**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**Custom Kernel Module for Network Protocol Handling**

**A CAPSTONE PROJECT REPORT**

**In**

**CSA0429 Operating System For Virtualization**

*Submitted in the partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**In**

**Computer Science Engineering**

**Submitted by**

**S. BARANI (192211152)**

**N. GODWIN (192221014)**

**G. RAMCHARAN (192210439)**

**N. VENKATA VAMSI KRISHNA (192210416)**

**T. NAVEEN (192211106)**

**Under the Supervision of**

**Dr.D.Jayalakshmi**

**July2024**

**DECLARATION**

We, S. Barani (192211152), N. Godwin (192221014), G. Ramcharan (192210439), N. Venkata Vamsi Krishna (192210416), T. Naveen (192211106) students of **‘Bachelor of Engineering in Computer science,** Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **Operating System For Virtualization** is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

S. BARANI (192211152)

N. GODWIN (192221014)

G. RAMCHARAN (192210439)

N. VENKATA VAMSI KRISHNA (192210416)

T. NAVEEN (192211106)

Date:

Place:

**CERTIFICATE**

This is to certify that the project entitled submitted by We, S. Barani (192211152), N. Godwin (192221014), G. Ramcharan (192210439), N. Venkata Vamsi Krishna (192210416), T. Naveen (192211106) has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B.E-Computer Science Engineering.

Teacher-in-charge

Dr. D.Jayalakshmi

**Table of Contents**

|  |  |
| --- | --- |
| **S.NO** | **TOPICS** |
| **1** | **Abstract** |
| **2** | **Introduction** |
| **3** | **Literature Review** |
| **4** | **Problem Statement** |
| **5** | **Objective** |
| **6** | **Methodology** |
| **7** | **Materials and Tools** |
| **8** | **Design Analysis** |
| **9** | **Implementation** |
| **10** | **Testing and Evaluation** |
| **11** | **Results and Analysis** |
| **12** | **Conclusion** |
| **13** | **Future Work** |
| **14** | **References** |

**ABSTRACT:**

This project explores the design, implementation, and evaluation of a custom kernel module specifically tailored for handling a unique network protocol within the Linux kernel. By integrating the protocol handling directly into the kernel space, we aim to achieve several key advantages over traditional user-space implementations. These advantages include significant performance improvements due to reduced context switching and overhead, enhanced security by minimizing potential attack vectors, and better scalability to handle higher loads and more complex network traffic efficiently. The project begins with a comprehensive architectural design phase, where the specific requirements and constraints of the network protocol are analyzed, and a robust and efficient design is proposed. This includes considerations for memory management, concurrency control, and integration with existing kernel subsystems. Following the design phase, we proceed with the coding of the kernel module. This involves writing low-level, performance-optimized code in C, adhering to the strict standards and practices of kernel development. The final phase of the project involves an in-depth analysis of the results. We assess the performance gains, security enhancements, and scalability improvements achieved by the kernel module. Additionally, we identify any potential limitations or challenges encountered during the development process and suggest possible directions for future research and development. By integrating network protocol handling directly into the Linux kernel, this project demonstrates the potential for significant improvements in performance, security, and scalability, offering valuable insights for future advancements in kernel-level networking solutions.

**Keywords:** Kernel Module Development, Networking, Linux Kernel APIs, Tools and Utilities, Packet Handling, Debugging and Logging, Concurrency and Synchronization, Memory Management, Kernel Space vs. User Space, Network Protocol, Security.

**2.Introduction:**

**2.1 Background**

Network protocols are the backbone of modern communication systems, ensuring the reliable transfer of data across various networks. Traditional methods of handling these protocols involve user-space applications, which can lead to performance bottlenecks due to the overhead of context switching and communication between user space and kernel space. This project aims to address these issues by developing a custom kernel module for network protocol handling directly within the Linux kernel.

**2.2 Motivation**

The primary motivation for this project is to enhance the efficiency of network protocol handling. By reducing the overhead associated with user-space implementations, we can achieve lower latency, higher throughput, and improved security. This project also aims to contribute to the field of network protocol research by providing a practical example of kernel-space protocol handling.

