MobiVPN: Towards a Reliable and Efficient Mobile VPN

by

Abdullah O. Alshalan

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved August 2017 by the
Graduate Supervisory Committee:

Dijiang Huang, Chair
Gail-Joon Ahn
Adam Doupé
Yanchao Zhang

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

A Virtual Private Network (VPN) is the traditional approach for an end-to-end secure connection between two endpoints. Most existing VPN solutions are intended for wired networks with reliable connections. In a mobile environment, network connections are less reliable and devices experience intermittent network disconnections due to either switching from one network to another or experiencing a gap in coverage during roaming. These disruptive events affects traditional VPN performance, resulting in possible termination of applications, data loss, and reduced productivity. Mobile VPNs bridge the gap between what users and applications expect from a wired network and the realities of mobile computing.

In this dissertation, MobiVPN, which was built by modifying the widely-used OpenVPN so that the requirements of a mobile VPN were met, was designed and developed. The aim in MobiVPN was for it to be a reliable and efficient VPN for mobile environments. In order to achieve these objectives, MobiVPN introduces the following features: 1) Fast and lightweight VPN session resumption, where MobiVPN is able decrease the time it takes to resume a VPN tunnel after a mobility event by an average of 97.19% compared to that of OpenVPN. 2) Persistence of TCP sessions of the tunneled applications allowing them to survive VPN tunnel disruptions due to a gap in network coverage no matter how long the coverage gap is. MobiVPN also has mechanisms to suspend and resume TCP flows during and after a network disconnection with a packet buffering option to maintain the TCP sending rate. MobiVPN was able to provide fast resumption of TCP flows after reconnection with improved TCP performance when multiple disconnections occur with an average of 30.08% increase in throughput in the experiments where buffering was used, and an average of 20.93% of increased throughput for flows that were not buffered. 3) A fine-grained, flow-based adaptive compression which allows MobiVPN to treat each

tunneled flow independently so that compression can be turned on for compressible flows, and turned off for incompressible ones. The experiments showed that the flow-based adaptive compression outperformed OpenVPN's compression options in terms of effective throughput, data reduction, and lesser compression operations.

To my mother (Sarah)„,

To my father (Othman)„,

To my wife (Rayya)„,

To my daughter (Maysan)„,

To my sisters (Huda, Amal, Taghreed, Eman)„,

To my brothers (Mohammed, Abdulrahman, Badr, Ayman, Hossam)„,

To my nephews and nieces„,

To the soul of my uncle (Abdulrahman Alfouzan)„,

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Dr. Dijiang Huang. His guidance, discussions, immense knowledge, patience and encouragement have tremendously allowed me to grow as a research scientist. Without his support and mentorship, this dissertation would have not been possible.

I would also like to express my thanks to my committee members: Prof. Gail-Joon Ahn, Dr. Adam Doupé and Dr. Yanchao Zhang. Their discussions and feedback have been extremely helpful.

My gratitude is extended to King Saud University for providing me with a scholarship to pursue my doctoral degree. I also want to thank the Saudi Arabian Cultural Mission in the U.S. for facilitating my PhD scholarship.

I want to thank my parents (Othman and Sarah), my brothers (Mohammed, Abdulrahman, Badr, Ayman, Hossam), my sisters (Huda, Amal, Taghreed, Eman), my nephews and nieces for their support and patience throughout the years. Words cannot express how thankful and grateful I am to them. I could have not done this without them.

I would like to thank my lab-mate and colleague Dr. Sandeep Pisharody for his contribution to my published research work and for the helpful discussions we had. I would also like to thank my colleagues and lab-mates for the stimulating discussions and weekly siminars. They are: Dr. Chun-Jen Chung, Dr. Bing Li, Yuli Deng, Adel Alshamrani, Oussama Mjihil, Dr. Huijun Wu, Dr. Zhijie Wang, Dr. Tianyi Xing, Dr. Zhibin Zhou and Ankur Chowdhary.

Special thanks to my friends Dr. Mohammed Alhussein and Dr. Ziming Zhao for their support and helpful discussions, and to my father-in-law Dr. Sulaiman Aljarallah for his support and encouragement.

I also want to extend my thanks to all of my friends here in Arizona who made it feel like home.

Finally and most importantly, I would like to deeply thank my wife, Rayya Aljarallah, for her unconditional love and support. She has been a great source of encouragement and comfort for the past three years of my life.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

The global computing industry is quickly evolving toward having powerful cloud computing resources aimed at providing services over the Internet, with mobile devices behaving as the user interface into this cloud. In such an environment, having a way to securely connect these mobile terminals to a cloud computing resource like MobiCloud, introduced by (Huang *et al.*, 2010), is of great importance, not just for information assurance and protection of intellectual property, but often for regulatory and compliance reasons.

In addition, according to (StatCounter, 2016), in October, 2016, worldwide mobile and tablet Internet usage exceeded desktop usage for the first time. Mobile users may want to establish a secure connection with home networks when they are using public networks so that their traffic is always encrypted while visiting an untrusted network.

Classical Virtual Private Network (VPN) establishes secure connections between a remote user and a protected network by encrypting and tunneling packets sent through the Internet rather than building a true private network (Heyman, 2007). These VPN connections however, are best suited for stationary devices which, unlike mobile devices, tend to have a stable network connection (Tzvetkov, 2010). Most mobile devices are susceptible to intermittent network disconnections while switching from one network to another or experience a gap in coverage and could remain disconnected for any lengths of time (Goff *et al.*, 2000). For example, a mobile device might switch between WiFi and cellular, or between one WiFi and another. Such connection losses or connection changes can cause the VPN connection to break, causing the tunneled mobile applications to experience some undesirable side effects such as pos-

sible termination of their flows, packet loss, or degraded performance. This produces an inconvenient user experience due to the user possibly having to redo incomplete jobs.

Given the ever increasing popularity of remote workers and Bring-Your-Own-Devices (BYOD) in work places along with the ubiquitous presence of wireless networks that these devices have access to, it is prudent to have a mobile VPN solution that can provide a VPN experience that does not require the user to reset and reconfigure the VPN session upon switching between networks, and to ensure that connection-oriented flows can be resumed once connectivity has been restored. According to a survey done by (Data, 2014), 79% of over 1600 surveyed IT and security professionals ranked mobility as a top priority. In the same survey, 71% of the respondents expressed that data security is the major concern about mobility.

OpenVPN is a widely used open source VPN. The goal in this dissertation is to design and develop MobiVPN which address the limitations of OpenVPN, and make it suitable for mobile environments. The aim of MobiVPN is to satisfy the mobile VPN requirements which are set forth in the following section.

1.1 MobiVPN Requirements

A mobile VPN is required to maintain the VPN session between a VPN client and a VPN server despite interruption of network connectivity, or when the mobile device moves between networks and possibly obtains new IP addresses. Network disruptions and network changes due to mobility should not affect an application sessions. Figure 1.1 shows how a mobile device connected to the cloud can travel between networks, get new network information, and still appear to maintain the same session from an application perspective.

The mobile VPN was formally defined to have the following requirements:

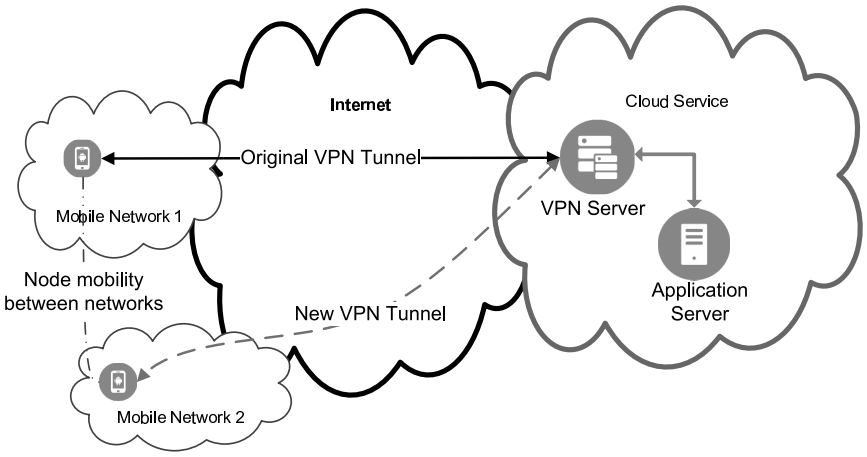

Figure 1.1: Overview of the Network Infrastructure.

1. *Network Roaming*: The VPN session remains alive during roaming and the virtual connection remains connected when the device switches to a different network, hence getting a new IP address.

2. *Applications Persistence*: Open application connections remain active when the network connection changes or is interrupted even for long periods.

3. *Security*: a mobile VPN should enforce a mechanism for authenticating the user and providing encryption of the data traffic along with integrity assurance. Eavesdroppers should not be able to infer what the data in the encrypted tunnel is.

4. *Performance*: a mobile VPN should take into consideration the constraints on the mobile resources as opposed to desktops. For example, compression and encryption can be done adaptively in which unnecessary compression or encryption can be avoided.

MobiVPN inherits the security requirements from OpenVPN. We aimed for MobiVPN to satisfy the network roaming requirement where the VPN session is resumed in a fast and light-weight manner once the VPN client switches networks. In addi-

tion, MobiVPN provides persistence to applications that utilize TCP as a transport protocol by hiding the network layer disruptions from the application layer in order to maintain independence of the end-to-end application sessions from issues caused by mobility. Finally, the performance of MobiVPN was improved by supporting flow-based adaptive compression, which outperformed OpenVPN's adaptive compression.

1.2 Contributions

The contributions made in this dissertation are:

- The mobile VPN technologies were surveyed and a state-of-the-art literature review is provided in Chapter 2.

- Discussed in Chapter 3, is the design and development of a fast and light-weight VPN session resumption. This allows MobiVPN to detect network changes and resume the VPN session in a fraction of the time when compared to OpenVPN. Not only does the mobile VPN client benefit from our protocol, but our design is also included in the VPN server so that it can resume the VPN session of connected clients when the VPN server's IP address changes, as can happen in Moving-Target-Defense (MTD) systems that employ for instance, IP hopping.

- In Chapter 4, a model, design, and implementation of TCP-based applications persistence is provided. In this work, TCP flows can survive intermittent network disconnections, resume data transmission as soon as the VPN tunnel is resumed, and maintains or recovers the TCP sending rate when buffering is enabled in MobiVPN.

- Finally, provided in Chapter 5 is the design and implementation of a flow-based adaptive compression mechanism in which the decision to whether or not

compress packets is done on a per flow basis. Compression is then enabled for compressible flows and turned off for incompressible ones. It is shown that the flow-based adaptive compression can reduce traffic size the most with the least number of compression operations, as it was able to avoid unnecessary and unfeasible compression operations.

Some of the content of Chapters 2 and 4 have been published in the following publications, respectively:

- Alshalan, A., Pisharody, S., & Huang, D. (2016). A Survey of Mobile VPN Technologies. IEEE Communications Surveys & Tutorials, 18(2), 1177-1196.
- Alshalan, A., Pisharody, S., & Huang, D. (2016). MobiVPN: A mobile VPN providing persistency to applications. In Computing, Networking and Communications (ICNC), 2016 International Conference on. IEEE.

Chapter 2

LITERATURE REVIEW OF MOBILE VPN TECHNOLOGIES

Mobile VPN is a broad class of protocols that seek to deliver secure IP mobility (Tzvetkov, 2010). An ideal protocol would satisfy the requirements that were set forth in Section 1.1. Of the several products and protocols that seek to adapt VPNs for mobility, a few different approaches were studied. They are discussed in the remainder of this chapter. Figure 2.1 shows a taxonomy of the mobile VPN technologies discussed in this section.

2.1 Mobile VPN Through Network Mobility

In this section, several mobile VPN technologies that support mobility at the network layer are discussed.

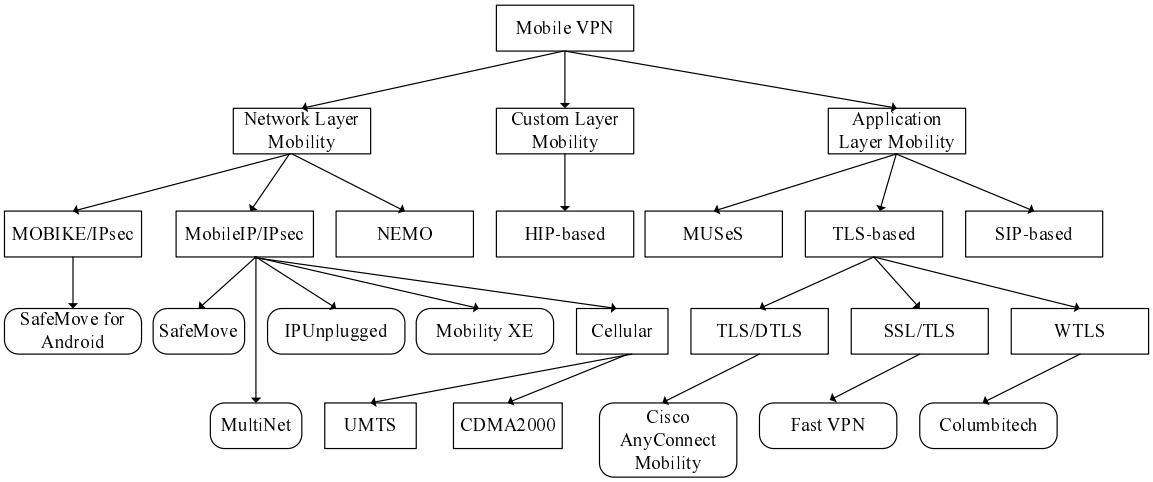

Figure 2.1: Mobile VPN Technologies and Solutions Taxonomy.

2.1.1 Mobile IPv4 Based VPNs

This type of mobile VPN relies on two protocols, IPsec and MIPv4. A mobile node (MN) first obtains an IP address for its Home Network and registers it with a home agent (HA). When the MN roams and connects to a foreign network, it obtains a new IP address and registers with a foreign agent (FA). As shown in Figure 2.2, the FA establishes an IPsec tunnel between itself and the HA and informs the HA that FA's IP address is the new Care-of-Address (CoA) of the MN1. All packets sent from a correspondent node (CN) to the MN1 go at first to the HA, which then sends them to the FA though the IPsec tunnel. The FA has the capability to realize to which MN these packets are destined to, and, therefore, it will forward them to the MN1. This is the compulsory approach of this mobile VPN. A voluntary approach is achieved by having the MN acting as its own FA as the case for MN2 in Figure 2.2.

The IPsec tunnel is established between the MN2 and the HA. The MN2 will register its newly obtained IP address with the HA. Just like the compulsory approach, packets destined to MN2 have to be routed to the HA first, which causes the triangular routing anomaly.

Authors in (Uskov, 2012) presented a benchmark for the performance of authentication and encryption algorithms used in the IPsec-based mobile VPNs.

2.1.2 Mobile IPv4 with Two HA Based VPNs

Incorporating MIP into IPsec based VPN gives rise to several technical issues. When an MN moves away from its home network, it must establish an IPsec tunnel with the VPN gateway using the CoA it received after moving. Since all packets including MIP messages are encrypted by IPsec, the FA cannot decrypt them, thereby rendering it unable to relay the MIP messages (Adrangi and Levkowetz, 2005a). This

problem can be avoided by having a mechanism with two HAs, one for internal and one for external networks (Vaarala and Klovning, 2008a). The MN would use the internal HA (i-HA) if it is in the home network and an external HA (x-HA) when it moves out of its home network. This device adds another layer of MIP, which is underneath IPsec, as shown in Figure 2.3. Upon receiving a new CoA, the IPsec tunnel will not have to be broken, and the FA would be able to decrypt the messages as well. When the MN ventures out to visit a network, it would follow a registration process, as shown in Figure 2.4.

This solution, proposed in IETF RFC 5265, has several merits. First, there is no modification required to the MIP and IPsec standards. Modifications to the MN are slight (Huang *et al.*, 2005). The solution, however, leads to problems determining:a) where the x-HA should be placed; b) the trustworthiness of the x-HA; c) how to protect traffic going to the x-HA; and d) the performance impact of having three extra headers to the payload (Huang *et al.*, 2005).

(Benenati *et al.*, 2002) built on the work of (Feder *et al.*, 2003) and used a variant of this IETF solution, along with multiple tunneling protocol standards, to offer a transport layer solution across 3G and WLAN. The proposed solution provides a solution for mobility between interconnected WLAN and 3G networks. The authors

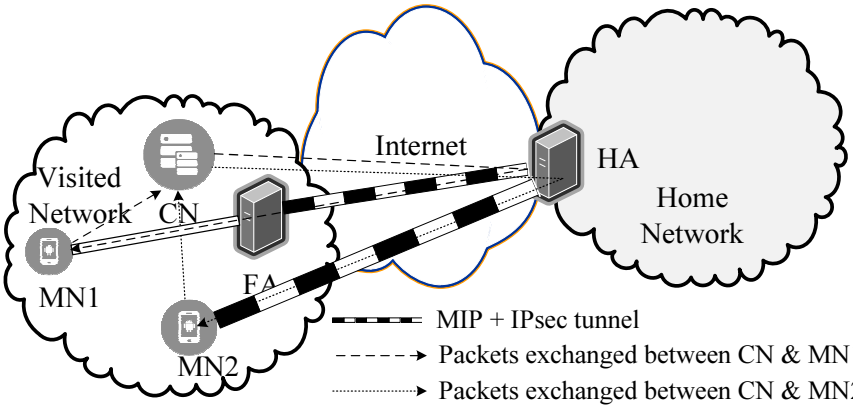

Figure 2.2: MIPv4 Based VPN: MN1 Utilizes a FA, MN2 Acts as Its Own FA.

considered integration of a WLAN system with an existing 3G network either as a wireless Ethernet extension (Tight internetworking) or as complementary to the 3G network (Loose internetworking), with the essential difference being the amount of shared infrastructure between the 3G network and the wireless providers. At a minimum, the Authentication, Authorization, and Accounting (AAA) server is shared between the two technologies. Further, in their solution, (Benenati *et al.*, 2002) assume that the MN is intelligent enough to engage the proper protocols while using a minimal set of credentials for authentication, which are inherently different for various 3G and WLAN technologies. The specifications in IETF RFC 5265 can be adapted for VPN protocols other than mobile IPsec, provided the MN has IPv4 connectivity with an address suitable for registration. Instead of an IPsec gateway, if an TLS gateway or SSH node was used, it could adapt into mobile TLS or mobile SSH VPN connection, as per (Rosado, 2013).

Figure 2.3: Mobile IPsec Packet Format.

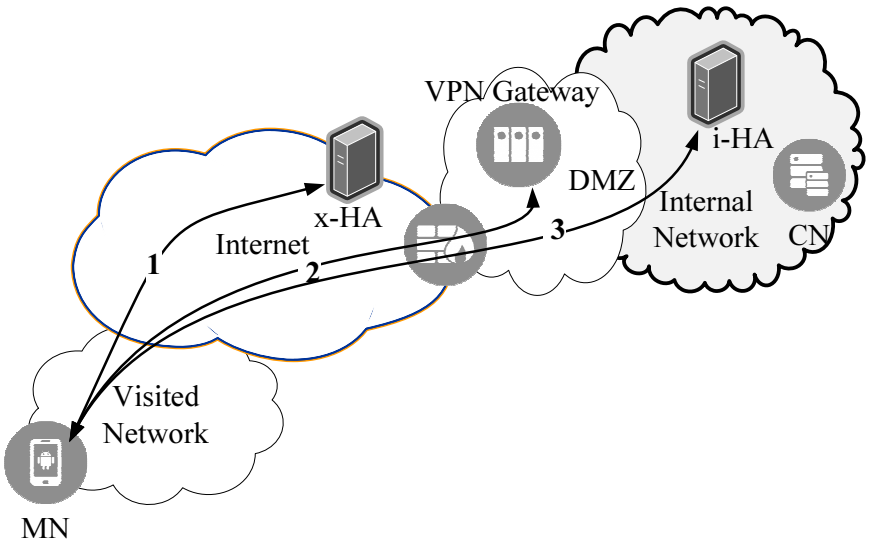

Figure 2.4: Mobile IPsec Registration.

(Dutta *et al.*, 2005) presented a framework named Secure Universal Mobility (SUM) that utilizes the dual HA concept. Their framework suggests a make-before-break approach to reduce the delay incurred while reconstructing the two MIP tunnels and the IPsec tunnel. Based on signal strength, a MN can initialize the handover process before it actually moves from one network to another. This includes activating the target interface and obtaining an IP from the target network. This approach only works if the MN is in the range of both the current network and the future network.

2.1.3 Mobile IPv6 Based VPNs

MIPv6 represents a logical combination of IPv6 and MIP, with knowledge gained from the development of MIP (or, specifically, MIPv4). MIPv6 shares a lot in common with MIP, but, naturally, offers many improvements over MIP. In its native state, IPv6 has features that support mobility, such as the ability of an MN to use its CoA as the source address, along with carrying a home address in the IPv6 header. According to (Braun and Danzeisen, 2001), since every node in an IPv6 network has the ability to interpret this information, there is no longer any need to deploy FA, as used in an MIP deployments . The functions satisfied by an FA in a MIP network, such as discovery and address configuration in foreign networks are not necessary since MNs can operate in any location with no special support required from its local router.

