*

   Connect to the Edge Status Router and subscribe to the variable *var* every 100ms.

BIBLIOGRAPHY

[1] Netfilter. www.netfilter.org.

[2] Guide to realtime programming. Digital Equipment Corporation, March 1996. DIGITAL UNIX Version 4.0 or higher. Available at http://mia.ece.uic.edu/papers/WWW/books/posix4/TOC.HTM.

[3] Complete Linux loadable kernel modules. pragmatic / THC, version 1.0, March 1999. Available at http://packetstormsecurity.org/docs/hack/LKM_HACKING.html.

[4] Real-time in the Solaris 8 operating environment. Sun Microsystems, October 2000. Available at http://www.opengroup.org/rtforum/oct2000/slides/litchfield.pdf.

[5] IEEE standard communication delivery time performance requirements for electric power substation automation. *IEEE Std 1646-2004*, pages 1–24, 2005.

[6] Roadmap to secure control systems in the energy sector. Energetics Incorporated. Prepared for US Dept. of Energy and US Dept. of Homeland Security, January 2006. Available at http://www.controlsystemsroadmap.net/.

[7] Transmission technology road map. Bonneville Power Administration, September 2006. Available at http://www.bpa.gov/corporate/business/innovation/docs/2006/RM-06_Transmission.pdf.

[8] PJM 2007 strategic report. PJM, April 2007. Available at www2.pjm.com/documents/downloads/strategic-responses/report/20070402-pjm-strategic-report.pdf.

[9] Trustworthy Cyber Infrastructure for the Power grid. TCIP Project, 2008. http://www.iti.uiuc.edu/tcip/index.html.

[10] S. F. Abelsen. Adaptive GridStat information flow mechanisms and management for power grid contingencies. Master's thesis, Washington State University, Pullman. Washington, August 2007. Available at www.gridstat.net/publications/TR-GS-012.pdf.

[11] M. M. Adibi and L. H. Fink. Overcoming restoration challenges associated with major power system disturbances - restoration from cascading failures. *Power and Energy Magazine, IEEE*, 4 Issue: 5:68–77, September 2006. Available at http://ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=35600&arnumber=1687819&count=11&index=6.

[12] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, New York, NY, USA, 1999. ACM. Available at www.cs.cornell.edu/Courses/cs614/2003SP/papers/ASS99.pdf.

[13] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001. Chapter 18.

[14] David E. Bakken, Carl H. Hauser, Harald Gjermundrd, and Anjan Bose. Towards more flexible and robust data delivery for monitoring and control of the electric power grid. Technical Report 009, Washington State University, May 2007. Available at http://www.gridstat.net/publications.php.

[15] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly, first edition, December 2005.

[16] Kenneth P. Birman, Jie Chen, Enneth M. Hopkinson, Robert J. Thomas, James S. Thorp, Rober Van Renesse, and Werner Vogels. Overcoming communication challenges in software for monitoring and controlling power systems. *Proceedings of the IEEE*, 93 Issue: 5:1028–1041, May 2005. Available at `http://ieeexplore.ieee.org/iel5/5/30830/01428015.pdf?arnumber=1428015`.

[17] Steve Brosky and Steve Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. *ipdps*, 00:120a, 2003.

[18] Fengyun Cao and Jaswinder Pal Singh. Efficient event routing in content-based publish-subscribe service networks. *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, 2:929– 940, March 2004. Available at `http://ieeexplore.ieee.org/iel5/9369/29790/01356980.pdf?arnumber=1356980`.

[19] Amit Choudhary. Implementing a system call on Linux 2.6 for i386, October 2006. Available at `http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Implement-Sys-Call-Linux-2.6-i386.pdf`.

[20] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting real-time applications in an integrated services packet network: architecture and mechanism. *SIGCOMM Comput. Commun. Rev.*, 22(4):14–26, 1992.

[21] Justin Clarke and Nitesh Dhanjani. *Network Security Tools*. O'Reilly, April 2005. Section 7.2 : Intercepting System Calls.