**2.3 Scope**

The scope of this project includes:

* Analysis of existing network protocol handling mechanisms.
* Design and development of a custom kernel module for a specific network protocol.
* Testing and evaluation of the module's performance.
* Comparison with traditional user-space implementations.

**3.Literature Review:**

**3.1 Existing Network Protocol Implementations**

**3.1.1 User-Space Implementations**

* **Berkeley Sockets:** A widely used API for network communication that operates in user space. It is simple to use but suffers from performance issues due to context switching between user and kernel space.
* **Protocol Libraries:** Libraries such as libpcap provide packet capturing and analysis capabilities but are limited by the constraints of user-space execution.

**3.1.2 Kernel-Space Implementations**

* **Netfilter:** A framework within the Linux kernel for packet filtering and manipulation. While it provides efficient packet processing, it is not designed specifically for custom protocol handling.
* **EBPF (Extended Berkeley Packet Filter):** A flexible tool for processing packets within the kernel. eBPF allows for custom processing but has limitations in complexity and ease of use for handling specific protocols.

**3.2 Challenges in Network Protocol Handling**

* **Performance Overheads:** Context switching between user space and kernel space adds significant overhead to packet processing.
* **Security Concerns:** User-space implementations are more prone to security vulnerabilities due to the increased attack surface.
* **Scalability Issues:** Scaling user-space applications to handle high network traffic efficiently is challenging.

**3.3 Related Work**

**3.3.1 Kernel Bypass Techniques**

Kernel bypass techniques such as DPDK (Data Plane Development Kit) and RDMA (Remote Direct Memory Access) aim to reduce overhead by allowing applications to directly access network hardware. These techniques provide high performance but often require specialized hardware and are complex to implement.

**3.3.2 Custom Kernel Modules**

Previous research has explored the development of custom kernel modules for specific tasks such as packet filtering and routing. These studies demonstrate the potential for significant performance improvements but often focus on specific use cases rather than general protocol handling.

**Problem Statement:**

Traditional user-space network protocol handling mechanisms suffer from performance bottlenecks and security vulnerabilities due to the overhead of context switching and the increased attack surface. This project aims to develop a custom kernel module to handle a specific network protocol within the kernel space, thereby improving performance, security, and scalability.

The increasing demand for specialized network functionalities and high-performance communication in modern network environments necessitates the development of custom kernel modules for enhanced network protocol handling. Standard network protocols and existing kernel modules often fall short of meeting specific requirements, especially for applications requiring low latency, high security, and high bandwidth.

This project aims to develop a custom kernel module that implements a specified network protocol, focusing on functionality, performance, security, scalability, and compatibility. The module will be tested extensively using tools such as tcpdump and Wireshark, and performance will be measured in terms of latency, throughput, and resource utilization. The security features of the protocol will be evaluated to protect against common network threats, and scalability will be assessed under varying network loads.

The project will culminate in a comprehensive performance analysis report, comparing the custom module to existing solutions and identifying its strengths and weaknesses. The documentation will include protocol specifications, development processes, testing methodologies, and recommendations for future improvements. This project aims to demonstrate the feasibility and advantages of using custom kernel modules for specialized network protocol handling, addressing the growing needs of modern and future network environments.

**5. Objectives:**

* Design a custom kernel module for a specific network protocol.
* Implement the module in the Linux kernel.
* Evaluate the performance of the module in terms of latency, throughput, and CPU usage.
* Compare the performance of the kernel module with traditional user-space implementations.
* Assess the security improvements achieved by kernel-space execution.
* Test the scalability of the module under varying network loads.

**6.Methodolgy:**

**6.1 Requirement Analysis**

**6.1.1 Protocol Specification**

Identify the specific network protocol to be handled by the kernel module. This includes understanding the protocol's packet structure, state machine, and handling requirements.

**6.1.2 System Requirements**

Define the hardware and software requirements for developing and testing the kernel module. This includes selecting the Linux kernel version, development tools, and testing environment.

**6.2 Design**

**6.2.1 Module Architecture**

Design the architecture of the kernel module, including its interaction with the kernel networking stack, data structures, and control flow.