Figure 2.5 shows a sample header structure in an MIPv6 when two MNs need to communicate with one another while in visited networks. Note the capability provided in an IPv6 header to incorporate Extension Headers (EH) that can add multiple IP addresses for mobile situations.

2.1.4 BGP/MPLS Based Mobile VPN

In a BGP/MPLS based mobile VPN, the MN is registered and authenticated using a Diameter server. The MN generates a MIP registration request when it moves into a visited network, as per (Byun and Lee, 2008). In order to have the registration request to be delivered to the Provider Network server (PNS) in the home network, the address of the VPN server replaces the address of the HA in the HA field of the MIP registration request message. (Byun and Lee, 2008) assumed this address to be pre-configured in the MN. A new field named Foreign Customer Equipment (FCE) address is added to specify the address of the MN's gateway in the visited network so that the PNS can determine the gateway serving the MN. Additionally, in the extension field of the MIP registration request message, the address of the visited network AAA is specified instead of the home network.

When the FA receives the MIP registration request message, it generates a message to the AAA in the visited network for authentication. Upon successful authentication and authorization, the AAA sends a message to the PNS to obtain the address of the HA for the MN. When the PNS receives this message, it prepares an IPsec VPN

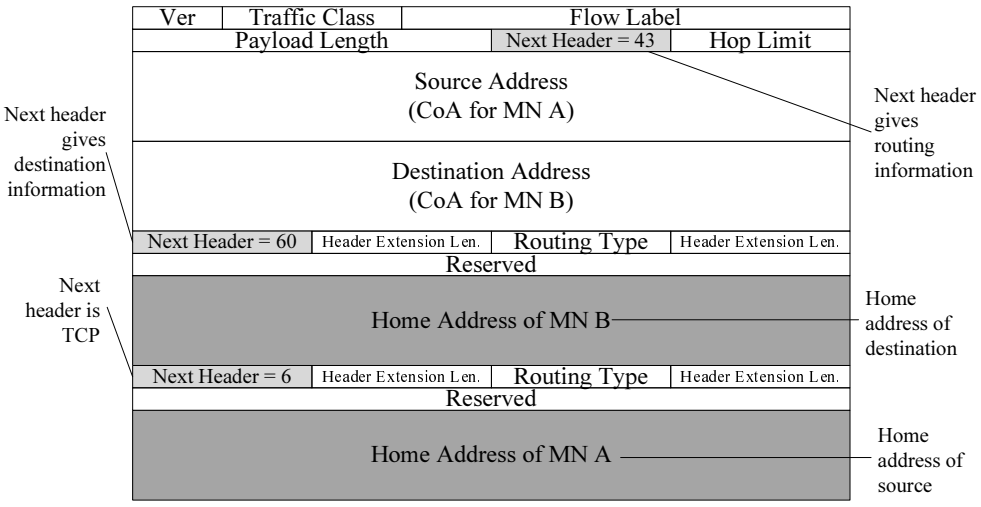

Figure 2.5: MIPv6 Header.

between the visited network and the provider. After establishing an IPsec tunnel between the PE and the visited network CE, the PE inserts the mapping between the address of the MN and the IPsec tunnel into the Virtual Routing and Forwarding (VRF) table of the corresponding MPLS VPN. The other PEs update their VRF table with the updated routing information, and forward the information as determined by the BGP/MPLS protocol. Figure 2.6 illustrates how a mobile VPN user obtains access to a VPN from a visited network.

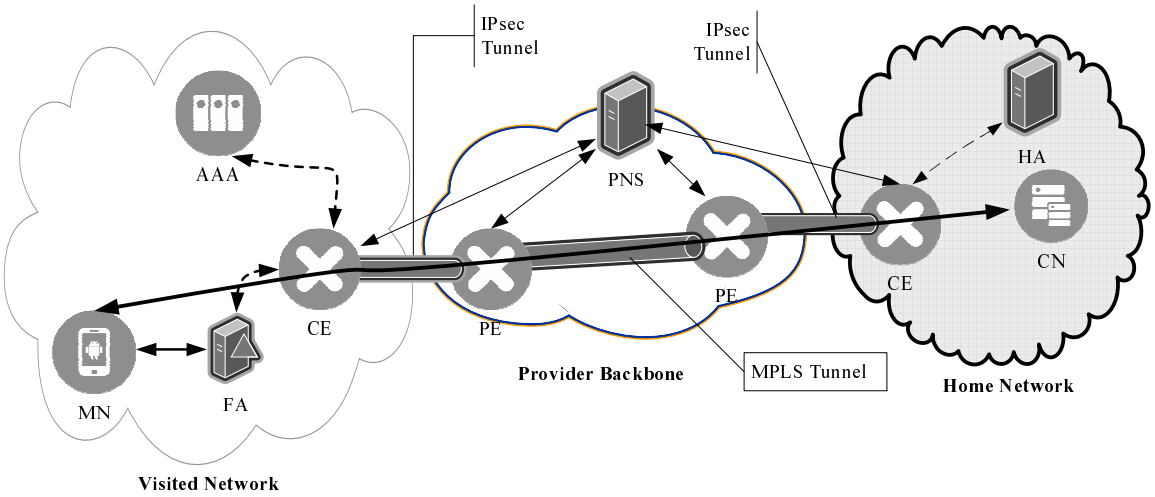

Figure 2.6: MPLS Based Mobile VPN.

2.1.5 MOBIKE-based VPNs

According to (Eronen, 2006), the IKEv2 Mobility and Multihoming Protocol (MOBIKE) solves an inherent problem with IKEv2 and IPsec when the IP address of a MN changes. MOBIKE provides mechanisms to enable MNs with VPN connectivity using an IPsec tunnel mode to preserve the Security Associations (SA) during a Layer 3 handoff (Dutta and Schulzrinne, 2014).

With IKEv1 and IKEv2, the IPsec SAs are created implicitly with the initial IP address of the MN. If the IP address changes, the IPsec tunnel will be torn down and a new SA has to be fully reestablished. MOBIKE enhances this by providing

the ability to create SAs (IKE SA and IPsec SA) that are associated with multiple IP addresses. It also provides the ability to update such addresses without having to reestablish the SAs. Such features are very suitable for MNs with multiple network interfaces like cellular and WiFi. The initiator of the connection (usually the MN) and the responder (VPN Gateway) may include one or more `ADDITIONAL_IP4_ADDRESS` and/or `ADDITIONAL_IP6_ADDRESS` notification messages in the `IKE_AUTH` exchange. During vertical handover, the MN simply notifies the server to use another IP address already agreed upon through the `ADDITIONAL_*_ADDRESS` notification. For horizontal handover, the MN will simply send an `UPDATE_SA_ADDRESSES` notification to update the IP address. The server would then perform a "return routability" check before accepting the new address (Eronen, 2006).

MOBIKE helps in giving the application sessions persistence only if a handover happens fast enough before the application session or the underlying transport layer session times out. Therefore, applications may not survive a long coverage gap where both cellular and WiFi are not available.

2.1.6 Network Mobility (NEMO)

(Devarapalli *et al.*, 2005) proposed a network mobility (NEMO) protocol that treats entire networks, rather than hosts, as mobile. A real-world scenario would be a corporate bus. It is conceivable that every person on the bus would want to VPN into the corporate network. Instead of having several individual VPNs, it would make practical sense to have the network on the corporate bus be an extension of the corporate intranet. The hosts in the bus are static with respect to each other, as the network on the bus moves through different access networks. The protocol is essentially an extension of MIPv6 and is illustrated in Figure 2.7.

A new network device, called a Mobile Router (MR) is introduced in NEMO. The MR registers at the HA as an MN does in a MIPv6 network. But, instead of registering one IP, the MR registers one or many subnets. Packets with destination to the network(s) behind the MR are intercepted by the HA forwarded through a tunnel to the network behind the MR.

While NEMO makes minimal extensions to MIPv6, it has the HA as a single point of failure. However, it reduces overhead and improves performance for a few niche applications.

2.1.7 Cellular networks - CDMA2000 Mobile VPN

CDMA2000 is a 3G technology for cellular systems. It is widely deployed in the Americas and in some regions in Asia and East Europe (Shneyderman and Casati, 2003). The main components in CDMA2000 as shown in Figure 2.8 are:

- CDMA2000 Radio Access Network (RAN). An MN connects to RAN through radio access.
- Packet Control Function (PCF). RAN and PCF communicate through a Radio-Packet (R-P) interface.

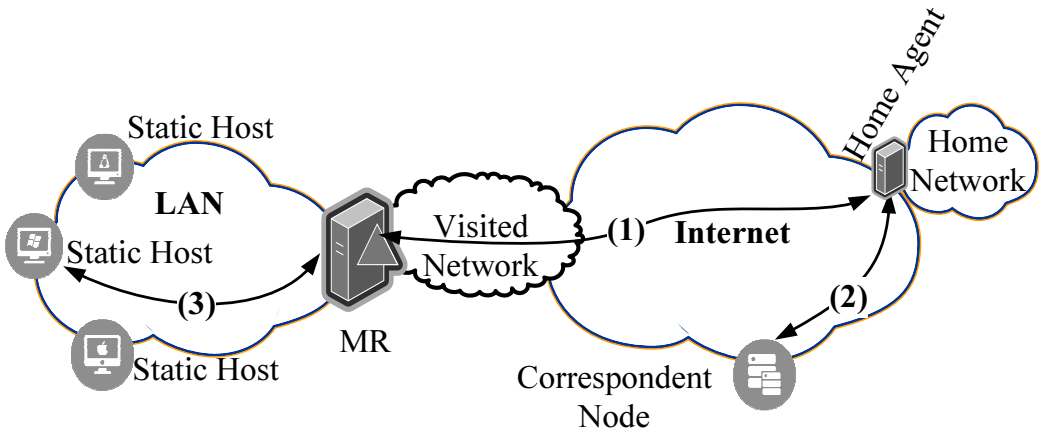

Figure 2.7: NEMO Network Setup.

- Home and foreign AAA servers.
- Packet Data Serving Node (PDSN) acting as a Foreign Agent (FA). PDSN and PCF communicate through a GRE tunnel.
- Home Agent (HA) which communicates with the FA through a MIP/IPsec tunnel.

When an MN visits a CDMA2000 network, it establishes a PPP session with the PDSN (FA). The PPP traffic is actually encapsulated inside R-P traffic. When it reaches the PCF, it decapsulates the R-P traffic to obtain the PPP frames and further encapsulates them inside a GRE packet that gets transferred to the PSDN. The PPP session is terminated at the PSDN. The payload of the PPP frames can then be transfered from the PSDN to the HA via an MIP/IPsec tunnel.

When a MN register with a PDSN, the PDSN delegates the IP assignment to the HA. The HA assigns a dynamic or static IP to that MN. When a MN roams, there are three different levels of mobility:

- The MN leaves the range of one RAN to another. Here a physical layer soft hand-off occurs transparent to the above layers.
- The MN may move far enough to join a range of a new PCF. Here, link layer mobility takes place transparent from layer 3.
- The MN roams to another network. At this point, IP mobility takes place. The MN will register with a new PDSN and the HA will update the mobility binding table resulting in all subsequent traffic being routed via the new PDSN.

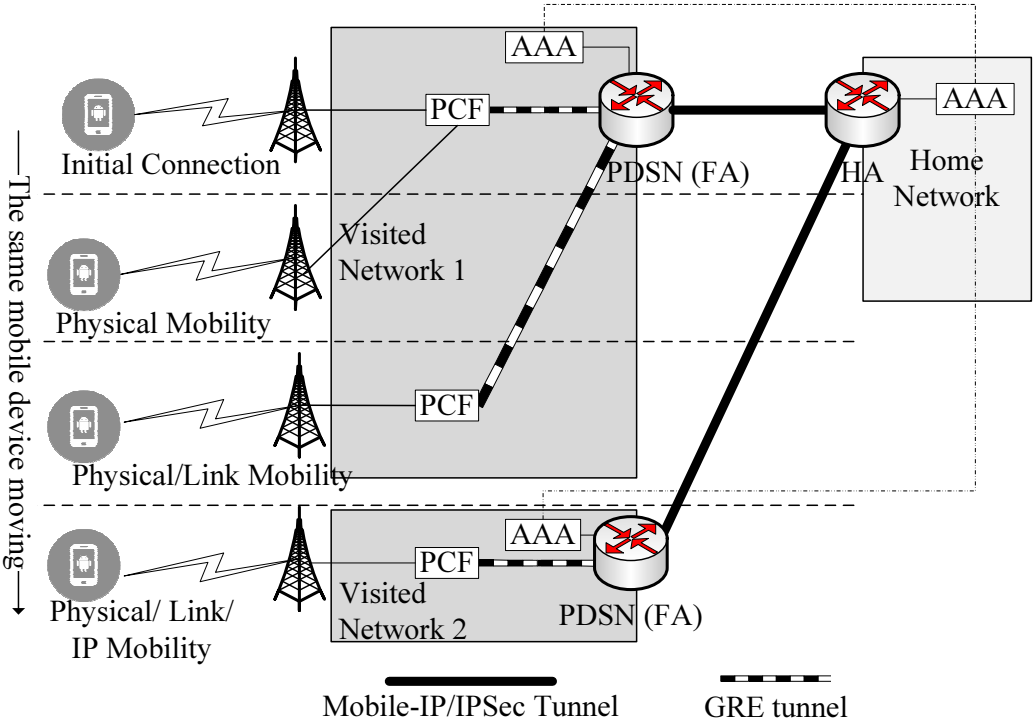

Figure 2.8: Mobile VPN in CDMA2000.

2.1.8 Cellular networks - UMTS Mobile VPN

In cellular networks, VPN mobility is provided through the cellular access network consisting of towers and base stations, and mobile VPNs in cellular networks use a combination of GPRS tunneling protocols (GTP) and IPsec (Shneyderman *et al.*, 2000) as shown in Figure 2.9. GTP encapsulates packets over IP/UDP transport paths and provides control messages to setup and modify tunnels.

An MN in such a setup obtains dynamically allocated IPs and are authenticated by the cellular network providers by the Gateway GPRS support node (GGSN) (Shneyderman *et al.*, 2000). In non-GPRS networks, a node with a different name, but similar functionalities would replace the GGSN. IPsec tunnels are setup between the GGSN and ISPs to transmit traffic to the final destination.

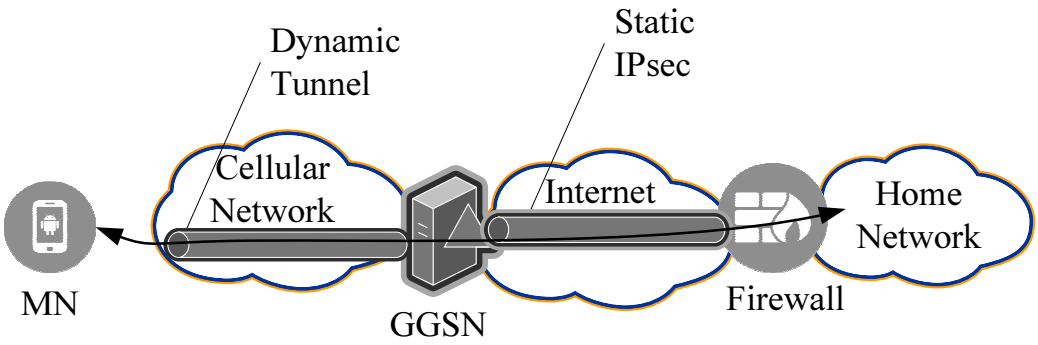

Figure 2.9: Mobile VPN in UMTS Cellular Networks.

2.2 Mobile VPN Through Application Mobility

In this section, mobile VPN solutions that support mobility at the application layer of the TCP/IP protocol stack are discussed.

2.2.1 SIP-Based Mobile VPN

(Huang *et al.*, 2005) propose a SIP-based mobile VPN solution for real time applications, tailored to delivering security and mobility to real-time applications. Figure 2.10 illustrates the proposed SIP-based mobile VPN architecture.

When an MN roams from a home network, an SIP proxy server located within the VPN gateway authenticates the incoming SIP messages, and routes the messages through to another SIP proxy server, which is designated as the SIP registrar. An Application Layer Gateway (ALG) interacts solely with an SIP Proxy server, and oversees all the traffic. When the ALG receives an incoming RTP stream from the home network to a host in the Internet, it replaces the IP/UDP/RTP headers with a SRTP header, and deliveries the stream to the destination. Communication in the reverse direction is handled by verifying the validity of the SRTP packet, and by replacing the SRTP headers with a new RTP header. The payload remains unchanged in both directions. Every such bi-directional communication is represented as a session in the ALG.

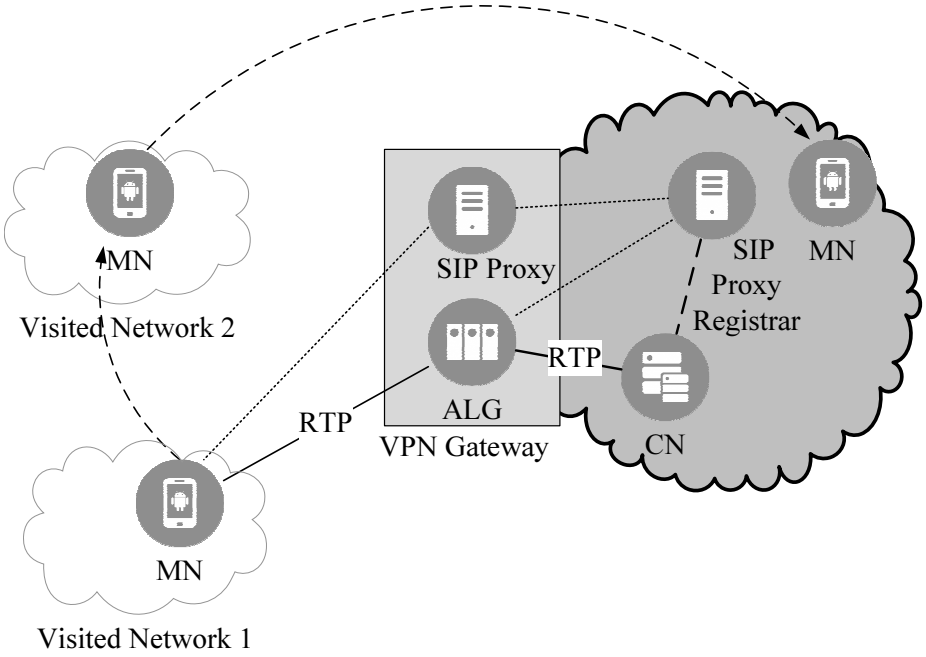

Figure 2.10: SIP-Based Mobile VPN.

As and when an MN enters and leaves its home network, it registers its new location with the SIP registrar during initial session setup. Huang *et al.* used a Diameter service for the registration process. After the MN registers with the SIP registrar, it checks whether there are active sessions in the ALG (Liu *et al.*, 2009). If an active session is found, the MN needs to `RE-INVITE` the CN, where a `RE-INVITE` is essentially an `INVITE` message with the same call-ID as the initial `INVITE` message, with the new contact address of the MN. The `RE-INVITE` is sent to the SIP Proxy in the VPN gateway, which in turn routes the message to the SIP Registrar. If authentication is needed, then the SIP registrar leverages the Diameter server. If the MN is allowed access to the home network, the SIP Registrar uses the ALG to allocate enough resources to guarantee session protection. At this point, the `RE-INVITE` message is routed to the CN (Liu *et al.*, 2009).

When an MN returns back to its home network, the messages do not need to go through the SIP proxy in the VPN gateway. Therefore, upon registering its new

address with the SIP Registrar and sending the `RE-INVITE` message, the SIP Registrar will free all the resources previously allocated. The MN can then communicate directly with the CN without going through the ALG (Liu *et al.*, 2009).

Since the proposed architecture is based on SIP, there is no need to tunnel a packet three times, as is required in the IETF mobile VPN (Section 2.1.2), thereby significantly reducing overhead. Additionally, the proposed architecture is particularly useful for real-time application as are most SIP-based applications are (Liu *et al.*, 2009). Performance of the SIP-based mobile VPN seems to indicate that it is especially suitable for real-time applications, given the small payload in real-time applications (Liu *et al.*, 2009).

The SUM framework we discussed in Section 2.1.2 also utilizes SIP along with MOBIKE as an alternative approach in their mobile VPN framework (Dutta *et al.*, 2005). The main objective is to achieve a dynamic VPN tunnel establishment in order to use a secure VPN tunnel on demand. For example, a secure tunnel is not needed when the mobile client is inside the internal home network or when it is roaming externally, but is not sending sensitive data.

2.2.2 WTLS-based Mobile VPN

One of the most popular commercial mobile VPN products is (Col, 2007), which uses the idea of an application layer solution to add mobility to VPN. By addressing mobility concerns at the application layer, the product liberates the network and transport level connections from having to address mobility, and have those layers working as they were originally designed. The solution relies on recovery mechanisms at the transport layer.