[22] Allan B. Cruse. File lesson27.ppt, newcall.c and try17.cpp. Department of Computer Science and Department of Mathematics, University of San Francisco. Available at `http://www.cs.usfca.edu/cruse/cs635f07/`.

[23] Jeffery E. Dagle. Postmortem analysis of power grid blackouts.the role of measurement systems. *Power and Energy Magazine, IEEE*, 4 Issue : 5:30–35, September 2006. Available at http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1687815.

[24] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM. Available at www.cs.toronto.edu/syslab/courses/csc2209/06au/papers/fairq.pdf.

[25] Constantinos Dovrolis, Dimitrios Stiliadis, and Parameswaran Ramanathan. Proportional differentiated services: delay differentiation and packet scheduling. *IEEE/ACM Trans. Netw.*, 10(1):12–26, 2002. Available at http://ieeexplore.ieee.org/iel5/90/21263/00986503.pdf?arnumber=986503.

[26] M. Rio et al. A map of the networking code in Linux kernel 2.4.20. Research and Technological Development for a TransAtlantic Grid, March 2004. Available at www.labunix.uqam.ca/~jpmf/papers/tr-datatag-2004-1.pdf.

[27] Patrick Th. Eugster. Kernel korner: Linux socket filter: sniffing bytes over the network. *Linux J.*, 2001(86):8, 2001. Available at http://www.linuxjournal.com/article/4659.

[28] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30(2):115–126, 2001. Available at www.msrg.utoronto.ca/publications/sigmod01.pdf.

[29] Kjell Harald Gjermundrod. *Flexible qos-managed status dissemination middleware framework for the electric power grid*. PhD thesis, Washington State University, Pullman, WA,

USA, 2006. Available at https://research.wsulibs.wsu.edu:8443/dspace/handle/2376/550.

[30] Carl H. Hauser, David E. Bakken, and Anjan Bose. A failure to communicate: next generation communication requirements, technologies, and architecture for the electric power grid. *Power and Energy Magazine, IEEE*, 3 Issue: 2:47–55, Mar-Apr 2005.

[31] Carl H. Hauser, David E. Bakken, Ioanna Dionysiou, K. Harald Gjermundrd, Venkata Irava, Joel Helkey, and Anjan Bose. Security, trust and QoS in next-generation control and communication for large power systems. *International Journal of Critical Infrastructures*, 4(1/2):3–16, 2008.

[32] Wenbo He and Klara Nahrstedt. Impact of upper layer adaptation on end-to-end delay management in wireless ad hoc networks. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 59–70, Washington, DC, USA, 2006. IEEE Computer Society. Available at http://ieeexplore.ieee.org/iel5/10742/33866/01613323.pdf?tp=&isnumber=&arnumber=1613323.

[33] Joel Helkey. Achieving end-to-end delay bounds in a real-time status dissemination network. Master's thesis, Washington State University, Pullman. Washington, May 2007.

[34] Magen Hughes. Hacking with Linux kernel module (lkm). Available at www.pcmahughes.com/m/hackingwithlkm.ppt.

[35] Ed Hursey. Overview of the real time Linux. Available at http://ece-www.colorado.edu/~ecen5623/ecen/Overview_of_Real_Time_Linux.pdf.

[36] Venkata S. Irava. *Low-Cost Delay-Constrained Multicast Routing Heuristics and their Evaluation*. PhD thesis, Washington State University, Pullman, WA, USA, August 2006. Available

at https://research.wsulibs.wsu.edu:8443/dspace/bitstream/2376/552/1/v_irava_072106.pdf.

[37] Venkata S. Irava and Carl Hauser. Survivable low-cost low-delay multicast trees. *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, 1:6, November-December 2005. Available at http://ieeexplore.ieee.org/iel5/10511/33285/01577363.pdf?tp=&arnumber=1577363&isnumber=33285.