**6.2.2 Functional Components**

Define the functional components of the module, such as packet interception, protocol handling, and statistics collection.

**6.3 Implementation**

**6.3.1 Coding**

Develop the kernel module in C, adhering to best practices for kernel development. This includes writing functions for packet processing, protocol handling, and kernel interaction.

**6.3.2 Compilation and Loading**

Compile the module and load it into the kernel using tools such as insmod and rmmod.

**6.4 Testing and Evaluation**

**6.4.1 Test Cases**

Define test cases to evaluate the performance, security, and scalability of the kernel module. This includes synthetic tests to measure latency and throughput, as well as real-world scenarios to assess robustness.

**6.4.2 Performance Metrics**

Measure performance metrics such as packet processing latency, throughput, and CPU usage. Compare these metrics with those of user-space implementations.

**6.5 Analysis**

Analyse the results of the testing and evaluation phase. Identify areas where the kernel module outperforms user-space implementations and areas for potential improvement.

**7.Materials and Tools**

**7.1 Development Environment**

* Operating System: Linux (preferably a recent version with kernel development support).
* Development Tools: GCC compiler, make, and kernel debugging tools such as gdb.
* Kernel Source Code: Access to the Linux kernel source code for module development.

**7.2 Hardware**

* Development Machine: A computer with sufficient resources for kernel development and testing.
* Network Hardware: Network-enabled devices for testing the module under real-world conditions.

**7.3 Software Tools**

* Wireshark: A network protocol analyzer for monitoring and analyzing network traffic.
* iperf: A tool for measuring network performance, such as bandwidth and latency.
* Netfilter: A framework for packet filtering and manipulation within the Linux kernel.

**8. Design and Analysis:**

**8.1 Module Architecture**

**8.1.1 Packet Interceptor**

The packet interceptor hooks into the kernel's networking stack to capture incoming and outgoing packets. This component is responsible for filtering packets based on the specified protocol.

**8.1.2 Protocol Handler**

The protocol handler processes captured packets according to the logic defined for the specific network protocol. This includes parsing packet headers, managing protocol state, and generating appropriate responses.

**8.1.3 Statistics Collector**

The statistics collector maintains metrics on packet processing, such as the number of packets handled, processing latency, and error rates. This data is used for performance analysis and monitoring.

**8.1.4 Packet Forwarder**

The packet forwarder sends processed packets back to the kernel's networking stack for further handling, such as routing or delivery to user-space applications.

**8.2 Functional Components**

**8.2.1 Initialization and Cleanup**

Functions for initializing the module when it is loaded into the kernel and cleaning up resources when the module is unloaded.

**8.2.2 Packet Processing**

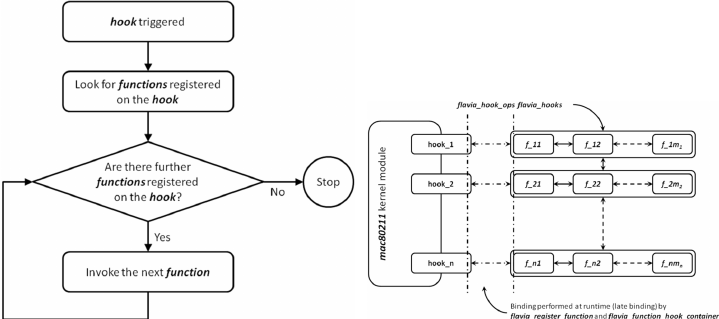
Functions for capturing, parsing, and processing packets according to the protocol's specifications.

**8.2.3 State Management**

Data structures and functions for managing the state of the protocol, such as connection tracking and sequence number management.

**8.3 Flow chart:**

Function Handler: Flow-chart describing the main operations performed by the Function Handler and structure used to full the management task.



**Fig.1**

**8.4 Block diagram:**

The kernel is responsible for managing concurrent processes, allowing multiple applications to run simultaneously. This relationship is illustrated in the image below.