(Col, 2007) splits the client-server connection into three connections as shown in Figure 2.11. The first connection is a TCP/UDP connection inside the MN between

the application client and the mobile VPN client. The VPN client then establishes a session with the VPN server using reliable UDP. Similar to the VPN client, the VPN server establishes a TCP/UDP connection with the application server. This split is used to fool the applications in the MN into believing they are connecting directly to the application server, when, in reality, the application client connection ends at the VPN client.

When the VPN client receives an application request to connect to an application server, mobile VPN will intercept that request and ask the VPN server to connect to the application server. After learning that the VPN server has completed setup with the application server, the mobile VPN client will inform the application that the end-to-end connection to the server is completed successfully. The mobile VPN server and the client setup the VPN session using WTLS. In addition, the system supports multiple VPN servers with a multiplexer that can distribute the load to the VPN servers. When a VPN server experiences failure, all connected clients will lose their sessions and will have to initiate a new connection with a different VPN server since the system does not provide a transparent way to hand-over the sessions of a failed VPN server to an active one.

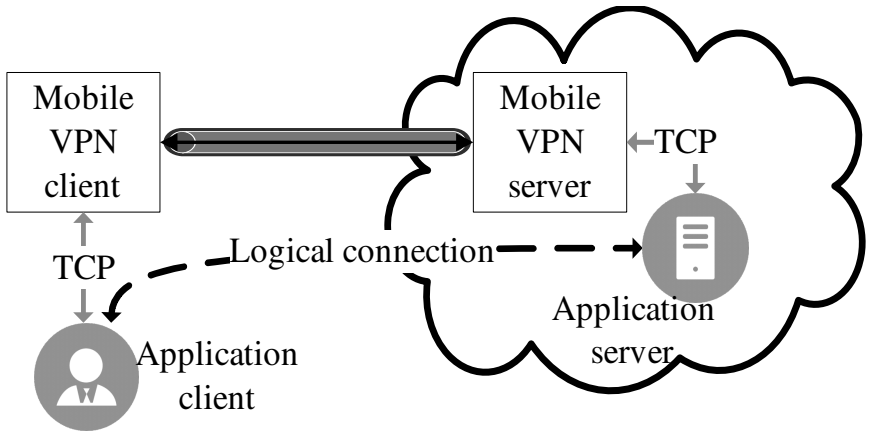

Figure 2.11: Columbitech Mobile VPN Setup.

2.2.3 MUSeS

(Ahmat and Magoni, 2012) suggested a similar application control mobile VPN solution. Their solution, called MUSeS supports both the mobility and traffic security. MUSeS allows user connections to survive disruptions caused by mobility. Similar to Columbitech (Col, 2007), MUSeS hides the network disruptions due to mobility from the user by creating a secure session using an application layer abstraction. MUSeS uses a peer-to-peer overlay network called CLOAK (Tiendrebeogo *et al.*, 2011) above any IP network. Instead of using IPsec or TLS for VPN, MUSeS relies on device identifiers provided and managed by CLOAK to provide encryption and authentication.

When a MUSeS node generates a packet to send to a remote MUSeS node, the packet makes its way through the underlying CLOAK node via a loop back TCP connection. The underlying CLOAK node routes the packet to the destination through the P2P overlay network. The CLOAK node associated with the destination MUSeS node intercepts the packet and locally forwards it to B. The P2P overlay network ensures, therefore, the proper routing of MUSeS secured packets over the network. The authors did not, however, provide explicit details of the security assurances of this mechanism. Instead, they stated that MUSeS protects user communications from common traffic attacks because it uses standard cryptographic algorithms. Since the communication between the MUSeS middleware and the local applications on a machine are not secured, the security of this system appears suspect compared to more traditional VPN solutions. Figure 2.12 shows how a packet is forwarded between the source and the destination.

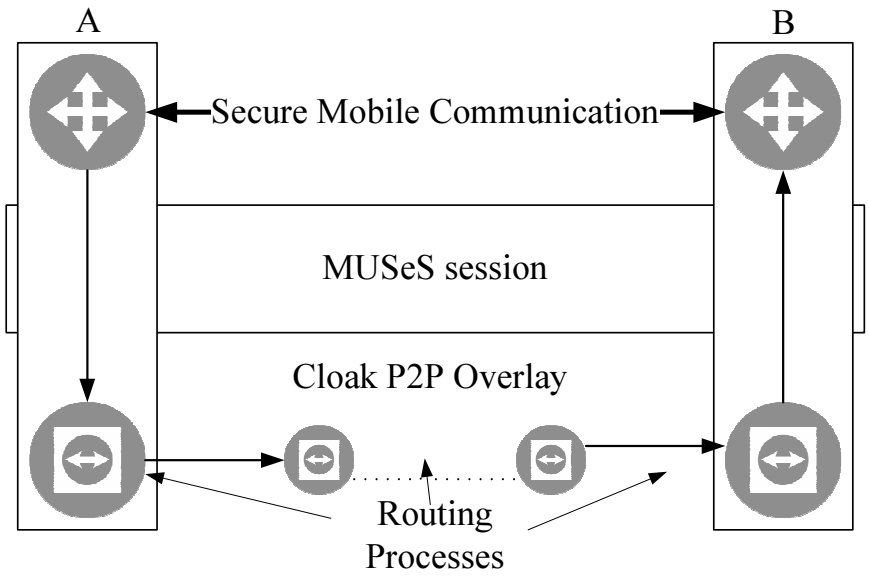

Figure 2.12: MUSeS Setup.

2.2.4 Zuquete and Frade's VPN

(Zúquete and Frade, 2010) suggested a solution for fast VPN mobility of OpenVPN clients across WiFi hotspots. The goal of their solution is to reconfigure an OpenVPN tunnel after a VPN client gets a new IP address post handover to a new network without having to terminate and reestablish the OpenVPN tunnel. This is achieved by updating the VPN tunnel context at the VPN server once the client receives a new IP address. Normally, an OpenVPN server looks up a tunnel context by the VPN client's physical IP address and UDP port. When the client obtains a new physical address due to joining a new network, the OpenVPN server will not be able to associate this client with its original tunnel context. This leads to two major side effects: 1) the client will have to reestablish a new tunnel, causing unnecessary overhead stemming

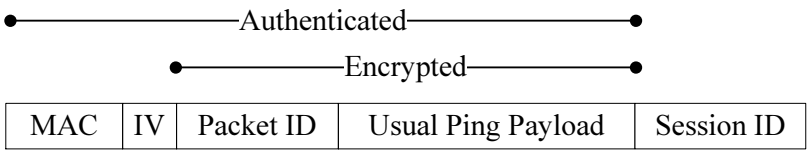

Figure 2.13: Reconfiguration Message in Zuquete and Frade's VPN.

from tunnel setup and new TLS handshake; and 2) the private IP address obtained by the VPN client in the previous session will be less likely be maintained as it will not be released until the previous tunnel context is eliminated by OpenVPN's garbage collection, which only occurs after a certain period of inactivity. Reusing the original tunnel context allows for maintaining the same private IP address, and allows for faster tunnel resumption by avoiding the reestablishment of the tunnel from scratch.

This solution reconfigures the original tunnel context by having the client send the original session ID to the VPN server whenever it obtains a new physical IP. Sending the session ID is done in two ways: a lazy approach and an aggressive approach. In the lazy approach, the session ID (64 bits) is sent in the Initialization Vector (IV) field in all data messages all the time. This works well for CBC cipher-mode as the randomness of the IV does not improve CBC security (Zúquete and Frade, 2010). For the Cipher Feedback mode (CFB) and the Output Feedback mode (OBF), 128-bit IV has to be used since randomness of the IV is a requirement. With 128-bit IV, only the first 64 bits will be constant (occupied by the session ID), while the other 64 bits are random.

In the aggressive approach, the client sends a keep-alive message to the server padded with a clear-text session ID at the end of the message payload. When the VPN server receives such a message, it will not be able to find an entry for the client with the new IP address in the tunnel context table. Thus, it checks the size of this keep-alive message and if it is longer that what it normally is, it detects that this is a reconfiguration message that contains a session ID. The session ID is then used to look up the tunnel context, and if found, the physical IP address associated with this context is updated with the new IP address. Figure 2.13 shows the format of the reconfiguration ping message. This approach is considered aggressive because the

client will keep sending the reconfiguration ping message until a confirmation from the OpenVPN server is received.

this solutions minimizes the packet loss but does not avoid it. In addition, there is no mechanism to maintain the application sessions while the MN is experiencing a gap in WiFi coverage. Allowing the VPN client to maintain the same private IP address is quite helpful, but such a solution would work only if the client was to move from one WiFi network to another immediately, without experiencing a long gap in coverage that could trigger TCP sessions to timeout.

2.3 Host Identity Protocol (HIP) based mobile VPNs

HIP seeks to change the TCP/IP protocol stack to enhance security, mobility and multi-homing capabilities of today's network. A new layer is introduced between Layer 3 and Layer 4 of the protocol stack that contains cryptographic host identifiers as shown in Figure 2.14. HIP provides IPsec encryption and enables authentication to a visiting network and to an intranet firewall.

The use of HIP enables Single Sign-on (SSO) functionality in a visited network, where the operator only has to obtain a list of hosts authorized to use the network. During the HIP handshake, the visited network can verify the identity of the MN (Gurtov, 2008). As long as the MN can authenticate with a network that has a HIP enabled access point, a VPN can continue to operate seamlessly (except for delay caused by the HIP handshake). Similar to the solutions using the IETF RFC 5265, TLS and IPsec VPN solutions can be configured to run on top of an HIP stack, thereby ensuring VPN functionality (Pulkkis *et al.*, 2010). Figure 2.15 shows a sample HIP based mobile VPN tunnel with the minimum required components.

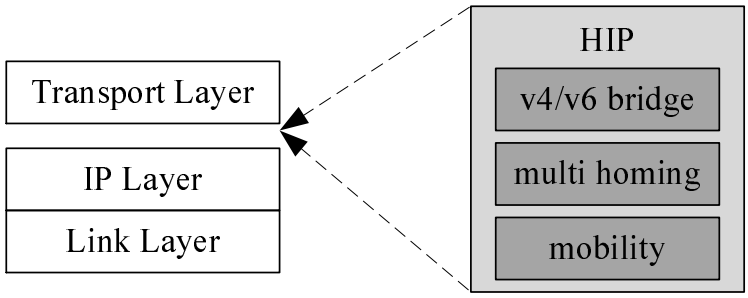

Figure 2.14: HIP Protocol.

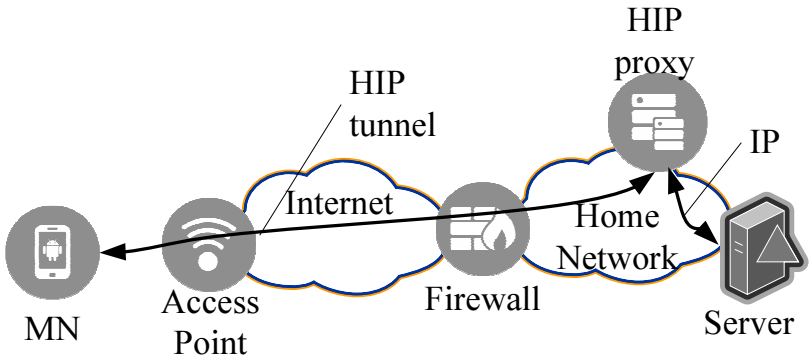

Figure 2.15: HIP Mobile VPN.

2.4 Comparative Analysis

The mobile VPN, based on MIPv4 and IPsec as proposed by IETF, meets the main criteria associated with a mobile VPN: it can handle mobility, and is proven to keep data confidential and authenticate the identity of the systems participating in the VPN. However, it adds a lot of protocol overhead. This could potentially result in throughput degradation and adds to configuration complexity. Throughput degradation is especially critical in low-speed wireless networks. An additional concern, depending on the application in question, is that this type of mobile VPN does not offer application persistence through network connection drops. Application persistence is only guaranteed if the underlying transport protocol, like TCP, remains idle (Comer and Stevens, 2003). It also suffers from the problem of triangle routing or the two-crossing problem in which traffic sent to the MN has to always go to the home agent first, even if the MN and the corresponding node are in the same net-

work (Comer and Stevens, 2003). Finally, this type of mobile VPN suffers from a performance problem which stems from having to reestablish the security association of IPsec. This problem is addressed in a similar mobile VPN that uses two HA. The IPsec tunnel between the external HA and the FA (can be the MN itself) is persistent since the external CoA does not change during mobility. This method however increase the tunneling overhead by adding an extra MIP layer.

MOBIKE-based VPNs offer native support for multiple network interfaces where switching from one interface to another causes no delays if both interfaces are active. If the interface switched to was not active, the delay incurred is only the delay required to obtain a layer 3 IP address. It also supports updating IP addresses during horizontal handover without tearing down IKE and IPsec SAs. Application persistence is guaranteed when there is at least one network available. However, there are no guarantees that applications will survive long coverage gaps.

NEMO is an excellent mobile VPN solution for a niche application. It does not meet many of the requirements of a true mobile VPN solution, but it can be used in association with another mobile VPN solution to reduce overhead and enhance efficiency.

While the BGP/MPLS based mobile VPN technically makes provisions for mobility, and has obvious VPN capabilities, it falls short of the other solutions, since it requires specialized equipment and configuration on part of the ISP. More than a mobile VPN, it should be considered a stationary VPN for nodes with limited mobility. Whenever a node moves from one location to another, the VPN drops, and with it the application sessions. After arriving at a new location, the VPN needs to be setup once again, thereby causing service interruption.

The encrypted radio communication between the user and the cellular access points in mobile VPN configurations in cellular networks are based on encryption

between the user and the mobile network provider (Shneyderman *et al.*, 2000). Additionally, there is a need for specialized network devices like the GGSN, which are owned by entities other than the one the MN is seeking a tunnel to. The setup of multiple tunnels adds overhead, which could impact performance in low bandwidth networks.

the SIP-based mobile VPN (Liu *et al.*, 2009) (Huang *et al.*, 2005) has a centralized client/server architecture owing to the nature of the SIP protocol. This inherently brings with it scalability issues. In addition, the solutions are adapted for real-time applications, and may not suitably convert over for other applications. Moreover, it suffers from the security vulnerabilities of the SIP, which have been widely studied (Geneiatakis *et al.*, 2006).

TLS-based mobile VPNs and its variants (WTLS, DTLS) are more mobility-friendly than the Mobile IP based solutions. This stems from the fact that TLS is an application protocol and, therefore, a TLS session is independent to any changes to the network layer (i.e. IP changes). In order to support application persistence, TLS-based mobile VPNs rely on establishing a virtual interface that remains active even during network disruption. The virtual interface maintains a fixed virtual IP (FVIP) which an application in the MN can use as a source address. True application persistence (TAP) is not natively supported by TLS-based mobile VPNs. If the MN experiences a long coverage gap, the underlying transport protocol may time out.

Mobile VPNs based on HIP have the potential to evolve into a universal mobile VPN solution, since HIP supports mobility in its native form. However, it requires the use of HIP enabled devices in all visited networks, which may not always be feasible, especially in legacy systems. However, HIP VPN solutions appear to lack maturity of other solutions discussed in this paper.

MIPv6 or other MIP type approaches which keeps the VPN tunnels active while an MN is visiting other networks, only partially solves the issues at hand. Depending on the application, communication disruptions while an MN switches networks might crash the application. For this reason, application session persistence during network disruptions is very important, and several of the more accepted mobile VPN solutions like (Col, 2007) (Ahmat and Magoni, 2012) offer the capability to mask network disruptions from the application.

Mobile VPNs that have provisions for application session persistence seem to be the most promising of all mobile VPN options, and appear to be well established in the market. But how these solutions will adapt to IPv6 remains to be seen.

Chapter 3

FAST AND LIGHTWEIGHT VPN SESSION RESUMPTION

3.1 Introduction

In today's computing world, a secure remote connection between two endpoints over a public network remains an essential need. Virtual Private Networks (VPN) still, to this day plays a major role in the computing industry to satisfy this need. Remote users can use a VPN client to create an encrypted tunnel with a VPN server over an insecure public network, which allows them to access resources protected behind the VPN server.

Remote workers rely extensively on VPN technology in order to do their job remotely accessing private company resources securely. However, traditional VPNs were designed for stationary devices. Both the VPN client and server were expected to maintain the same IP address during the entire VPN session. This assumption no longer holds. Nowadays, the use of mobile devices such as smart phones and tablets, is very prevalent. A VPN client's IP address will most likely change after each vertical handover. These devices experience continual IP address change due to roaming from one cellular network to another, switching either from a cellular network to a Wifi network or from a Wifi network to another Wifi network. A VPN server's IP address can also change. This could happen as a result of employing a moving target defense (MTD) technique on the VPN server. An MTD framework can migrate a VPN server from one virtual machine to another with a different IP address, or periodically changes the IP address of the VPN server.

In both cases, the change of the IP address must be handled gracefully maintaining the original VPN session. This is an essential objective since the IP address changes quite frequently in these two cases. In TLS-based VPNs, such as OpenVPN, the security association, unlike IPsec, is not tied to the IP address of the client or the server. Therefore, in theory an established TLS session between two endpoints will not suffer from a change of the IP address, and it can be used after an IP address change. However, an OpenVPN server maintains a list of VPN sessions (instances), one per client, and identifies them by the UDP address of the clients i.e. $< IPaddress, portno. >$. Hence, when the client's IP address changes, any packets the clients send over the VPN tunnel will be ignored by the VPN server since it will not be able to locate the correct VPN session using the new IP address. As a result, the client will have to wait until it eventually times out, and reconnect with the VPN server using a whole new VPN handshake that includes a full TLS handshake.

When the IP address of the VPN server changes, the VPN server performs nothing to preserve or resurrect the VPN sessions of the connected clients as it is designed to be passive in the relation between the VPN client and server. Therefore, all VPN clients will have to time out first, before they can attempt to reconnect with the VPN server. The attempts to reconnect will again require a full handshake and may even fail if the the VPN client is configured to connect to the VPN server's old IP address. In such case a manual user intervention from the client will be needed to update the configuration file with the new IP address, and the whole OpenVPN process has to be restarted. This, indeed, will not be a pleasant experience for the connecting users.

In our work, we developed a light-weight VPN session resumption protocol as part of the MobiVPN project, presented in (Alshalan *et al.*, 2016a), which is built on top of OpenVPN to make it a mobility-friendly VPN. Our design allows both the VPN client and server to be active participants, allowing them to initiate a signaling

protocol which informs the other party of the IP address change. We utilized the TLS session ID to assist in locating the VPN sessions. In this chapter, how we developed our protocol by modifying OpenVPN 2.2.2 is described, and our work was evaluated compared to the same original OpenVPN version.

The remainder of this chapter discusses the related work in Section 3.2. It presents a background about OpenVPN in Section 3.3. Sections 3.4, 3.5 and 3.6 discuss, respectively, the model, design and implementation of our protocol. The evaluation of our protocol is presented in Section 3.7, followed by the conclusion in Section 3.8.

3.2 Related Work

The problem of VPN client mobility has ignited several research studies about how to solve this problem. In IPsec-based VPNs, a MobileIP (MIP) protocol has been utilized to address the mobility problem, in which each mobile node has a fixed home agent that can be reached at, while packets are routed to a foreign agent in the newly visited network. The foreign agent ensures the delivery of these packets to the mobile node, as described in (Adrangi and Levkowetz, 2005b). This work was succeeded by that of (Vaarala and Klovning, 2008b), which utilizes two home agents to allow MIP to traverse multiple VPN gateways. (Chen *et al.*, 2006) proposed the dynamic assignment of the external home agent to reduce the delay caused by a handover.

The IPsec security association needed to be renegotiated with an IP address change until MOBIKE was developed, which enabled the IPsec in tunnel mode from preserving the security association during a layer 3 handover, according to (Eronen, 2006). This, essentially, allowed for multi-homing in mobile nodes. Likewise, our work, although differently, preserves the TLS session across layer 3 handover events.

A mobile VPN was developed by (Huang *et al.*, 2005) to support real-time applications using the Session Initiation Protocol (SIP). A VPN session here is identified by the SIP session address instead of by an IP address. The VPN server has an SIP proxy that maintains a binding between a SIP session ID and a mobile node IP address. The mobile node, after roaming, can send a request to update the address binding. Our work resembles the SIP-based mobile VPN in the fact that we identified VPN sessions using TLS session IDs and allowed mobile nodes to update the VPN server with the new IP address.

(Koponen *et al.*, 2006) extended TLS and SSH to support mobility. Their extension allows for a TLS session renegotiated in an abbreviated manner after an IP address change.

OpenVPN is a TLS-basd VPN that requires a full TLS handshake after a physical IP address change. It allows a client to preserve the same virtual IP across layer 3 handover events if it presents the same TLS certificate, according to (Yonan, 2008). However, the full TLS handshake only occurs after an inactivity timer is triggered.