[38] Ariane Keller. Manual: tc packet filtering and netem. ETH Zurich, July 2006. http://tcn.hypert.net/tcmanual.pdf.

[39] Giordana Lisa. QoS in packet-switching networks. Available at http://gd.tuwien.ac.at/.vhost/winpcap.polito.it/netgroup/hp/qos/index.html.

[40] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association. Available at http://www.acmesecurity.org/publicacoes/biblioteca/redes/net-bpf.pdf.

[41] Klara Nahrstedt, Hao hua Chu, and Srinivas Narayan. QoS-aware resource management for distributed multimedia applications. *J. High Speed Netw.*, 7(3-4):229–257, 1998. Available at http://mll.csie.ntu.edu.tw/papers/qos_jhsn1998.pdf.

[42] Arno Puder. MICO: An open source CORBA implementation. *IEEE Softw.*, 21(4):17–19, 2004. Available at http://ieeexplore.ieee.org/iel5/52/29063/01309641.pdf?tp=&isnumber=&arnumber=1309641.

[43] Yongho Seok, Jaewoo Park, and Yanghee Choi. Queue management algorithm for multirate wireless local area networks. *14th IEEE Proceedings on Personal, Indoor and Mobile Radio*

Communications, 2003. PIMRC 2003, 3 Issue : 7-10:2003 – 2008, September 2003. Available at http://ieeexplore.ieee.org/iel5/8905/28147/01259066.pdf?arnumber=1259066.

[44] Sameer Shende. Profiling and tracing in Linux. Department of Computer and Information Science, University of Oregon, Eugene. Available at http://www.cs.uoregon.edu/research/paraducks/papers/linux99.pdf, January 1988.

[45] Tim Szigeti and Christina Hattingh. *End-to-End QoS Network Design: Quality of Service in LANs, WANs, and VPNs*. Cisco Press, 2004.

[46] K. Tomsovic, David E. Bakken, M. Venkatasubramanian, and Anjan Bose. Designing the next generation of real-time control, communication and computations or large power systems. *Proceedings of the IEEE*, 93 Issue: 5:965–979, May 2005. Available at www.gridstat.net/publications/IEEE-Proceedings-Submitted.pdf.

[47] Ahmet Uyar, Shrideep Pallickara, and Geoffrey Fox. Audio video conferencing in distributed brokering systems. Department of Electrical Engineering and Computer Science, Syracuse University. Available at http://grids.ucs.indiana.edu/ptliupages/publications/NB-AudioVideo.pdf.

[48] Jun Wang, Li Xiao, King-Shan Lui, and Klara Nahrstedt. Bandwidth sensitive routing in DiffServ networks with heterogeneous bandwidth requirements. *ICC '03. IEEE International Conference on Communications, 2003*, 1:188–192, May 2003. Available at http://ieeexplore.ieee.org/iel5/8564/27113/01204167.pdf?arnumber=1204167.

[49] Johnathon Weare and Paolo De. Nictolis. Kernel comparison: Linux (2.6.22) versus Windows (Vista), 2008. Available at http://widefox.pbwiki.com/Kernel%20Comparison%20Linux%20vs%20Windows.

[50] Clark Williams. Linux scheduler latency. Red Hat, Inc, March 2002. Available at `http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf`.

[51] Zhaoxia Xie, G. Manimaran, Vijay Vittal, A. G. Phadke, and Virgilio Centeno. An information architecture for future power systems and its reliability analysis. *IEEE Transactions on Power Systems*, 17 Issue: 3:857–863, Aug 2002. Available at `http://ieeexplore.ieee.org/iel5/59/22197/01033736.pdf?arnumber=1033736`.

[52] Yuan Xue, Kai Chen, and Klara Nahrstedt. Achieving proportional delay differentiation in wireless LAN via cross-layer scheduling: Research articles. *Wirel. Commun. Mob. Comput.*, 4(8):849–866, 2004.

[53] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. *SIGCOMM Comput. Commun. Rev.*, 20(4):19–29, 1990.