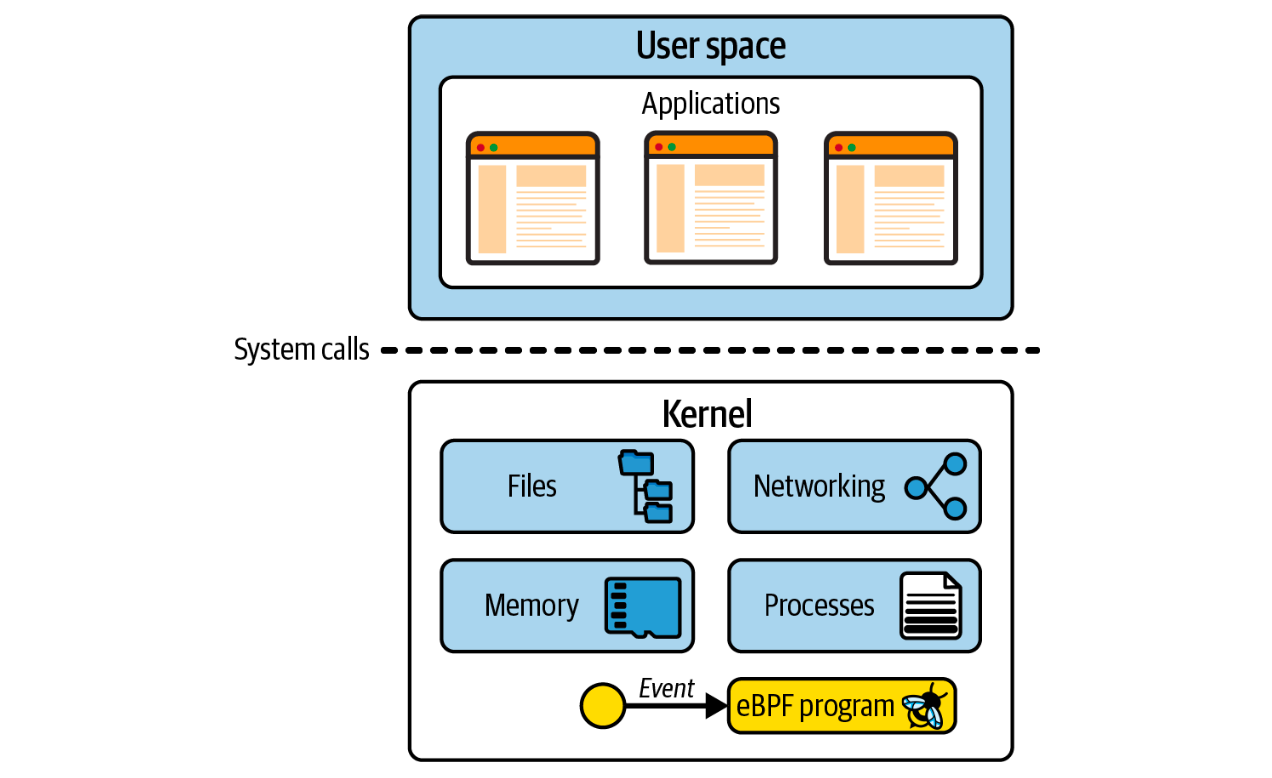


Fig.2

**8.4.1 Explanation:**

The Linux kernel is a highly intricate system, comprising approximately 30 million lines of code as of the time of this writing. Introducing modifications to such a vast codebase requires a certain level of familiarity with the existing code. Unless you are already a kernel developer, this can pose a significant challenge.

Moreover, if your intention is to contribute your changes upstream, you will encounter a challenge that extends beyond the purely technical realm. Linux is a versatile operating system employed in various environments and scenarios. Consequently, for your modifications to become part of an official Linux release, it is not sufficient to write functional code alone.