A solution to solve in OpenVPN was presented in the work of (Zúquete and Frade, 2010). Similar to our work, when a change of the IP address is detected, they send ping messages accompanied by the TLS session ID in order to update the UDP address of the client's VPN session information at the VPN server. Our work differs from this work in the following ways: 1) they detected the IP address change of the *tun* interface, whereas we preserved the IP address of the *tun* interface (virtual IP), and we detected the change of the IP address of the physical interface, 2) the signal to update the UDP address binding is sent over the control channel which is a reliable layer, instead of sending it through the unreliable data channel, 3) because OpenVPN uses TLS, which does not protect the IP layer, we send and encrypt the new IP address in the update signal message to prevent a middle-man from performing IP spoofing, 4)

we allow the VPN server to use the update message signal to inform all connecting clients of a server's IP address change. As of now, our work is the only work as of now that allows a TLS-based VPN server from preserving the VPN sessions of connection clients when its IP address changes.

The work of (Heydari *et al.*, 2016) introduced a VPN framework that hides the VPN server from attackers using a Mobile-IPv6 based MTD. This work, in fact, is motivating to our work, as our work enables for an MTD-protected VPN server without the need for Mobile-IPv6.

3.3 Background

OpenVPN is one of the most widely used VPNs since it is open source, free to use, and can easily navigate through NATing boxes without any additional infrastructural changes. However, the support of mobility in OpenVPN is lacking, and we aimed to overcome this in our work.

Before presenting our solution, in this section, we introduce a brief background on how OpenVPN operates.

OpenVPN is implemented as a user-space process that interacts with the TCP/IP stack through a virtual network interface, a TUN interface for a switched network mode (Layer 3), or a TAP interface for a bridged mode (Layer 2). Our work was focused on the former mode, where the TUN interface is assigned a virtual IP by the VPN server. For a setup with multiple VPN clients, OpenVPN uses a TLS-based mode to authenticate the connecting VPN clients and subsequently to negotiate session keys to encrypt and/or authenticate data packets.

Two communication channels are set up when a VPN client connects with a VPN server. The control channel is first set up starting with a full TLS handshake followed by several messages exchanged for the configuration of the VPN tunnel. Our

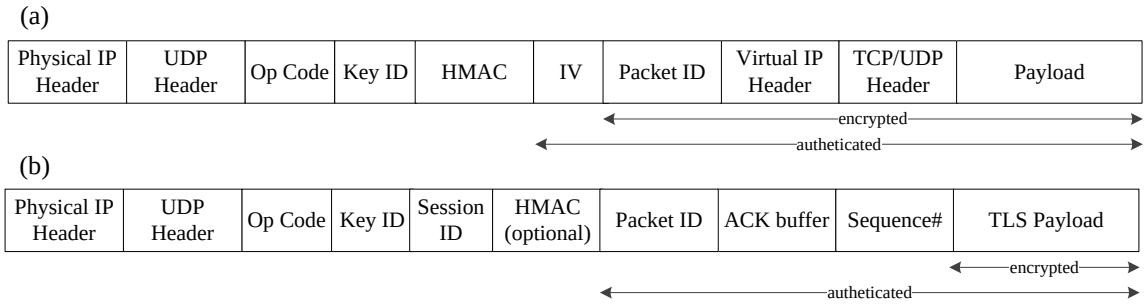

Figure 3.1: The Format of OpenVPN Packets: a) Data Packet, B) Control Packet.

experiment of OpenVPN 2.2.2 showed, in Wireshak, 132 packets exchanged between the OpenVPN client and server to establish the VPN tunnel with 33 RTTs. The control channel implements a reliability layer by acknowledging control packets, which is required by the TLS protocol. A data channel is then constructed after negotiating session encryption keys. This channel is not reliable, and therefore, no acknowledgments are sent back upon the reception of a data packet.

Figure 3.1 shows the format of data and control packets. One key difference is that data packets, unlike control packets, do not contain the TLS session ID. A UDP socket is created between the VPN client and server to transport both the control and data packet. The UDP address of the client is used by the VPN server as the identifier of the VPN session. The data packet, as indicated by its format in figure 3.1, first goes through the TUN interface and then gets encapsulated and sent out using the physical interface using the aforementioned UDP socket.

Once a VPN client changes its physical IP address, the VPN server will not be able to locate the VPN session information and, therefore, it will not be able to decrypt any packets sent from the client. The VPN client, being the only active party, will have to wait for an inactivity timeout to be triggered, at which time, a hard reset signal will be thrown to trigger a full VPN connection handshake exactly similar to the initial connection.

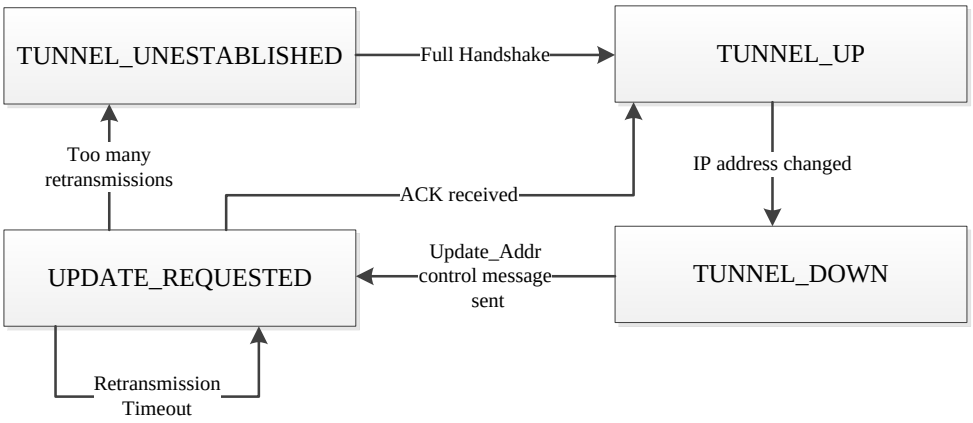

Figure 3.2: Lightweight VPN Resumption Finite State Machine of the VPN Client.

3.4 System Model

In this section, the models of our mobility-aware variant of OpenVPN is discussed.

3.4.1 Lightweight VPN resumption model

We modeled our lightweight VPN resumption as a finite state machine in both the VPN client and server.

VPN client model

The VPN client model is illustrated in Figure 3.2. The client starts with in the `TUNNEL_UNESTABLISHED` state. After a full TLS/VPN handshake, the VPN process enters the `TUNNEL_UP` state. When the physical IP address changes due to mobility, the VPN enters the `TUNNEL_DOWN` state.

The VPN then sends a control message requesting the VPN server to update the UDP address binding with the new IP address. After that, the client enters the `UPDATE_REQUESTED` state. The reception of an acknowledgment from the VPN server indicates that it was able to find the appropriate VPN session information as it was

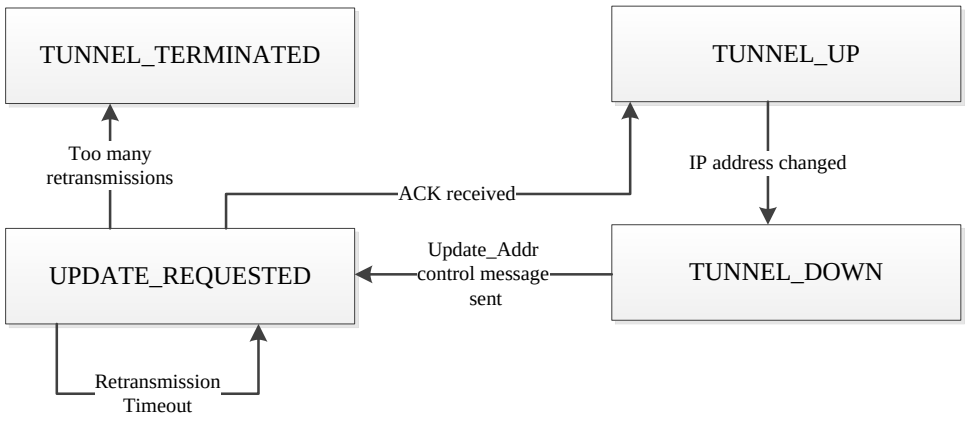

Figure 3.3: Lightweight VPN Resumption Finite State Machine of the VPN Server.

able to decrypt the control message. The VPN client can then move to the `TUNNEL_UP` state.

The absence of acknowledgment from the VPN server will take the VPN client back to the `TUNNEL_UNESTABLISHED` state in which the VPN will attempt to perform a full handshake as the original OpenVPN does.

VPN server model

The VPN server model is very similar to the VPN client model, as shown in Figure 3.3. The difference is that the VPN server cannot initiate a full handshake as this is the responsibility of the VPN client. Therefore, the VPN server starts in the `TUNNEL_UP` state. Moreover, when the VPN server's IP address is changed deliberately as a result of an MTD mechanism, an unacknowledged request to update the IP address binding at the VPN client will result in the VPN server terminating the VPN session. We noted that the VPN server maintains a unique state per VPN session (VPN client).

3.4.2 Attack model

We took into consideration a man-in-the-middle attack model, in which the attacker can passively monitor the VPN traffic. The attacker can identify an update

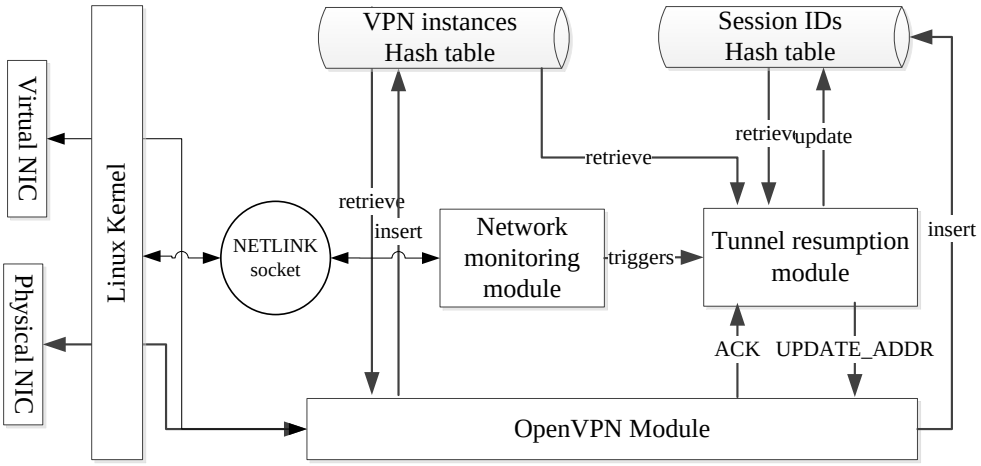

Figure 3.4: The Design of Fast VPN Resumption System.

address control message via inspecting the unencrypted *OpCode* and the length of the packet. Once successful, the attacker can alter the IP address of the outer IP header. The receiving endpoint of the update message may bind the VPN session with the altered IP address. The attacker, at this point, has successfully denied service from the update address requester.

3.5 System Design

Our design introduces two new modules to OpenVPN: a networking monitoring module and a tunnel resumption module which interacts with OpenVPN that we considered in our design as a one module.

These three modules interact with each other as presented in Figure 3.4. The current tunnel state, as described in section 3.4, is stored in a variable as part of a VPN instance's context. The modules we we added will now be described.

3.5.1 OpenVPN module

This module is responsible for fully establishing the tunnel in addition to sending and receiving packets in an encrypted and authenticated manner. In the VPN server's

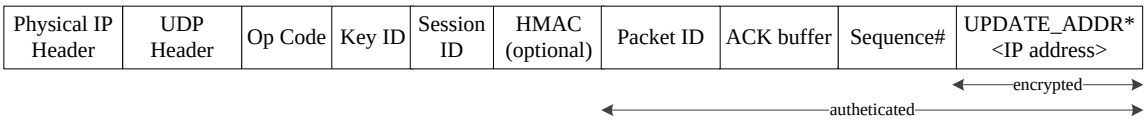

Figure 3.5: The Format of UPDATE_ADDR Control Message.

case, a hash table is used to save the VPN session information. Each entry of this hash table consists of a hash of the UDP address as the entry key, and the entry value is a $VPN\_instance$ record that contains the the VPN session information including, cryptographic keys and the IP of the client for verification purposes.

We made a modification to this module by introducing another hash table where the key of entries is the TLS session ID, and the value is the hash of the UDP address. After a full handshake is concluded between a VPN client and server, the VPN server adds a new entry into the session IDs hash table. This modification is added to the VPN server. The VPN client maintains a single $VPN\_instance$ and thus does not need a lookup table.

3.5.2 Network monitoring module

This module is designed to detect any changes to the IP address of the physical network interface, or if the packets are being routed through a different network interface. Once such an event is detected, this module sets the tunnel state variable to `TUNNEL_DOWN`, and triggers the tunnel resumption module. As Figure 3.6 shows, this module helps eliminate the idle time which in OpenVPN lasts until the inactivity timeout triggers.

3.5.3 Tunnel resumption module

This module is the brains of this project. Once it is triggered, it checks whether this VPN process works in a client mode or a server mode.

In the client case, it sends an `UPDATE_ADDR` control message to the VPN server. We defined the format of the update message as illustrated in Figure 3.5. The VPN server will not at first be able to locate the appropriate $VPN\_instance$ in order to process the packet since the packet is sent from a new IP address.

The VPN then inspects the `OP_CODE` which is unencrypted. If it determines that this is a control message, it uses the packet's session ID to perform a lookup on the session IDs hash table. If found, the old UDP address is retrieved and used to find the correct $VPN\_instance$. At this point, the TLS payload is decrypted and checked for 1) the existence of the command `"UPDATE_ADDR"` and 2) followed by an IP address that matches the IP address in the IP header. The second check is to account for the attack model presented in section 3.4.2. If any of the two checks fails, the update message is ignored and dropped without acknowledgment. If the two checks hold, the VPN server updates the session IDs hash table with the new IP address. It also updates the $VPN\_instances$ hash table with the new IP address as well as updating the relevant information in the $VPN\_instance$ with the new IP address.

This is done because OpenVPN, when processing a packet, always checks that the IP address of the packet matches the IP address in the $VPN\_instance$. Finally, an acknowledgment is sent to the VPN client indicating a successful IP update.

In the server case, it goes through the $VPN\_instances$ hash table sequentially, and sends an `UPDATE_ADDR` control message to every connecting client. Upon receiving the message at the client side, the client unlike the server does not perform any lookups and, instead, tries to decrypt the packet with the only $VPN\_instance$ it maintains. If the message is found to be an `"UPDATE_ADDR"` and the two checks from above holds, the IP address information in the $VPN\_instance$ record is updated with the new IP address. An acknowledgment is then sent to the VPN server to confirm the success of the IP address update.

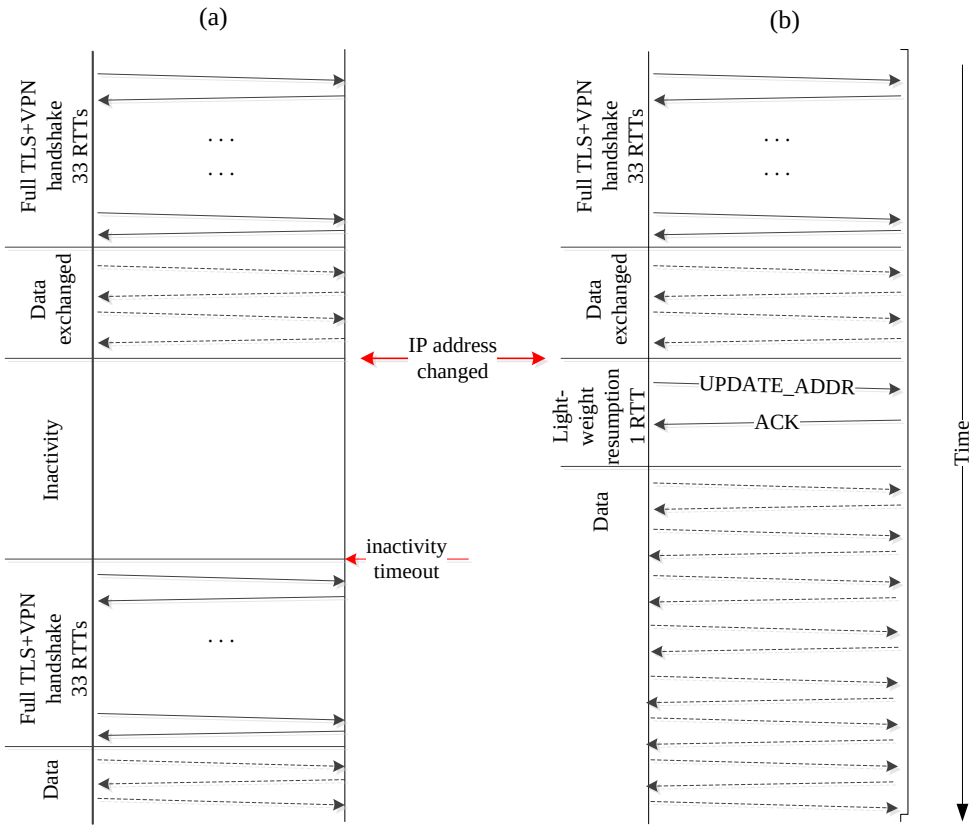

Figure 3.6: OpenVPN Tunnel Resumption Vs. Our Lightweight Tunnel Resumption.

In both cases, `"UPDATE_ADDR"` messages are retransmitted if not acknowledged. After a configured number of retransmission times, the pursuit to update the IP address stops. The VPN server just terminates the $VPN\_instance$ in question, while in the case of a VPN client, it returns the `TUNNEL_UNESTABLISHED` state where it attempts to perform a full handshake.

3.6 Implementation

3.6.1 Connection Monitor Module

The aim in this module is to provide for an early detection of when the tunnel is broken. It controls two *context* boolean variables: *network_connected* and *network_switched*. We used three mechanisms that update the values of these variables.

The first mechanism is OpenVPN's pinging mechanism that is controlled by configuration parameters to detect tunnel breakdown due to absence of activity. We modified the OpenVPN code to reduce the threshold in order to enable early detection of tunnel breakdown. A triggered timer will set *network_connected* to *false*, and the variable is reset to *true* as soon as an activity is registered.

The second mechanism detects network disconnections through the error messages reported by TCP/IP to the VPN's socket. We looked for four errors in particular: `EHOSTDOWN` (host down), `EHOSTUNREACH` (no route to host), `ENETDOWN` (network is down), `ENETUNREACH` (network is unreachable). If any of these errors are reported, the *network_connected* is set to *false*.

In the third mechanism, we implemented a *NETLINK* socket connection with Linux kernel through which any changes to the physical network interfaces are reported to MobiVPN. We look for the following events: `RTM_NEWADDR`, `RTM_DELADDR`, `RTM_NEWROUTE` and `RTM_DELROUTE` to detect when a network interface goes up or down or when it's IP address changes which sets the *network_switched* to *true*.

When a change of IP address is detected, the module registers a `SIGUSR2` signal, which triggers the tunnel resumption by the tunnel management module.

3.6.2 Tunnel Management Module

The implementation of the tunnel resumption module includes an algorithm that implements the logic explained in section 3.5.3. When updating the VPN instance information with the new IP address, the VPN server updates the to_link_addr in both the level 2 context structure and in tls_multi, as well as the $remote\_addr$ in all keys records. The VPN client updates the address in $link\_socket\_info$.

This module implements Algorithm 1, which is called by the network monitor module when a network event occurs. The algorithm handles two cases. The first

case is where the network is connected but not switched. Here, there is no need to resume the VPN session as the mobile device did not acquire a new physical IP. In the second case , the mobile has switched to a different network. The VPN sends and `UPDATE_ADDR` message to initiate the light-weight VPN resumption protocol. We noted that in the case of the VPN server, the procedure *Resume_ VPN()* is called for every *vpn_ instace* in its instances hash table (*vi_ hash_ tbl*).

Algorithm 1 Tunnel Manager

```
1: procedure RESUME_VPN(c : context)
2:     if ¬c.network_switched and c.network_connected then
3:         c.tunnel_state ← TUNNE_UP
4:     else if c.network_switched and c.network_connected then
5:         id = SEND_CONTROL_CHANNEL_STRING(c,"UPDATE_ADDR *
   ", c.real_IP)
6:         c.tunnel_state ← UPDATE_REQUESTED
7:         c.resume_timer = now + resume_timeout
8:         c.resume_packet_id = id
9:     end if
10: end procedure
11: procedure CHECK_INCOMING_CONTROL_CHANNEL(c : context)
12:     cp = GET_INCOMING_CONTROL_PACKET(c)
13:     if cp.Op_Code = P_ACK_1 then
14:         if c.tunnel_state ← UPDATE_REQUESTED and cp.packet_id =
   c.resume_packet_id then
15:             c.tunnel_state ← TUNNEL_UP
16:         else
17:             if now > c.resume_timer then
18:                 c.tunnel_state = TUNNEL_UNESTABLISHED
19:                 REGISTER_SIGNAL(c, SIGUSR1)
20:             end if
21:         end if
22:     else
23:         if cp.Op_Code = P_Control_1 then
24:             vpn_instance ← LOOK_UP_HASH_TBL(c.vi_hash_tbl, cp.ip)
25:             if vpn_instance = NULL then
26:                 sid ←GET_SESSION_ID(cp)
27:                 old_IP ← LOOK_UP_HASH_TBL(c.si_hash_tbl, sid)
28:                 vpn_instance ← LOOK_UP_HASH_TBL(c.vi_hash_tbl, old_IP)
29:             end if
30:             if vpn_instance == NULL then
31:                 Drop cp; return
32:             end if
33:             p ← DECRYPT(vpn_instance.context, cp)
34:             if p.paylaod begins with "UPDATE_ADDR" then
35:                 if p.ip = p.appended_ip then
36:                     UPDATE_CONTEXT_IP(c, p.ip)
37:                 else
38:                     Drop p; return
39:                 end if
```

Algorithm 1: Tunnel Manager (Continued)

```
40:             else
41:                 ...
42:             end if
43:         end if
44:     end if
45:     .....
46: end procedure
47: procedure UPDATE_CONTEXT_IP(c : context, ip)
48:     if c.options.mode = MODE_SERVER then
49:         UPDATE_IP(c.c2.to_link_addr, ip)
50:         UPDATE_IP(c.c1.ks.remote_addr, ip)
51:         UPDATE_IP(c.c2.tls_multi, ip)
52:     else
53:         UPDATE_IP(c.c2.link_socket_info, ip)
54:         UPDATE_IP(c.c2.accept_from, ip)
55:     end if
56: end procedure
```

3.7 Evaluation

3.7.1 Security Evaluation

Here, we evaluate our work against the attack model presented in section 3.4.2. The MITM attacker, can definitely intercept an update packet, and replace the IP address in the IP header with his or her own IP address, or with a bogus IP just to deny the update receiver from updating its context to the correct address of the update sender. Our protocol is resilient to such an attack since the new IP address is appended to the `UPDATE_ADDR` command in TLS payload. This IP address is protected by TLS from tampering and only the real owner of the TLS keys can create such a packet.

The attacker can keep a copy of a legitimate update message and replay it at a later time when the IP in this packet is no longer the current IP of the VPN sender. Our system is also resilient to this attack as OpenVPN already implements a replay attack counter measure using a monotonic packet ID and an acceptable receive window at the receiver side.

3.7.2 Performance Evaluation

3.7.3 Testbed Setup

We setup the testbed as shown in Figure 3.7. The VPN server and the application server are installed in virtual machines running on VMware Fusion for Macbook. These VMs run Ubuntu 16.04 with 2GB RAM. The client VM and runs Ubuntu 12.04 with 2GB RAM. In the local testbed, the VM is hosted in the same servers' Macbook, whereas in the distant testbed, it is hosted on a separate Macbook.

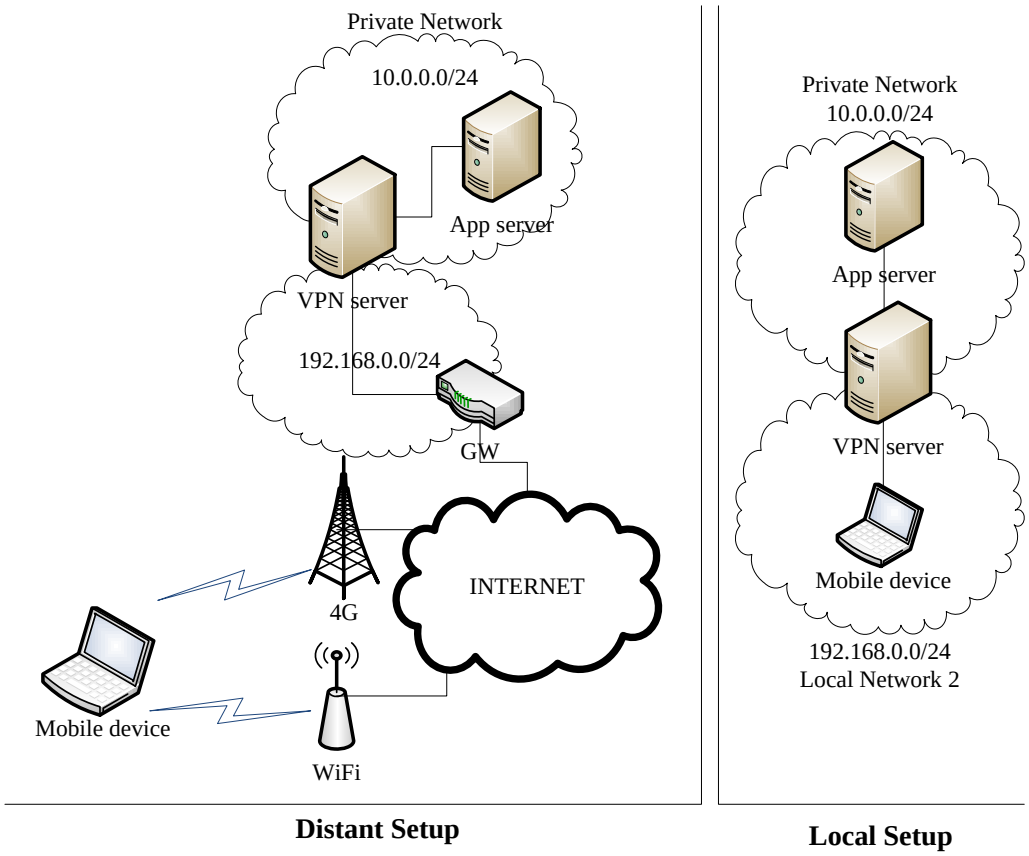

Figure 3.7: The Setup of the Evaluation Testbeds.

The VPN server is connected to two networks, a private network (10.0.100.0/24) along with the application server. The VPN client communicates with the VPN server through another local network (192.168.100.0/24) in the local testbed, or through the Internet in the distant testbed.

In the local testbed, we emulated a client mobility event or a VPN server migration event by changing the IP address of their respective physical network cards using the $ifconfig$ command. In the distant testbed, we triggered a mobility event by turning off the Wifi interface and turning on the cellular interface. The cellular connection is obtained by USB tethering. In all of our experiments, we took the average of 5 trials.

3.7.4 VPN Tunnel Resumption at the VPN client

In order to evaluate the performance effect of our work, we conducted our experiment over the local and distant testbed setup. The experiment was conducted by sending data using iperf through the VPN tunnel between the mobile client and the application server. After 7 seconds of data transfer, a one mobility event is triggered. We sent 200MB in the local setup, and 40 MB in the distant setup. The mobility event causes the mobile client to get a new IP address.

Figure 3.8 shows the amount of data transfer over time using OpenVPN with the timeout set to the recommended default value of 60 seconds denoted as (OpenVPN-60s). The same was done with the timeout of OpenVPN set to the minimum value possible of 3 seconds denoted as (OpenVPN-3s). We also ran the same test on MobiVPN.

Figure 3.8 shows one trial of data transmission over the local testbed. After the mobility event, OpenVPN-60s took 64.123 seconds to re-instantiate the VPN session. The data transfered was delayed even further due to TCP retransmission backoff, which is addressed in Chapter 4. The data transfer took 130.1 seconds. OpenVPN-3s took 6.907 seconds to re-instantiate the VPN session. The data transfer took 33.1 seconds. MobiVPN was the fastest with 222 milliseconds to resume the VPN session including the time to detect the change of IP address. The data transfer took 24 seconds. MobiVPN was able to decrease the time to resume the VPN by 99.65% compared to OpenVPN-60s, and by 96.79% compared to OpenVPN-3s.

Performing a similar experiment over the distant testbed resulted in MobiVPN surpassing both OpenVPN-3s and OpenVPN-60s. Figure 3.9 shows one trial of data transmission over the local testbed. Figure 3.10 shows a comparison of the time it took to resume the VPN tunnel after the mobility event. MobiVPN clearly outper-

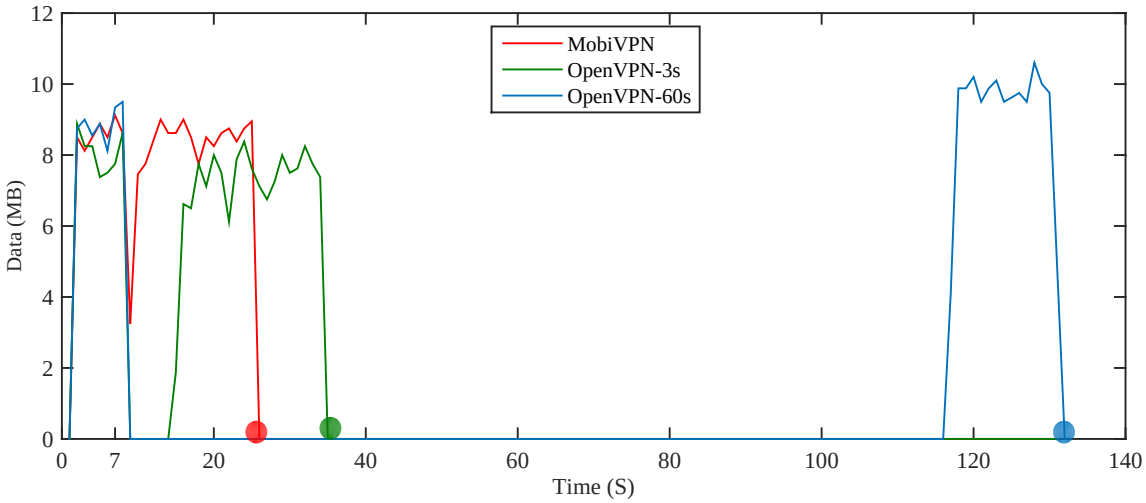

Figure 3.8: Effect of Fast VPN Resumption on Data Transfer - Local Testbed.

formed OpenVPN in both of its configurations. Table 3.1 summarizes our performance measurements and shows the percentage of decrease in time to accomplish both the VPN tunnel resumption and the data transfer. We noted that these savings in data transfer time would increase if multiple network switching occurs.

During the experiment, we observed that sometimes the data transfer in the OpenVPN-3s was delayed for about 3 seconds due to the aggressive timer of 3 seconds as the tunnel was unavailable during an unnecessary attempt at restarting the tunnel.

Performing an aggressive timeout such as 3 seconds is not a practical solution for two reasons. 1) Figure 3.11 shows how active the VPN tunnel was with the aggressive timeout (3 seconds) during 60 seconds of user inactivity as opposed to MobiVPN. MobiVPN achieves faster tunnel resumption without consuming the tunnel during inactivity. This is quite an important feature especially for battery-constrained mobile phones. Sending unnecessary data over the tunnel prevents the radio module of mobile devices from going to sleep mode during inactivity, which, as a result, consumes more power. 2) Even if the mobile client sets an aggressive timeout, the VPN server can overwrite that, and pushes its default settings onto the mobile client.

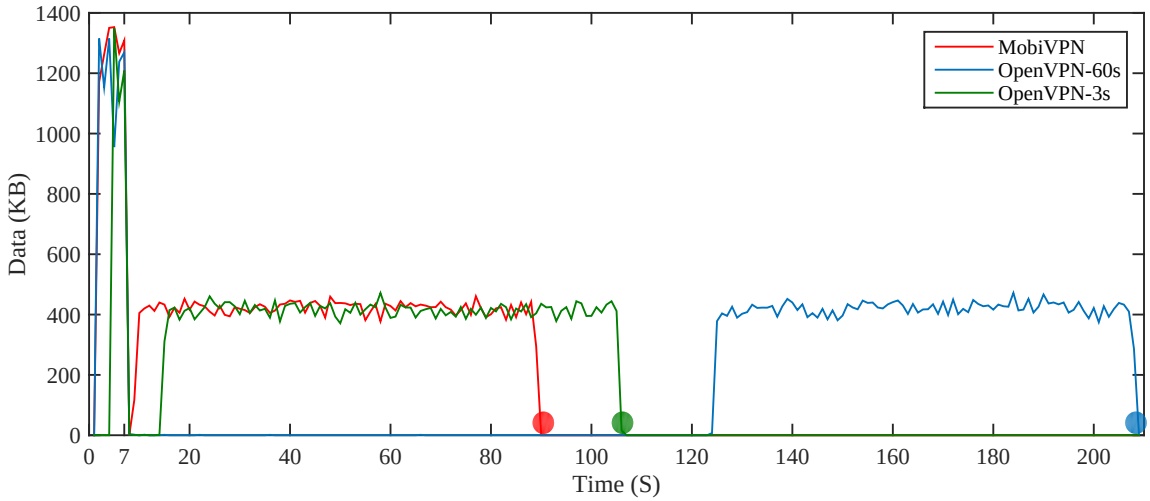

Figure 3.9: Effect of Fast VPN Resumption on Data Transfer - Distant Testbed.

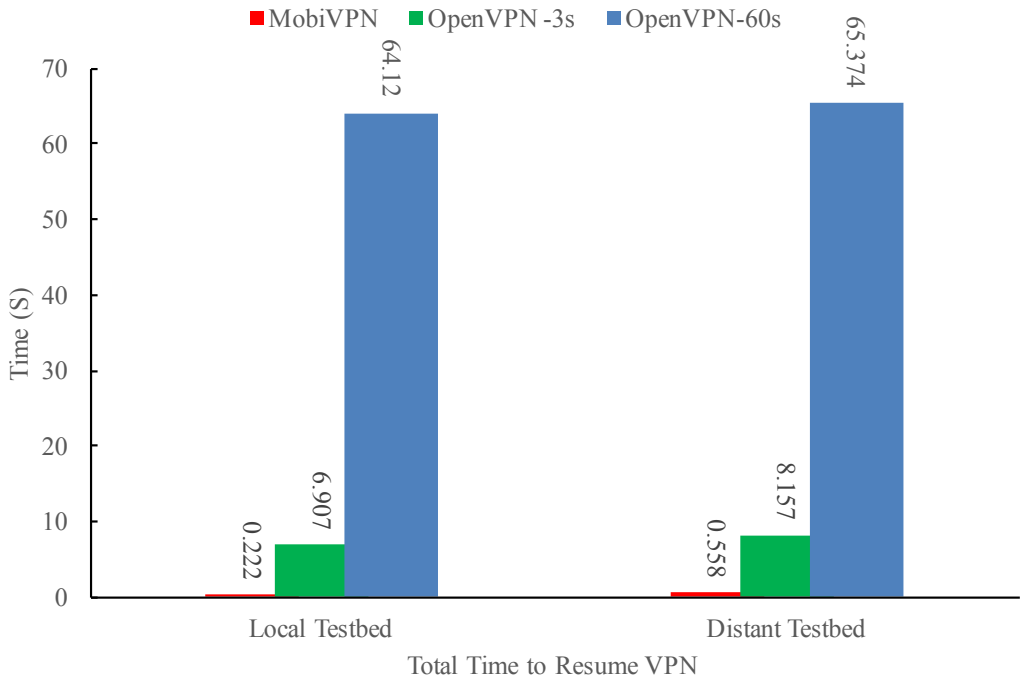

Figure 3.10: Total Time To Resume the VPN.

Table 3.1: Performance Measurements When the VPN Client Changes Its IP Address.

Testbed	**Measured Metric**	**MobiVPN**	**OpenVPN -3s**	% Decrease by MobiVPN	**OpenVPN -60s**	% Decrease by MobiVPN
Local	VPN Unavailability Time	214 ms	4.498 s	95.24%	61.711 s	99.65%
	VPN Session Resumption Time	8 ms	2.409 s	99.67%	2.412 s	99.67%
	Total Time To Resume VPN	222 ms	6.907 s	**96.79%**	64.123 s	**99.65%**
	Data Transfer Time (200MB)	24 s	33.1 s	**27.49%**	130.1 s	**81.55%**
Distant	VPN Unavailability Time	512 ms	4.710 s	89.13%	61.927 s	99.17%
	VPN Session Resumption Time	46 ms	3.462 s	98.67%	3.447 s	98.67%
	Total Time To Resume VPN	558 ms	8.172 s	**93.17%**	65.374 s	**99.15%**
	Data Transfer Time (40MB)	87 s	104 s	**16.35%**	207 s	**57.97%**

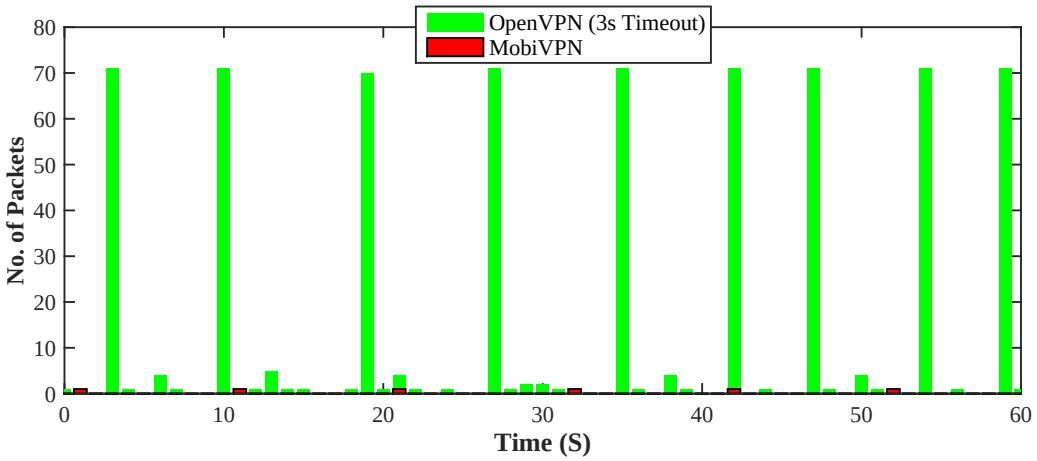

Figure 3.11: MobiVPN Vs. OpenVPN with Aggressive Timeout During Idle User Activity.

3.7.5 VPN Tunnel Resumption at the VPN server

In this experiment, we aimed to measure how long it took all connecting OpenVPN clients to resume the VPN tunnel after changing the IP address of the VPN server. We performed our experiment on the local testbed. We used one client machine but ran multiple OpenVPN client processes as each process is considered by the OpenVPN server as a unique VPN client. We configured the VPN server to accept the same certificate from multiple clients.

Since OpenVPN server cannot resume/re-instantiate a VPN session, as this is the role of the VPN client in OpenVPN's design, we emulated a mobility event at the OpenVPN server by issuing a *kill* command that terminates the VPN session of all connected clients since they have the same certificate's common name. This made all VPN clients re-instantiate the VPN session after their configured timeout.

Figure 3.12 shows the outcome of the experiment in the same three scenarios we explained in the VPN client experiment. The figure shows how MobiVPN server was able to inform the clients of its IP address change and resume the the VPN tunnel in a much smaller fraction of time than that of original OpenVPN. As more clients

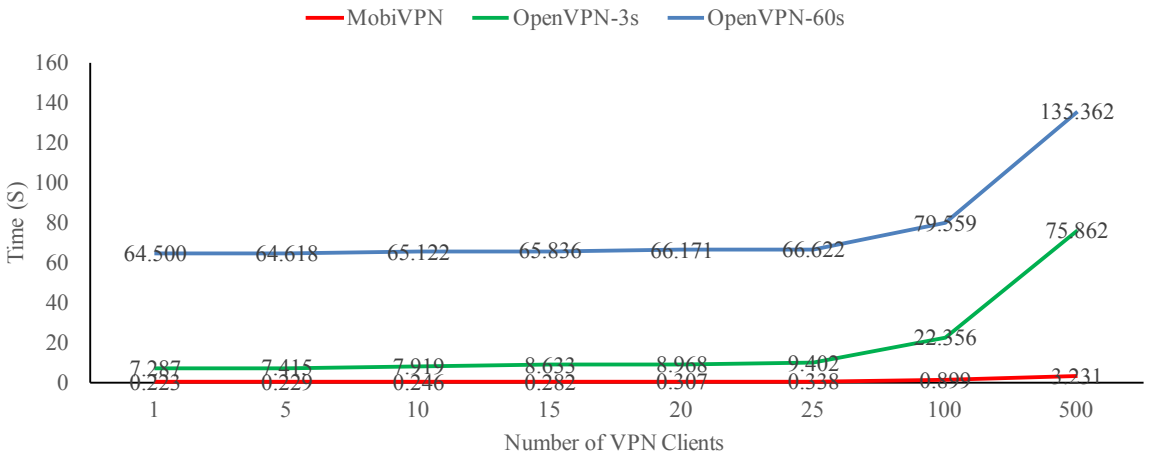

Figure 3.12: Evaluation of MobiVPN Vs. Original OpenVPN When the VPN Server Changes Its IP Address.

were connected, the time it took OpenVPN in both configurations to resume all VPN sessions was increasing in a much higher rate than that of MobiVPN.

In an MTD framework where servers IP addresses change frequently, resuming VPN sessions in OpenVPN becomes impractical. The VPN server's IP address may be due for a new change of IP address even before all clients have reconnected. In addition to the costly time it requires to reestablish the VPN sessions with the clients, packet loss can be another problem which we highlight in the next section.

3.7.6 VPN Resumption Impact on Packet Loss

In this experiment, our goal was to measure the packet loss when using our light-weight VPN session resumption as opposed to a full VPN handshake. We eliminated the timeout effect in this experiment by issuing a `SIGUSR1` signal through the management interface to perform a full VPN handshake. Our light-weight handshake was similarly triggered by issuing a `SIGUSR2` signal.