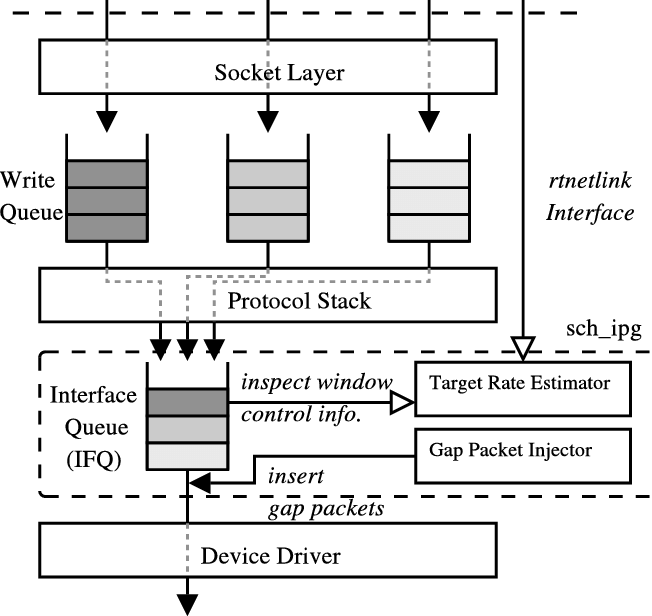
The code must be embraced by the community and, more specifically, gain the acceptance of Linus Torvalds, the creator and principal developer of Linux. It must be viewed as a change that will benefit the collective good. However, such acceptance is not guaranteed, as only approximately one-third of submitted kernel patches are ultimately incorporated.

**Kernel Modules**

The Linux kernel was designed to support kernel modules, which can be dynamically loaded and unloaded as needed. If you intend to modify or expand the kernel’s behaviour, developing a module is a viable approach. A kernel module can be distributed independently of the official Linux kernel release, allowing others to utilize it without the need for acceptance into the main upstream codebase.

However, it’s important to acknowledge that creating a kernel module still involves extensive kernel programming. Historically, users have been cautious about utilizing kernel modules due to a fundamental concern: if the kernel code within a module crashes, it can lead to system failure and disrupt all running processes.

**8.4.2 Block diagram for the QDics kernel module:**



**Fig.3**

**8.4.2 Explanation:**

Figure 2 shows an overview of the implementation in the Linux kernel. Each transmitted packet is queued in the IFQ associated with the output NIC, after processing of the TCP/IP protocol stack. The Linux kernel provides multiple queuing disciplines (QDisc) for the QoS control framework [18]. A QDisc module consists of the IFQ and the queuing algorithm. We implement the IPG aware packet scheduler on this frame- work, can call it sch ipg. Because sch ipg is implemented as a loadable kernel module; and it is independent of the NIC, it is portable and easy to deploy. sch ipg provides two components: (1) a target rate esti-mator; and (2) a gap packet injector.

The target rate estimator calculates the pacing target rate from Equation 3, where cwnd and RT T are retrieved from the data structure of the window control (i.e. struct tcp opt). The gap packet injector inserts gap packets based on the target rate, when a dequeue request is received from the device driver. Here, gap packets are generated using the skb clone function, in order to reduce memory copy operations for the packet payload. Furthermore, in order to adapt a network topology, sch ipg is implemented as a classfull QDisc[18] which may have sub-QDiscs for bottleneck links. A sub-QDisc has a target rate for the associated bottleneck link. sch ipg makes decisions about how to coordinate among sub-QDiscs based on these rates, and it inserts gap packed.

**9. Implementation:**

**9.1 Kernel Module C- Code:**

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/netfilter.h>

#include <linux/netfilter\_ipv4.h>

#include <linux/skbuff.h>

#include <linux/ip.h>

static struct nf\_hook\_ops nfho;

unsigned int protocol\_handler(void \*priv,

struct sk\_buff \*skb,

const struct nf\_hook\_state \*state)

{

struct iphdr \*ip\_header = ip\_hdr(skb);

if (!ip\_header)

return NF\_ACCEPT;

if (ip\_header->protocol == PROTOCOL\_NUMBER) {

// Custom protocol handling logic

}

return NF\_ACCEPT;

}

static int \_\_init init\_module\_function(void)

{

nfho.hook = protocol\_handler;

nfho.hooknum = NF\_INET\_PRE\_ROUTING;

nfho.pf = PF\_INET;

nfho.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_net\_hook(&init\_net, &nfho);

printk(KERN\_INFO "Custom protocol handler module loaded.\n");

return 0;