We performed the experiment by streaming UDP packets over the VPN tunnel using *iperf* at 1Mbps sending rate for 60 seconds. We performed 5 cases, where in

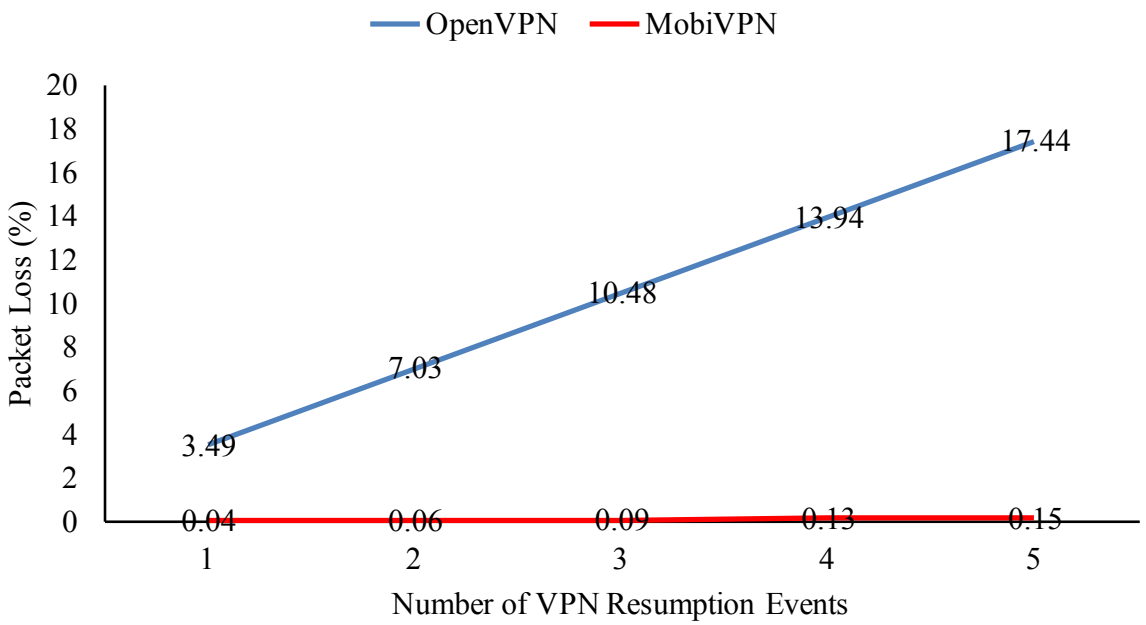

Figure 3.13: Data Loss Caused By OpenVPN's Full Handshake Vs. MobiVPN Lightweight Handshake.

each case the number of signals triggered is increased by one. Figure 3.13 shows the significance reduction of packet loss in MobiVPN compared to OpenVPN.

3.8 Conclusion

In this chapter, we presented a new protocol for a light-weight VPN session resumption that allows both MobiVPN client and server to resume an already-established VPN tunnel.

The evaluation we performed on our implementation of the protocol showed the feasibility of employing it for mobile VPNs and MTD-enabled VPN servers. The time it took to resume a VPN tunnel after a mobility event was decreased in MobiVPN by an average of 97.19% compared to the time it took OpenVPN to re-instantiate the VPN tunnel in both timeout configurations.

Therefore, we believe our light-weight VPN resumption will improve the mobile user experience as well as enable VPN servers from utilizing MTD protection without negative effect on the connected clients.

Chapter 4

PERSISTENCE AND FAST RESUMPTION OF TCP-BASED APPLICATIONS

4.1 Introduction

The usage of mobile devices has seen enormous growth in recent years. Users no longer employ just their computers, but utilize their mobile devices to interface with computing resources. Establishing a secure and reliable connection between a mobile device and a protected network is of most importance.

Virtual Private Networks (VPN) are used widely as a solution that provides secure and private connection between two end-points. However, conventional VPNs are designed to work best for stationary devices which, unlike mobile devices, do not experience frequent network disconnections. Mobile devices are susceptible to intermittent connection loss while switching from one network to another or experiencing a gap in coverage as indicated in (Alshalan *et al.*, 2016b; Dinh *et al.*, 2013). Such network disruptive events can cause the VPN connection to break, causing a possible termination of applications communicating through the VPN.

The work that was presented in Chapter 3 enables the VPN session to resume as soon as network connectivity is restored. However, during disconnection periods like the ones illustrated in Figure 4.1, applications' TCP sessions sending rate will drop due to TCP interpreting the disconnection as a congestion event. TCP also employ a retransmission timeout where the resumption of packet transmission is delayed even after the resumption of the VPN session. Moreover, TCP session may terminate if the connectivty is not restored in a timely manner.

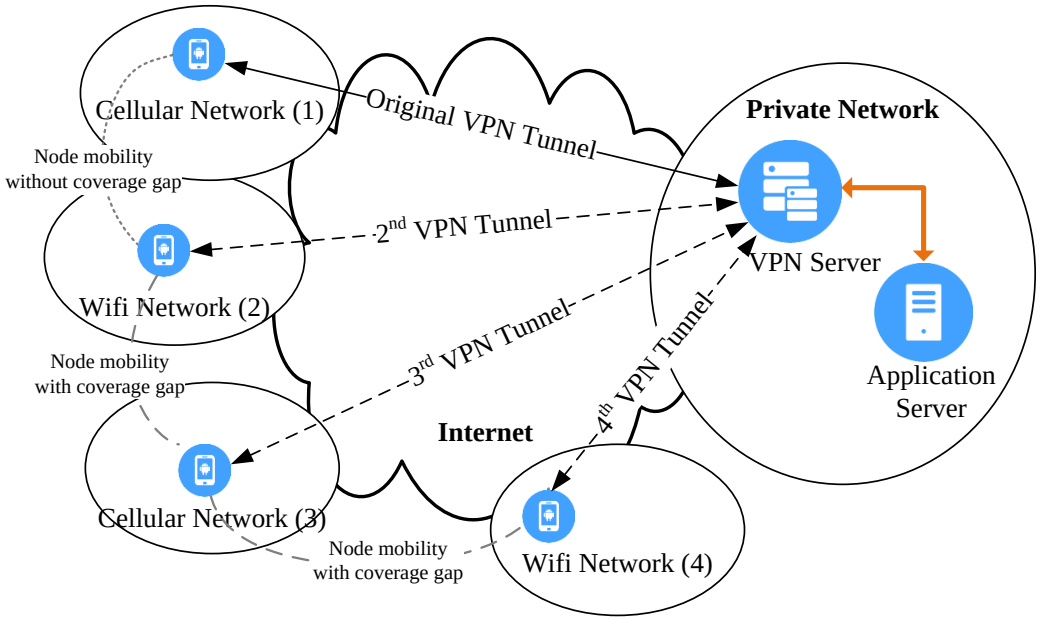

Figure 4.1: Overview of the Network Infrastructure.

The impact of one network disconnection event on a TCP flow is discussed in more details in Section 4.3.

Although, OpenVPN implements a persistence feature that can reestablish a VPN tunnel preserving the virtual IP if the VPN client uses the same TLS certificate, which may help in maintaining the tunneled applications' sessions, a more robust system would resume the VPN tunnel as soon as network connectivity is restored and hide the disconnections and the resumption of the VPN tunnel from the applications.

In this work, a persistence and fast resumption feature was developed for TCP flows that are tunneled through MobiVPN by modifying OpenVPN 2.2.2. A scheme to overcome the problem of disconnecting mobile clients was introduced. The contributions in this chapter include:

- Handling the loss of VPN tunnel connectivity in a way that makes both application clients and application servers unaware of the VPN disconnection.

- Preventing tunneled TCP flows from terminating during disconnection periods allowing them to survive these disconnections.

- Maintaining the sending rate of the tunneled TCP flows by buffering and acknowledging TCP packets on behalf of the remote applications.

- Suspending TCP flows during disconnections; where non-buffered TCP flows are suspended immediately whereas buffered flows are suspended after enough packets have been buffered to recover TCP's sending rate.

- All TCP flows are resumed immediately after reconnecting the VPN tunnel without having to wait for a retransmission timeout, and without the need to modify TCP in either the mobile client or the application servers.

The experiments performed in our evaluation showed an increase in throughput for a tunneled TCP flow as high as 54%, with an average throughput increase by 30.08% when buffering is used, and by 20.93% when the buffering option is disabled. Therefore, our MobiVPN is able to provide a better mobile user experience by overcoming the side effects that are associated with intermittent network disconnection due to mobility.

In the remainder of this chapter, other related work in this area is presented in Section 4.2. The motivation behind MobiVPN is discussed in Section 4.3. The model of MobiVPN is presented in Section 4.5. In Section 4.6, a detailed design of our MobiVPN is presented. The implementation of this design is presented in Section 4.7. In Section 4.8 the outcome of the performance evaluation is provided and discussed. Section 4.9 summarizes the conclusions of this work.

4.2 Related Work

The mobility in VPNs and how to solve the problem that arises during network roaming when the mobile device joins a new network and obtains a new physical IP has been addressed in the literature. Solutions to this problem have revolved around finding ways to route packets destined for the old physical IP address to the new physical IP address. The solutions do not offer persistence to TCP sessions that are tunneled through the mobile VPN. TCP sessions can be maintained only if the network disconnection is not long enough for TCP sessions to time out. TCP also would suffer from a performance hit with every disconnection event due to the droppage of the congestion window and the transmission idle periods dictated by retransmission timeouts.

To address the mobility effect on TCP, several solutions have been presented in the literature. However, none of these ideas have been incorporated in mobile VPNs. Our MobiVPN addresses both the network roaming and the TCP persistence problems. The related work in both areas is discussed in the following sections.

4.2.1 Mobility in VPN

To address the effect of mobility in VPN, researchers and engineers have tackled this problem at different layers of the TCP/IP stack according to (Alshalan *et al.*, 2016b).

In the network layer, Mobile-IP, in conjunction with IPsec, was introduced as a mobile VPN solution. Mobile-IP (MIP) uses a fixed home agent for each mobile node and routes the packets to the foreign agent in each visited network. This causes the triangular routing anomaly (Alshalan *et al.*, 2016b). MIP with two home agents was proposed in (Vaarala and Klovning, 2008b) to solve the problem of MIP traversal of

multiple VPN gateways. In their studies, (Benenati *et al.*, 2002) and (Feder *et al.*, 2003) used a variant of (Vaarala and Klovning, 2008b) to build a a transport layer mobility support across wireless and cellular networks. The MOBIKE protocol, which was introduced by (Eronen, 2006) allows IPsec security association to support multiple IP addresses.

In the application layer, (Huang *et al.*, 2005) utilized SIP protocol to build a mobile VPN that supports real-time applications. The addressing used for the application session uses SIP session addresses. An SIP proxy in the VPN gateway, updates the binding between the SIP address and the mobile node's new physical address. SIP and MOBIKE were combined by (Dutta *et al.*, 2005) in their mobile VPN framework.

A solution for a seamless mobility of OpenVPN clients moving across WiFi hotspots was presented by (Zúquete and Frade, 2010). The OpenVPN tunnel is reconfigured after a client gets a new IP address post handover to a new network. The VPN tunnel context is updated at the VPN server when the client presents the previous session's ID after receiving a new IP address. This approach minimizes the packet loss but does not avoid it.

Mobility of SSH and TLS protocols is proposed in (Koponen *et al.*, 2006) and (Schonwalder *et al.*, 2009). Both protocols can be used in place of IPsec to provide the security requirements in VPNs. In (Schonwalder *et al.*, 2009), a session resumption concept is introduced in order so as to resume SSH sessions without having to renegotiate new session keys. In (Koponen *et al.*, 2006), extensions of the SSH and TLS protocols have been added to allow SSH and TLS sessions to survive long network disruption events. The extension allows applications running over an SSH or TLS protocols to resume a previously established connection regardless of the change of IP address. This is achieved by not binding the TLS or SSH session to the mobile's IP address. However, the TCP sockets are not maintained and new TCP sockets

are created after reconnection. Such a solution can be used by TLS based VPN for faster resumption of the VPN tunnel, but it does not help the TCP sockets tunneled through the VPN to survive the network disruption events.

MUSeS, which allows user connections to survive mobility-driven disruptions, was introduced by (Ahmat and Magoni, 2012). MUSeS creates a secure session using an application layer abstraction to hide the network disruptions from the user. It uses a peer-to-peer overlay network called CLOAK (Tiendrebeogo *et al.*, 2011) above any IP network. MUSeS relies on device identifiers provided and managed by CLOAK to provide encryption and authentication. The work of (Ahmat *et al.*, 2016) extends MUSeS and provides SEMOS which adds a layer on top of the CLOAK layer to maintain application sessions. This work, however, requires applications to use SEMOS APIs to create their sessions.

OpenVPN developed by (OpenVPN Technologies, 2011) addresses the problem of obtaining a new physical IP by providing the mobile node with a fixed virtual IP. After a disconnection event, the mobile node has to present the TLS certificate used in the previous session in order to maintain the same virtual IP. However, a fast reconnection after roaming to a new network is not supported by OpenVPN since it waits for an inactivity timeout to be triggered to reestablish the VPN tunnel.

A generic model for mobile VPN on Android was introduced by (Chunle *et al.*, 2016). This model define a finite state machine in which the VPN can be started, paused, restarted or stopped based on the network status. Although an FSM was similarly used FSM to model MobiVPN, this work is different. The mobile client is allowed to resume the VPN session in a light-weight manner, in addition to maintaining the TCP sessions that run through the VPN tunnel, which engages the VPN server.

4.2.2 Mobility in TCP

Mobility affects the performance of TCP in mobile environments that exhibit disconnections. To tackle this problem, (Brown and Singh, 1997) proposed M-TCP which splits the TCP connection between a Mobile Host (MH) and a Fixed Host (FH) at the Base Station (BS), just like I-TCP proposed by (Bakre and Badrinath, 1995). The BS buffers and acknowledges the packets sent from the FH to the MH. Using a wireless-optimized protocol, the BS ensures the delivery of these packets to the MH. During disconnections, the BS sends a zero window packet to suspend the FH.

The Zero Window Message (ZWM) concept was used by (Goff *et al.*, 2000) to freeze TCP. It is assumed in their work that the receiver is able to detect when it is about to disconnect and send a ZWM to pause the sender. The resumption of a disconnection is carried out by triggering a fast retransmission where the receiver sends three acknowledgments of the last segment received. This technique is promising, but its success relies greatly on reliable prediction of disconnection. While this may work by measuring fading signal rate as they suggest, abrupt disconnections may not be supported. Example of an abrupt disconnection is when the mobile device loses data connectivity due to the mobile user picking up a phone call, a behavior exhibited in several wireless providers. PETS, designed by (So-In *et al.*, 2009), leverages the TCP freeze concept and uses it in conjunction with MIP in a module employed in routers between the TCP client and server which performs the suspension on both ends during disconnections.

4.3 Motivation

Figure 4.2 shows how TCP reacts during a disconnection period when slow-start and AIMD are employed as congestion strategies as observed for instance in TCP

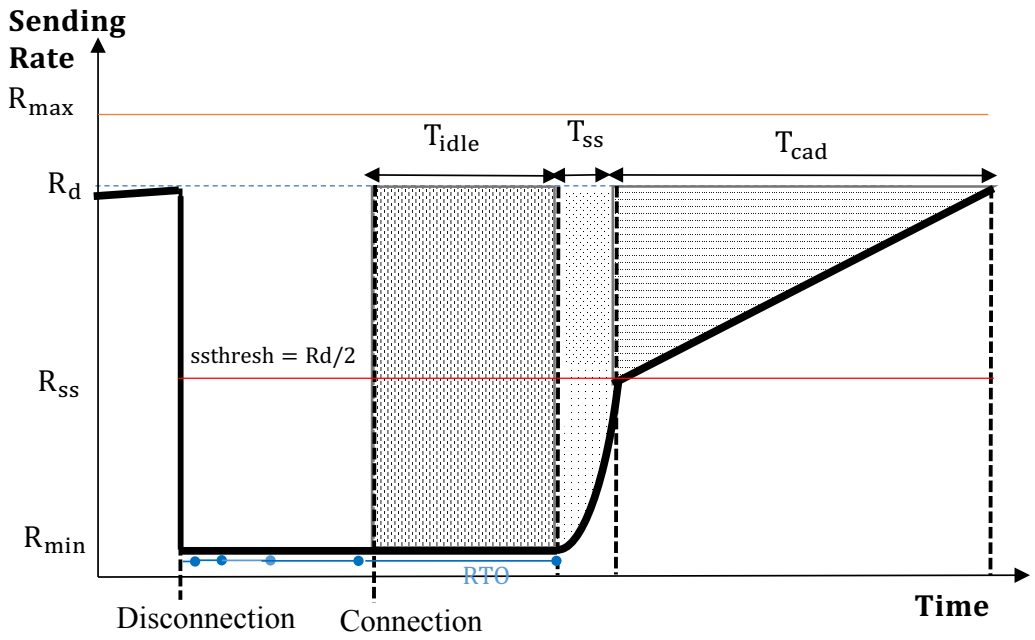

Figure 4.2: Effect of Mobility on TCP Sending Rate.

Reno. Given R_d as the current TCP sending rate right before the disconnection, R_{ss} as the sending rate at TCP's slow-start threshold (*ssthresh*) and R_{min} the minimum sending rate which is equal to the maximum segment size: the unused bandwidth due to a disconnection event is shown in the shaded area in Figure 4.2, and can be calculated in the following equation, where T_{idle} is the time TCP remains idle waiting for the retransmission timeout (RTO) to be triggered, T_{ss} is the time spent during the slow-start phase, T_{cad} is the time spent in congestion avoidance phase until TCP's sending rate reached R_d and RTT_c is the round-trip time after the reconnection:

$$\begin{aligned} Unused\_BW = &(R_d - R_{min}) \times T_{idle}/RTT_c \\ &+ \Sigma_{i=0}^{T_{ss}/RTT_c}(R_d - 2^i \times R_{min}) \\ &+ \Sigma_{j=0}^{T_{cad}/RTT_c}(R_d - (R_{ss} + j \times R_{min})) \end{aligned} \tag{4.1}$$

The T_{idle} value depends on how many RTO has been triggered before reconnection. RTO doubles every time it is triggered and may have an upper bound which varies based on the operating system in use, but has to be at least 60 seconds according to

(Paxson and Allman, 2000). Assuming RTO_d is the initial RTO before disconnection and T_d is the total disconnection time, T_{idle} can be calculated as:

$$T_{idle} = [(2^{\lfloor \log_2(\frac{T_d}{RTO_d}) \rfloor + 1} - 1) \times RTO_d] - T_d \tag{4.2}$$

We observed this idle period in both TCP Reno and TCP Cubic, the only congestion control algorithms supported by Linux kernel 3.2.

In a mobility scenario like the one illustrated in Figure 4.1, four disconnection periods occur resulting in the need to establish four VPN tunnels with full handshake, and underutilized bandwidth as shown in equation 4.1. The motivation behind MobiVPN is to resume the initial VPN session as soon as network connectivity is restored, and to resume TCP flows immediately with the same sending rate before disconnection. Doing so will eliminate the wasted bandwidth illustrated in the shaded area in Figure 4.2, which is not available in the mobile VPN solutions presented in the literature.

TCP congestion control is designed to deal with packet loss due to congestion in the network. However, a network disconnection or switching due to mobility will be treated the same as congestion when the network may not be congested. Our aim is to alleviate this penalty by hiding the network disconnection from TCP. Therefore, MobiVPN only interferes when there is a disconnection at the VPN layer, but not during congestion

4.4 Requirements and Assumptions

MobiVPN is required to maintain the application sessions between the mobile client and the application servers despite interruption of network connectivity, or when the mobile device moves between networks and obtains new IP addresses. In other words, network disruptions and network changes due to mobility should not terminate an application session.

In essence, our main goal with MobiVPN is to provide the application layer transparency to network layer disruptions so as to maintain independence of the end-to-end application sessions from issues caused by mobility.

4.4.1 Assumptions

We made the following assumptions in this work:

1. The mobile device does not experience either hardware or software failure. The mobile device may however lose its network connectivity for an unspecific period of time. The mobile device can also switch from one WiFi or cellular network to another at anytime.

2. Once the mobile device regains network connectivity, it may obtain a new physical IP address.

3. The VPN server has a reliable network connectivity, and is always available. If the VPN server loses its connection or fails, all open applications in the mobile device may lose their active sessions with the application servers.

4. There is a reliable communication channel between the MobiVPN server and the application servers. Our goal is to provide continuity of service when the VPN tunnel itself is unavailable because of client mobility.

5. Applications that use connection-less protocols such as UDP are not supported as they are not connection-oriented. Therefore, in order to maintain application sessions, the connectivity of their TCP flows is maintained. Applications that employ their own communication protocol are therefore not supported.

4.5 MobiVPN System Model

Presented in this section are the models of MobiVPN as follows:

4.5.1 MobiVPN Finite State Mode

MobiVPN is modeled as a finite state machine that has four possible states:

1. *Normal state:* Here the VPN tunnel is healthy and the applications' TCP sessions are behaving normally. MobiVPN behaves as OpenVPN normally would, with the exception that the buffering module will intercept and cache a copy of all packets being sent from buffered flows. These copies will be discarded when ACKs corresponding to the packets are received.

2. *Suspend state:* The VPN enters this state when the VPN tunnel fails due to network disruptions. MobiVPN caches and acknowledges packets coming from buffering-enabled applications and eventually suspends these applications when the in-flight packets consume a whole receiver's window. Non-buffered flows (applications) are suspended immediately.

3. *Resume state:* The VPN enters this state when the VPN tunnel is restored. The packet resending module sends out cached packets to the intended recipients until the buffer is cleared, at which point it resumes the suspended applications. Non-buffered flows are resumed immediately.

4. *Terminate state:* This is a final state in which the VPN terminates the tunnel and exits in the case of the VPN client. The VPN enters this state either by user request or when the persistence timer times out.

So we have:

$$MobiVPN\_State = \{Normal, Suspend, Resume, Terminate\}$$

We also have three states for the VPN tunnel as follows:

$$Tunnel\_State = \{Down, Up, Unestablished\}$$

Notice in the work of (Chunle *et al.*, 2016), the status of the network connectivity is considered, which serves their purpose as their model is only implemented in the mobile device side but not in the VPN server. In our model, both the VPN client on the mobile device and the VPN server are actors in our mobile VPN framework. Therefore, since the VPN server has no knowledge of the network status change of the mobile client, the status of the VPN tunnel is considered as the indicator of connectivity. It is also imperative to state that one of the design goals is for the VPN client and server to use the same model in order to reduce coding and increase interoperability.

The buffer introduced in MobiVPN has three possible states as follows:

$$Buffer\_State = \{Empty, Not\_empty, Full\}$$

It was decided that a new TCP flow in our system was to be either buffered or not buffered throughout the entire TCP session. Every buffered flow has a share of the buffer capacity which is determined using Equation 4.8. The state of each flow's buffer share is monitored. Therefore, the following is defined:

$$Buffering\_State = \{Enabled, Disabled\}$$

$$bShare\_State = \{sFull, Not\_sFull\}$$

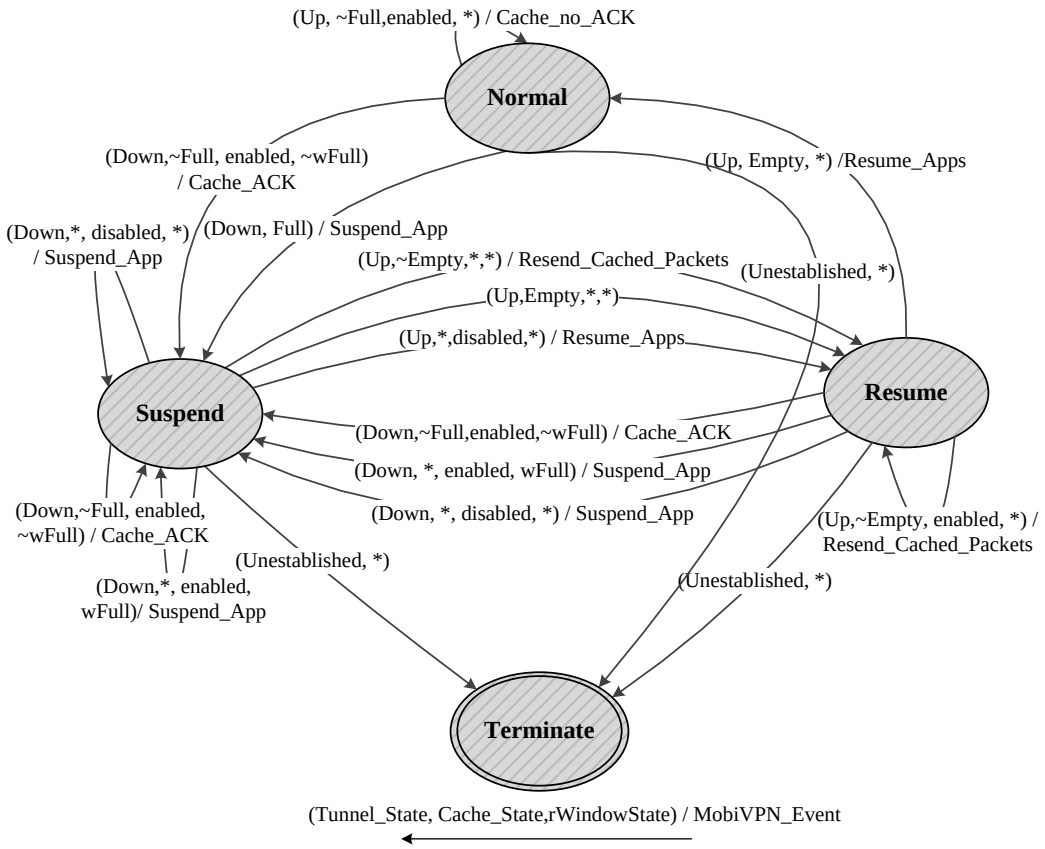

Figure 4.3: MobiVPN Finite State Machine.

In order to provide persistence to application sessions, four MobiVPN events is introduced as follows:

$$MobiVPN\_Event = \{Cache\_no\_ACK, Cache\_ACK, Suspend\_App, Resume\_Apps\}$$

The $Cache\_no\_ACK$ event is performed during the $Normal$ MobiVPN state in which outgoing packets are cached but not acknowledged. The $Cache\_ACK$ event occurs during the $Suspend$ state when the buffer's state and the flow's buffer share state are not full. Otherwise, $Suspend\_App$ is performed to suspend the packet sender. Finally, $Resume\_Apps$ is carried out when MobiVPN enters the $Resume$ state which is triggered by the tunnel state changes from $Down$ to Up.

Using a 7-tuple deterministic finite state transducer $(Q, \Sigma, \Gamma, \delta, \omega, q_0, F)$, where Q is a finite set of states, Σ is a finite set of input alphabet, Γ is a finite set of output

alphabet, δ is a transition function, ω is the output function, $q_0 \in Q$ is the start state and $F \subseteq Q$ is the set of accept states, MobiVPN is modeled as follows, and its state transitions are shown in Figure 4.3:

$$\begin{aligned} Q &= MobiVPN\_State \\ \Sigma &= Tunnel\_State \times Buffer\_State \times \\ &\quad Buffering\_State \times bShare\_State \\ \Gamma &= MobiVPN\_Event \\ \delta &: Q \times \Sigma \to Q \\ \omega &: Q \times \Sigma \to \Gamma \\ q_0 &= Normal \\ F &= \{Terminate\} \end{aligned} \tag{4.3}$$

4.5.2 Tunnel Management Finite State Model

To model the tunnel management, these self-explanatory events are defined:

$Network\_Monitoring\_Event = \{Network\_Disconnected, Network\_Connected,$
$Network\_Switched\}$

$Tunnel\_Management\_Event = \{Create\_Tunnel, Reconnect\_Tunnel, Destroy\_Tunnel\}$

$Tunnel\_Alert\_Event = \{Tunnel\_Up, Tunnel\_Down\}$

$Termination\_Event = \{User\_Terminate, Persistence\_Timeout\}$

Following the definition of FST in the previous section, a 6-tuple FST $(Q, \Sigma, \Gamma, \delta, \omega, q_0)$ is used to model the VPN tunnel states as follows, and show its state transitions in

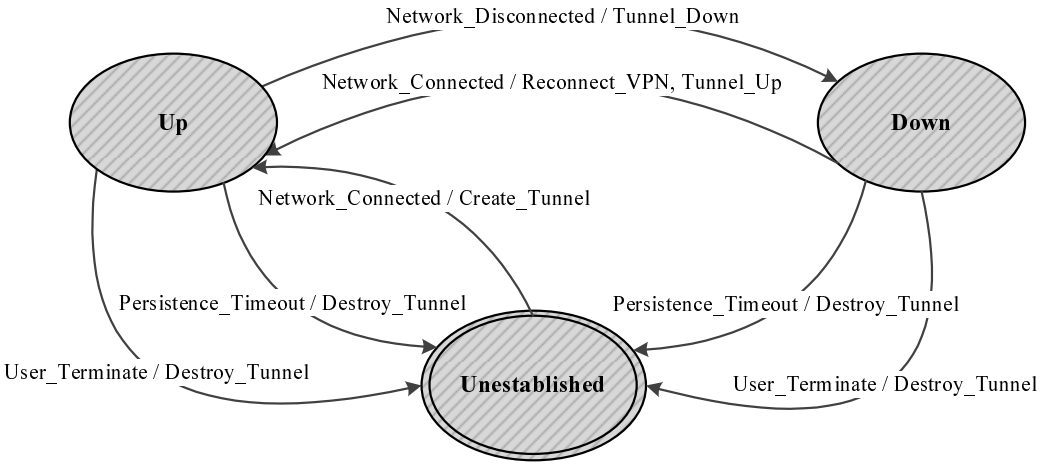

Figure 4.4: Tunnel Management Finite State Transducer.

figure 4.4:

$$
\begin{aligned}
Q &= Tunnel\_State \\
\Sigma &= Network\_Monitoring\_Event \\
&\quad \cup Termination\_Event \\
\Gamma &= Tunnel\_Management\_Event \\
&\quad \cup Tunnel\_Alert\_Event \\
\delta &: Q \times \Sigma \rightarrow Q \\
\omega &: Q \times \Sigma \rightarrow \Gamma \\
q_0 &= Unestablished
\end{aligned}
\tag{4.4}
$$

The specifics of each of the states above are detailed in Section 4.6.3.

4.5.3 Buffering Model

When a new TCP flow is detected by MobiVPN, it decides whether or not to buffer this flow based on remaining buffer capacity. The goal of buffering is to allow TCP after reconnecting the VPN tunnel to resume packet transmission at the same rate before disconnection. This is useful when the new network has the same or better

characteristics than the previous one. Upon connecting to a network, a statistical model is used to predict whether or not the next visited network is going to congest when TCP resumes at the previous sending rate. We use the network's delay as the indicator for possible congestion which proved to be a valid indicator as per (Mittal *et al.*, 2015). We measure the VPN's RTT every time the mobile client joins a new network. After we collect enough samples, we decide or not to perform buffering, if enabled by user, based on the following: Using S as set that contains the VPN's RTT value of each visited network, and r_n as the RTT of the newly joined network:

$$Buffering(r_n) = \left\{ \begin{array}{ll} True, & \text{if} \frac{\Sigma_{i=1}^{|S|}[r_i < r_n]}{|S|} \geq 0.5 \\ False, & \text{if} \frac{\Sigma_{i=1}^{|S|}[r_i < r_n]}{|S|} < 0.5 \end{array} \right\} \tag{4.5}$$

If buffering is enabled, MobiVPN decides whether or not to buffer a new TCP flow based on remaining buffer capacity. If the flow is to be buffered, MobiVPN will buffers its unacknowledged in-flight packets. The number of buffered packets depends on which TCP congestion algorithm is used. In our work, we model the buffering according to TCP Tahoe, Reno and New Reno. The model can be expanded in the future to support other TCP variants.

We denote the number of these packets at this sending rate as P_{R_d}. During a SUSPEND state, MobiVPN decides the number of packets (P) to be buffered and acknowledged for a flow (f), based on the following model: If no RTO is triggered before network disconnection is detected, we only need to buffer the in-flight packets. However if an RTO has been triggered already, the sending rate is going to drop to 1, and TCP will enter the slow-start phase. The number of packets needed to be buffered and acknowledged to reach R_{ss} is:

$$P_{R_{ss}}(f) = 2^{\lceil \log_2(\frac{P_{R_d}(f)}{2}\rceil)+1} - 1 \tag{4.6}$$

After that, TCP enters the congestion avoidance phase, and to reach original sending rate R_d, MobiVPN buffers and acknowledge this amount of packets, denoted

as ($P_{R_{cad}}$):

$$P_{R_{cad}}(f) = ((P_{R_d}(f) - P_{R_{ss}}(f)) + 1) \times ((P_{R_d}(f) + P_{R_{ss}}(f))/2 \tag{4.7}$$

Using equations 4.6 and 4.7, and given r as the number of RTO occurrences before the detection of network disconnection, the number of packets to be buffered for a TCP flow by MobiVPN during the suspend state is:

$$P(f,r) = \left\{ \begin{array}{ll} P_{R_{d(f)}}, & \text{if r=0} \\ P_{R_{ss}}(f) + P_{R_{cad}}(f), & \text{if r>0} \end{array} \right\} \tag{4.8}$$

4.6 System Design

Our design was motivated by the following goals:

1. Provide applications that utilize TCP as a transport protocol with persistent TCP sockets that can survive network interruption events no matter how long these interruptions may be. This relieves applications from handling any possible errors due to the termination of their TCP sockets.

2. Protect the sending rate of these applications after recovering from the network interruption events by preventing TCP from dropping its congestion window and resuming sending in a slow-start phase.

3. Allow TCP to resume transmitting data as soon as the network connectivity is regained instead of waiting for the retransmission timeout to be triggered. The second and third goals both increase the throughput of TCP where network interruptions occur due to mobility.

4. During periods of disconnection, the mobile VPN should utilize this idle time by encrypting and compressing as much of the applications' data as it can handle so they are ready for instantaneous transmission after reconnection.

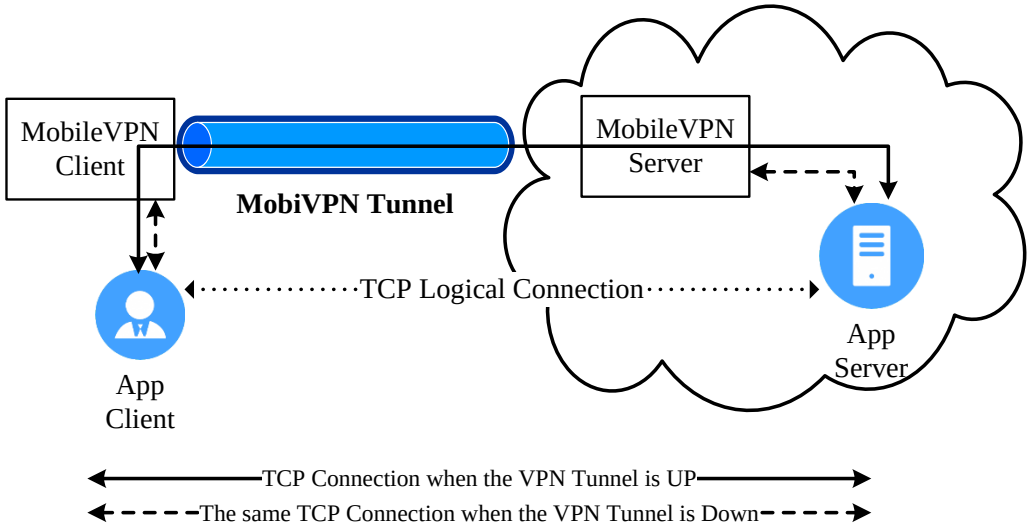

Figure 4.5: TCP Persistence Design Overview.

5. Resume the VPN tunnel as soon as the network connectivity is restored. If a new physical IP is obtained, we employ the light-weight tunnel resumption introduced in Chapter 3 to update the VPN server with the mobile node's new IP address, maintaining the same virtual IP, without the need for a full VPN handshake.

4.6.1 Design Overview

One way to achieve our first three goals is to split an application's TCP session into three TCP sessions as in (Col, 2007). However, we elected to go with what we viewed as a less complicated design in which the application clients have a direct end-to-end connection with the application servers. MobiVPN is designed to "passively observe and actively respond" only when needed. It does not interfere with the end-to-end application connection while the VPN tunnel is operating normally. This reduces the unnecessary overhead cost that would result from setting up three different TCP sessions. MobiVPN intervenes only when network connectivity is lost which results in the failure of the VPN tunnel.

In a nutshell, when a VPN tunnel fails due to network unavailability; both MobiVPN client and server enter the Suspend state in which they maintain the applications' TCP sessions. As illustrated in Figure 4.5, the MobiVPN client ensures the persistence of the applications in the mobile device by representing the application servers to TCP. At the same time, the VPN server represents the application clients preventing application servers from terminating TCP sessions while connectivity with the mobile client is unavailable. Since the VPN client and server maintain the applications' TCP sessions in a similar fashion, we will refer to the applications as either local or remote in the remainder of this chapter. We will also interchangeably refer to a TCP session as an application session.

When the mobile device is connected to a network and the VPN tunnel is healthy, MobiVPN caches the TCP packets generated by the local applications in a buffer, and removes them from it once they are acknowledged by the remote application.

Once the mobile device loses its connectivity with the network and disconnection with the remote VPN is detected, MobiVPN enters the Suspend state, and the local VPN acknowledges the buffered packets before it sends a TCP signal to the local applications on behalf of the remote applications in the form of a Zero Window message (ZWM). This pauses the local application, leading it to believe that the remote application is busy rather than being unreachable. Any packets that were already emitted by the local application before it received the ZWM, will be cached and acknowledged by the local MobiVPN. A suspended application will periodically send Zero Window probes (ZWP) to see if the remote application is still busy or not. MobiVPN responds to these probes to indicate that the remote application is still busy.

As soon as network connectivity is restored, MobiVPN enters the Resume state at which the mobile device has likely obtained a new physical IP address. In traditional

VPN connections this can be problematic, but protocols like MobileIP have been used to solve this problem. However, MobileIP-based VPNs neither guarantee the persistence of TCP sessions nor do they protect the TCP sending rate. MobiVPN allows the client to communicate it's new IP address to the VPN server using the original VPN session, providing the session's ID. This allows the mobile device to keep the same virtual IP which is used in all of the TCP sessions as the mobile device's address.

During this state, MobiVPN starts sending the buffered packets to the remote end and verifies that these packets are acknowledged before flushing them from the buffer. Replies from the remote applications will be forwarded to the local applications if they have data. Once all buffered packets of an application are acknowledged by the remote application, the local application will be resumed by forwarding the remote application's last acknowledgment to the local application with the window size field set to the remote application's receive window size.

4.6.2 System Modules

MobiVPN is composed of several interrelated modules added to OpenVPN to achieve our design goals. These modules, as illustrated in Figure 4.6 are: *a*) *Buffering* module, whose main functionality is to manage MobiVPN's buffer, decides, based on its capacity, whether MobiVPN will provide persistence with buffering or without it to a new TCP flow and caches unacknowledged packets; *b*) *Connection Monitor*, which informs the other modules of the status of the network connection; *c*) Suspention and Resumption (S&R) module, which suspends and resumes local applications; *d*) *Packet Resending* module, which sends out the buffered packets; *e*) *Verification* module, which determines what actions to take with return traffic from the remote VPN, and is responsible for flushing out the buffer; *f*) *Tunnel Management*

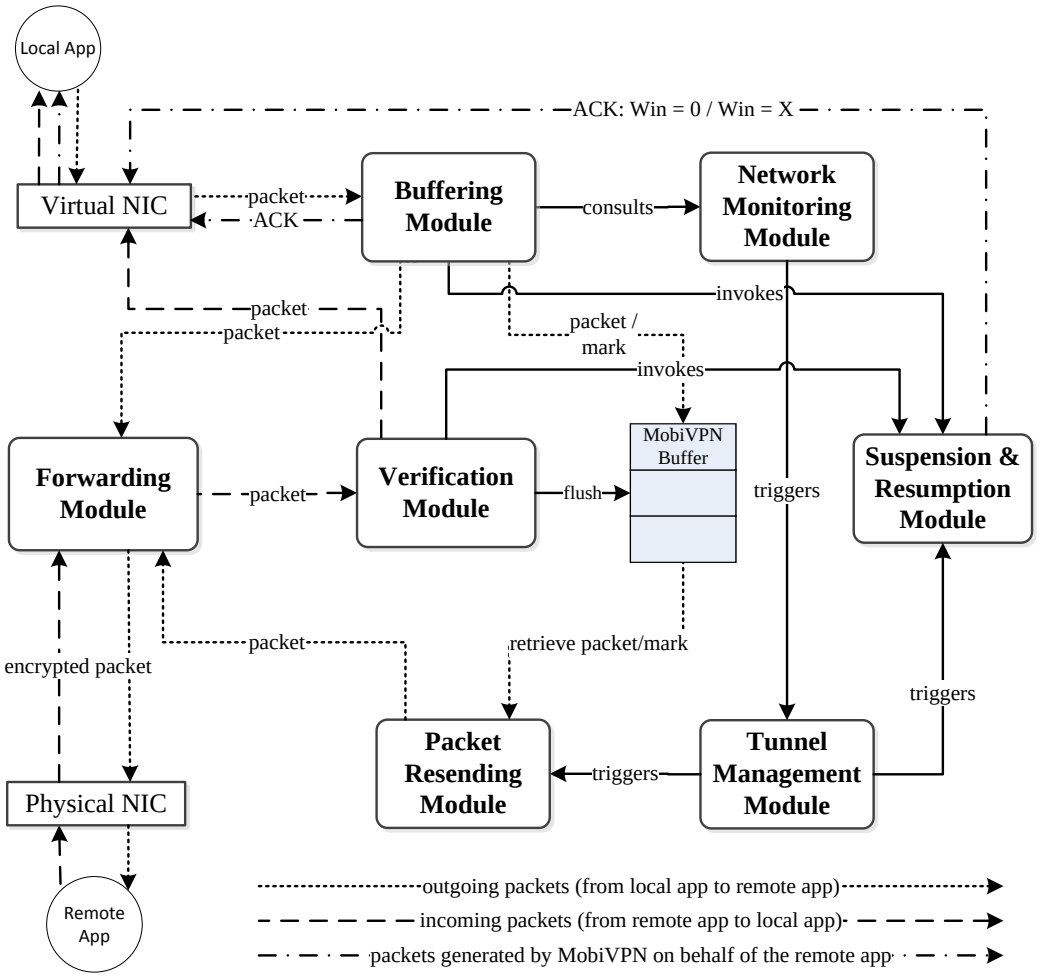

Figure 4.6: MobiVPN System Design.

module, which resumes the VPN tunnel after reconnection, or terminates it when the persistence timeout elapses. It also responsible for updating the MobiVPN states; *g*) *Forwarding* module, which is the part of the original OpenVPN that compresses, sign, encrypt and encapsulates outgoing packets as well as decrypt, verify, decompress and decapsulates incoming packets.