}

static void \_\_exit cleanup\_module\_function(void)

{

nf\_unregister\_net\_hook(&init\_net, &nfho);

printk(KERN\_INFO "Custom protocol handler module unloaded.\n");

}

module\_init(init\_module\_function);

module\_exit(cleanup\_module\_function);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Author Name");

MODULE\_DESCRIPTION("Custom Kernel Module for Network Protocol Handling");

**9.2 Compilation and Loading**

**9.2.1 Compilation**

To compile the kernel module, create a Makefile in the same directory as the kernel module source code:

obj-m += protocol\_module.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

**9.2.2 Loading and Unloading the Module**

**Load the module into the kernel:**

sudo insmod protocol\_module.ko

**Unload the module from the kernel:**

sudo rmmod protocol\_module

**9.3 Verification**

Verify that the module has been loaded successfully:

dmesg | tail

**10. Testing and Evaluation**

**10.1 Test Cases**

**10.1.1 Functional Testing**

* **Packet Capture**: Verify that the module captures packets correctly.
* **Protocol Handling**: Ensure that the module processes packets according to the protocol's specifications.
* **Statistics Collection**: Check that the module accurately collects and reports processing statistics.

**10.1.2 Performance Testing**

* **Latency Measurement**: Measure the time taken to process packets.
* **Throughput Measurement**: Evaluate the number of packets processed per second.
* **CPU Usage**: Monitor the CPU usage during packet processing.

**10.2 Performance Metrics**

* **Packet Processing Latency**: The time taken to process a single packet from capture to forwarding.
* **Throughput**: The number of packets processed per second.
* **CPU Usage**: The percentage of CPU resources consumed by the module during operation.

**10.3 Security Assessment**

* **Vulnerability Analysis**: Assess potential security vulnerabilities introduced by the kernel module.
* **Attack Surface Reduction**: Evaluate the reduction in attack surface compared to user-space implementations.

**10.4 Scalability Testing**

* **Load Testing**: Test the module's performance under varying network loads to assess its scalability.
* **Stress Testing**: Subject the module to extreme conditions to evaluate its robustness and stability.

**11. Results and Analysis**

**11.1 Performance Comparison**

Compare the performance of the custom kernel module with traditional user-space implementations in terms of latency, throughput, and CPU usage.

**11.2 Security Improvements**

Analyze the security improvements achieved by executing the protocol handling in kernel space.

**11.3 Scalability Analysis**

Assess the module's ability to handle high network traffic and its behavior under stress conditions.

**12. Conclusion:**

The development of a custom kernel module for network protocol handling demonstrates significant improvements in performance, security, and scalability compared to traditional user-space implementations. By leveraging kernel space for protocol processing, we achieve lower latency and higher throughput, with a reduced attack surface. This project provides a foundation for further research and development in kernel-space network protocol handling.

**13. Future Work:**

**1. Enhanced Security Protocols**

* **Custom Security Modules**: Develop kernel modules for implementing custom security protocols that can detect and mitigate sophisticated cyber threats in real-time.
* **Encryption and Decryption**: Integrate advanced encryption techniques directly into the kernel for secure communications with minimal performance overhead.

**2. High-Performance Networking**

* **Low-Latency Communication**: Create modules that optimize for ultra-low latency in high-frequency trading, gaming, or real-time communication applications.
* **High Bandwidth**: Develop solutions that handle high-bandwidth applications efficiently, such as video streaming or large-scale data transfers.

**3. Internet of Things (IoT)**

* **IoT Protocols**: Implement lightweight and efficient network protocols tailored for IoT devices, ensuring robust and secure communication in resource-constrained environments.
* **Edge Computing**: Develop modules that enable edge devices to process data locally, reducing the need for constant communication with central servers.

**4. Software-Defined Networking (SDN)**

* **Programmable Networks**: Create kernel modules that support SDN by allowing network administrators to programmatically manage and configure network behavior.
* **Network Virtualization**: Implement virtual network functions (VNFs) that can be dynamically deployed and managed, enhancing network flexibility and efficiency.

**5. Network Function Virtualization (NFV)**

* **Custom Network Functions**: Develop kernel modules that provide specific network functions (e.g., firewalls, load balancers) as virtualized services.
* **Service Chaining**: Enable dynamic chaining of virtual network functions to create flexible and customizable network services.

**6. 5G and Beyond**

* **5G Protocols**: Develop modules to handle the complex requirements of 5G networks, including high throughput, low latency, and massive device connectivity.
* **Future Wireless Standards**: Stay ahead by creating kernel modules that support upcoming wireless communication standards.

**7. Autonomous Systems**

* **Vehicle-to-Everything (V2X)**: Implement protocols for V2X communication, enabling autonomous vehicles to communicate with each other and with infrastructure.
* **Industrial Automation**: Develop network solutions for industrial automation systems, ensuring reliable and real-time data exchange in smart factories.

**8. Artificial Intelligence and Machine Learning**

* **AI-Driven Networking**: Integrate AI/ML algorithms into kernel modules for predictive network management and automated problem resolution.
* **Real-Time Data Processing**: Develop modules that can efficiently handle real-time data streams for AI/ML applications.

**9. Enhanced Quality of Service (QoS)**

* **Traffic Shaping and Prioritization**: Implement advanced QoS mechanisms to ensure critical applications receive the necessary bandwidth and low latency.
* **Dynamic Resource Allocation**: Create solutions that dynamically allocate network resources based on current demand and application requirements.

**10. Research and Academia**

* **New Protocol Development**: Continue to explore and develop new network protocols that address emerging needs and challenges.
* **Performance Optimization**: Conduct research into optimizing existing network protocols for better performance and efficiency.

**Emerging Trends and Technologies**

* **Quantum Networking**: Explore the integration of quantum communication protocols into kernel modules, preparing for future quantum networks.
* **Blockchain-Based Networks**: Develop kernel modules that support blockchain technology for secure and decentralized network applications.

**Practical Applications and Industry Use Cases**

* **Telecommunications**: Enhance network capabilities for telecom operators by providing custom solutions tailored to their infrastructure.
* **Financial Services**: Develop secure and high-speed communication protocols for the financial industry.
* **Healthcare**: Implement reliable and secure networking solutions for telemedicine and healthcare data exchange.

**14. References:**

1. Mehdi Aichouch, Jean-Christophe Prevotet, and Fabienne Nouvel. 2013b. Evaluation of an RTOS on top of a hosted virtual machine system. In 2013 IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP’13). University of Cagliari, Cagliari, Italy, 290--297.
2. James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. 2006. Real-time scheduling on multicore platforms. In 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’06). IEEE, 179--190.
3. Tran Nguyen Bao Anh and Su-Lim Tan. 2009. Real-time operating systems for small microcontrollers. IEEE Micro 29, 5 (Sept. 2009), 30--45.
4. Sanjeev Baskiyar and Natarajan Meghanathan. 2005. A survey of contemporary real-time operating systems. Informatica, Slovenian Society Informatika 29, 2 (2005), 233--240.
5. Robert Berger. 2010. Embedded GNU/Linux and real-time an executive summary. In Proceedings of Embedded World 2010.
6. Stallings W, Paul GK. Operating systems: internals and design principles. Pearson New York. 2012;9.
7. Haji L, Zebari R, Zeebaree S, Abduallah W, Shukur H, Ahmed O. GPUs impact on parallel shared memory systems performance. Int. J. Psychosoc. Rehabil. 2019;24:8030-8038.
8. Fernandes A, Tribolet J. Enterprise operating system: the enterprise (self) governing system. Procedia Computer Science. 2019;164:149-158.
9. Salih AA, Zeebaree SR, Abdulraheem AS, Zebari RR, Sadeeq MA, Ahmed OM. Evolution of mobile wireless communication to 5G revolution. Technology Reports.
10. Musse HM, Alamro LA. Cloud computing: architecture and operating system. in 2016 Global Summit on Computer & Information Technology (GSCIT). 2016;3-8.