Buffering Module

This module is a critical component of MobiVPN. When a VPN tunnel is established, a virtual network interface card (*vNIC*) is created. Local applications' data are for-

warded by the system's TCP/IP stack as packets to the vNIC. This module intercepts every packets read from the vNIC. If a packet belongs to a new TCP flow, it adds the flow information to a table that contains one entry for each TCP flow. Figure 4.9 shows the structure of a flow profile record. Moreover, this module determines whether or not buffering will be enabled for this TCP flow based on remaining buffer capacity. The module then stores the packet in MobiVPN's buffer if the packet has data, and buffering is enabled for the flow to which the intercepted packet belongs. The packet is then handed to the forwarding module for delivery to the destination.

As the buffering module stores outgoing packets in the buffer, it immediately invokes the S&R module, during a period of disconnection, to suspend the application that sent these packets once the buffered packets reaches the flow's buffer share limit, which we have it stored in the flow profile table. For a non-buffered flow, the same module is invoked to suspend such flow as soon as a packet is intercepted from it during a disconnection period.

The module also checks the status of the VPN tunnel with the connection monitor module. If the response indicates that the VPN is down, the buffering module confirms receipt, in lieu of the remote applications, by sending the local applications acknowledgments for all packets remaining in its buffer. This is performed while ensuring the acknowledgments are sent after a delay equal to the flow's RTT stored in the flow profile table. The delay is introduced so that MobiVPN's acknowledgments do not cause TCP to underestimate the RTT when using these acknowledgments in RTT measurement.

MobiVPN measures the RTT of each buffered flow when the TCP connection is initiated, and after each tunnel resumption event. Measuring the RTT after a reconnection is essential as the mobile device may have joined a network with a different delay.

Figure 4.8 shows the buffer reference record used in MobiVPN. The 4-tuple <*Src IP, Dest IP, Src Port, Dest Port*> is used to uniquely identify to which local application the packets stored in the buffer belong. Adding the expected acknowledgment number uniquely distinguishes the packets. This number is calculated by adding the payload length to the sequence number. The sequence number field is recorded to be utilized by the verification module. A one-byte `Mark` field is set to 0 when a packet is initially stored into the buffer. For all acknowledged packets, their `Mark` is set to 1. The packet itself is appended to a buffer record following the `Mark` field. During the disconnection period, all packets that are acknowledged by the local MobiVPN are encrypted and compressed, which is done to utilize the VPN's idle time. Figure 4.7 illustrates the packet processing logic of the buffering module.

In order to determine whether buffering ore not should be enabled for a TCP flow, we use the receiver window size as an upper bound for the sending rate. If the maximum number of packets the buffer can store is buf_s and the receiver's window size for flow f_i is w_i. Then to determine if we can provide buffering for flow i, we calculate $P(f_i, 1)$ given that $P_{R_d}(f_i) = w_i/MSS$, and ensure that the following condition holds:

$$P(f_i, 1) < size_{sb} - \sum_{n=0}^{i-1} P(f_n, 1) \tag{4.9}$$

Although buffering packets requires enhancing storage resources for the MobiVPN, mathematical models for similar projects have predicted an overall improvement in communication performance (Al-Ameen and Hasan, 2008).

Suspension & Resumption Module

The S&R is responsible for suspending and resuming applications. It does so by leveraging the freeze TCP technique introduced by (Goff *et al.*, 2000). Suspending an application means pausing the TCP flow of that application. To suspend a flow,

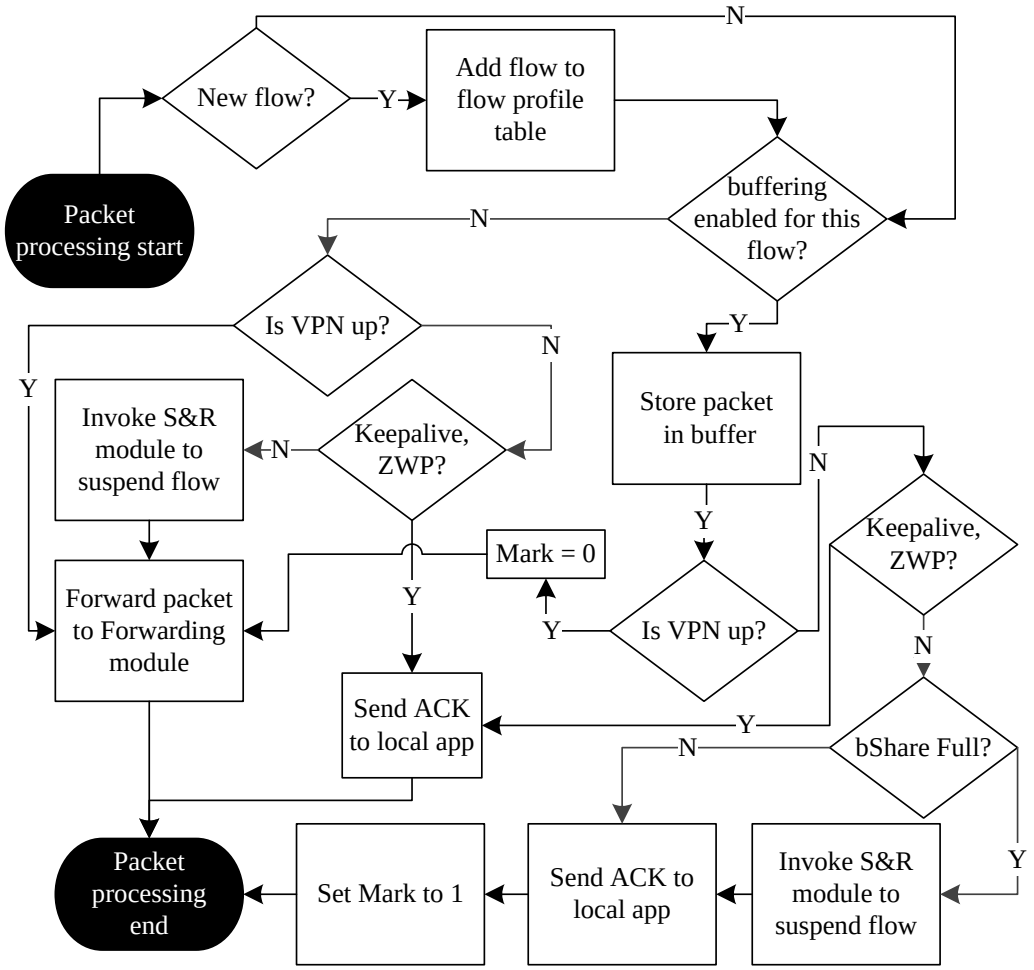

Figure 4.7: Buffering Module.

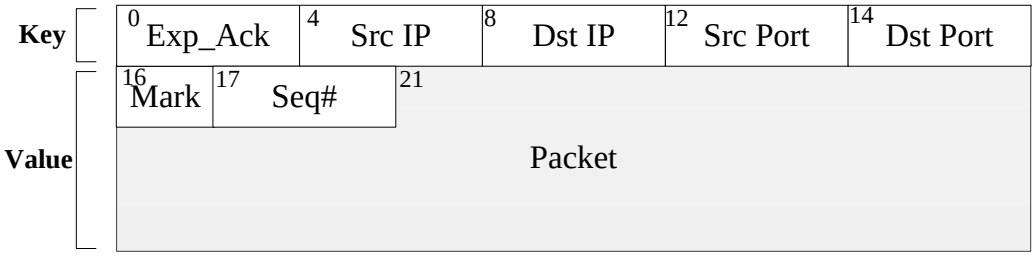

Figure 4.8: Buffer Reference Record.

the module receives a reference to a packet from the buffering module and uses it to create a zero window signal by setting the TCP window size to 0 in the TCP header. If the packet belongs to a buffered flow and has data, the module changes the acknowledgment field to confirm that the packet was received by the remote

0 src IP	4 dst IP	8 src port	10 dst port	
12 next_seq	16 last_ack	20 remote_win	22 synchronized	23 buf_enabled
27 srate	31 flight_size	35 flight_acked	39 bShare_size	43 total_acked
47 suspended	48 RTT	52 r_pak_seq	56 resume_packet	

Figure 4.9: Connection Profile Record.

application. The TCP checksum is then recalculated and the ZWM packet gets forwarded to the local application through the vNIC. This informs the application that the destination is busy and cannot handle any more data. The application then pauses sending data and will only send control packets (keepalives, ZWPs, ACKs, etc) for which the buffering module takes the responsibility of replying to such packets.

The process of resuming applications depends on whether or not the suspended flow is buffered. In the former, the S&R module receives a packet from the verification module, which belongs to a TCP flow that needs to be resumed. The module simply forwards this packet to the local application after verifying the ACK field is synchronized. This resumes the traffic immediately as TCP, in this case, will not need to wait for a retransmission timeout to be triggered since all of its in-flight packets have been acknowledged. As for the latter, the S&R module uses the resume packet stored in the flow profile table by the verification module. The module sends this packet three times to trigger fast retransmission due to the triple-ACK effect, as demonstrated in (Goff *et al.*, 2000). It is worth noting that while this action will drop the congestion window by half, it may avoid the slow start phase in case the suspension happened before an RTO was triggered, and it will prevent a possible idle time due to TCP's exponential back-off.

Connection Monitor Module

This module monitors the connectivity between the local MobiVPN and the remote MobiVPN, and reports its status to the other modules. It does so by monitoring error messages returned by the TCP/IP stack through MobiVPN's UDP socket which is used to tunnel all of the VPN traffic. It also utilizes OpenVPN's pinging mechanism to detect if the remote VPN peer is unreachable. This pinging mechanism is essential especially for the VPN server to detect that the mobile device is out of reach. The module also communicates with the OS kernel to learn any changes that could happen to the network interfaces such as IP address change or if an interface goes down etc.

Once the network is detected to be unavailable, this module makes MobiVPN enter the Suspend state. As soon as the network connectivity is restored, this module triggers the tunnel management module to resume the VPN tunnel. This module is consulted by the buffering module to determine the status of the VPN tunnel. Once the connection monitor determines the VPN tunnel is restored, it triggers the packet resending module to start sending buffered packets, as well as the S&R module to resume non-buffered flows

Tunnel Management Module

This module is responsible for creating, resuming and terminating the VPN tunnel. As our MobiVPN is meant to be a mobile VPN version of OpenVPN, we utilized the tunnel management of OpenVPN and made the appropriate changes to fit our mobile VPN goals. Our changes mainly addressed the resumption and termination of the tunnel. We closely tied the state of the tunnel to the state of the network reported by the connection monitor.

OpenVPN relies on the pinging mechanism to detect disconnections, which by default takes 60 seconds to conclude a disconnection and, thus, the first reconnection attempt always comes late when the VPN client obtains a new IP address. This is not done without merit. The OpenVPN design goal is to give a VPN client who has lost connectivity a chance to recover before the VPN session is dropped and deleted from the VPN server's memory. This is useful when the VPN client keeps the same physical IP address, in which case there is no need for a tunnel reestablishment. The tunnel, however, needs to be reestablished when the client obtains a new physical IP address. Our module responds to such event instantly and resume the VPN session without having to reesablish it in way that still links the new physical IP with the original virtual IP. This also allows MobiVPN to retain the VPN session's context which contains our buffer and flow profile table.

Our Module defines a persistence timeout option in which the user determines how long they want the VPN session and application sessions to persist. For example if the user chooses to set the persistence timeout to one hour, this means MobiVPN will keep application sessions alive for an hour, and also keep the VPN session alive for an hour.

Important to note is the fact that MobiVPN, unlike OpenVPN, will retain the data channel encryption and decryption keys of the original session to allow for the decryption of the packets in MobiVPN's buffer and the packets in any buffer in lower layers that have not yet left the mobile device yet.

Lastly, the creation or resumption of the VPN tunnel measures the RTT between the VPN client and VPN server during the exchange of packets when initializing the communication. This RTT value is later used by the packet resending module.

Packet Resending Module

Additionally, the `Mark` is incremented by 1 every time the buffered packet is sent out by the packet resending module

This module is responsible for sending the buffered packets to the remote applications which MobiVPN has, during a disconnection, acknowledged on their behalf. This occurs when the VPN tunnel is back online, as determined by the tunnel management module. MobiVPN retransmits the buffered packets if not acknowledged by the remote application for a configurable number of times.

The retransmission of buffered packets is performed in a TCP-like fashion. At first, we attempt to retransmit the entire window of packets and increment their mark by one. We calculate a retransmission timeout for a packet p from flow f based on the new VPN tunnel RTT as the following:

$$RTO_p = (f.RTT - VPN.oldRTT + VPN.newRTT) \times (2 \times (p.mark - 1)) \quad (4.10)$$

Doing so prevents the underestimation or overestimation of retransmission timeouts in case the mobile device joins a different network with a different delay.

When a timeout occurs, we drop the sending rate by half, and increase it linearly upon successful delivery of packets. Notice that TCP would have dropped the congestion window to one because of the disconnection, regardless of what the new network characteristics are. Figure 4.10 illustrates the how this module operates.

Verification Module

The verification module handles the incoming packets from the remote end. If a received packet belongs to a non-buffered flow, the module saves a copy of this packet if its `ACK` field has a higher value than the flow's highest seen `ACK` to be used as a resume

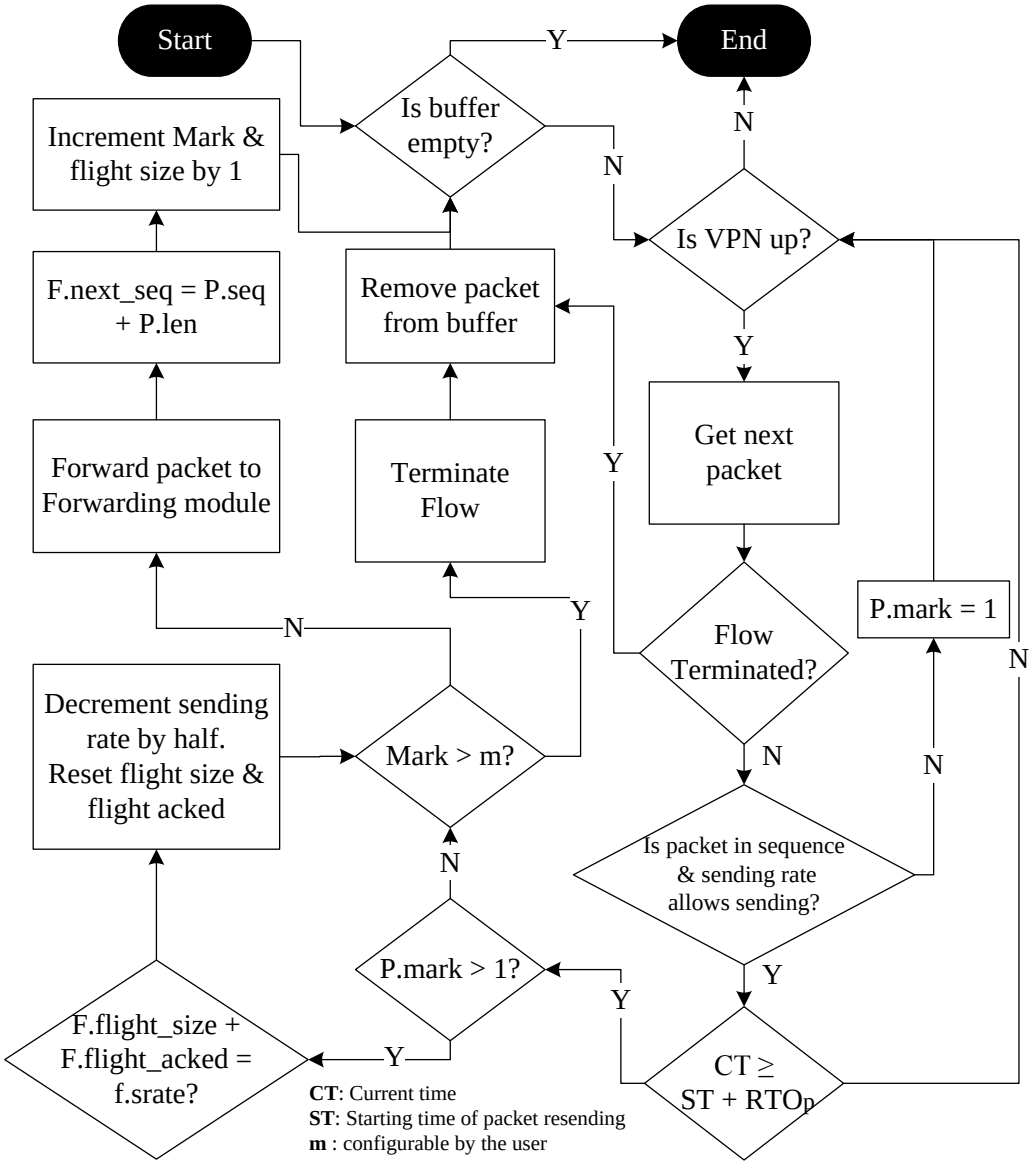

Figure 4.10: Packet Resending Module.

packet. After that, it forwards this incoming packet to the Forwarding module. If it belongs to a buffered flow, the module verifies if it is an acknowledgment of packet(s) that had been cached, by comparing the `ACK` field of this packet with the `Exp_ACK` of the packets in the buffer. If so, the acknowledged packets are flushed from the buffer and the flow profile table is updated. These acknowledgments are then forwarded to the local application through the forwarding module. During the RESUME state, the

packet is inspected for any piggybacked data. If the packet has any data present, it is forwarded to the local application with the window size changed to zero and with the ACK modified to match the highest `ACK` seen by the application. The acknowledged packets are then flushed from the cache.

Forwarding Module

This module contains the core VPN functionality which is provided by OpenVPN. It implements a Transportation Layer Security (TLS) based VPN tunnel between the mobile node and the VPN server. The TLS session is then used to derive session keys to encrypt and message-authenticate all packets going through the data channel. It receives packets from the buffering module, formulates the packet by compressing, encrypting, signing and encapsulating before sending the packet to the remote end through the physical NIC. The only modification made to this module is added logic that directs packets received from the remote end to the verification module instead of directing them to the vNIC. The buffering module also uses this module to compress and encrypt packets in its buffer during the SUSPEND state.

4.6.3 System Workflow

In a MobiVPN setup, the TCP connection between the application client and the application server is as shown in Figure 4.11. There are three operating states and one terminating state. The operating states are: Normal, Suspend and Resume. When the VPN tunnel is up, the VPN is in a Normal state and the TCP connection is end-to-end.

When the connection between the MobiVPN client and the MobiVPN server is down, MobiVPN enters the Suspend state where the TCP connection is virtually split in half, with one half between the application client and the MobiVPN client

and the other half between the MobiVPN server and the application server. The MobiVPN client acts on behalf of the application server, while the MobiVPN server acts on behalf of the application client. As long as the connection between the application client and the MobiVPN client stays up, the application client can be fooled into believing that the entire end-to-end connection is up. Similarly, as long as the application server does not lose its connection to the MobiVPN server, it can be made to believe that its end-to-end connection to the application client is up.

As soon as the VPN connection is back online, MobiVPN enters the Resume state in which any suspended TCP flow is resumed. This state is a transitional state. Once all applications are resumed, MobiVPN goes back to the Normal state.

A timing diagram for a sample scenario is shown in Figure 4.11. The transition between states is governed by the finite state model shown in Figure 4.3. MobiVPN's behavior in each state is described in the following paragraphs.

Normal State

MobiVPN is in the Normal state when the application client sends data to the application server while the VPN tunnel is up and operational.

When a data packet from a local application reaches the local MobiVPN, the buffering module intercepts the packet, and buffers it if it belongs to a buffered flow. The buffering module then checks with the connection monitor module to determine if the VPN link to the remote MobiVPN is up and operational. Upon receiving confirmation that the VPN is up, the buffering module forwards the packet to the Forwarding module, which proceeds to send the traffic through the tunnel to the remote MobiVPN. The remote MobiVPN then relays the traffic through to the remote application. Acknowledgments from the remote application gets to the

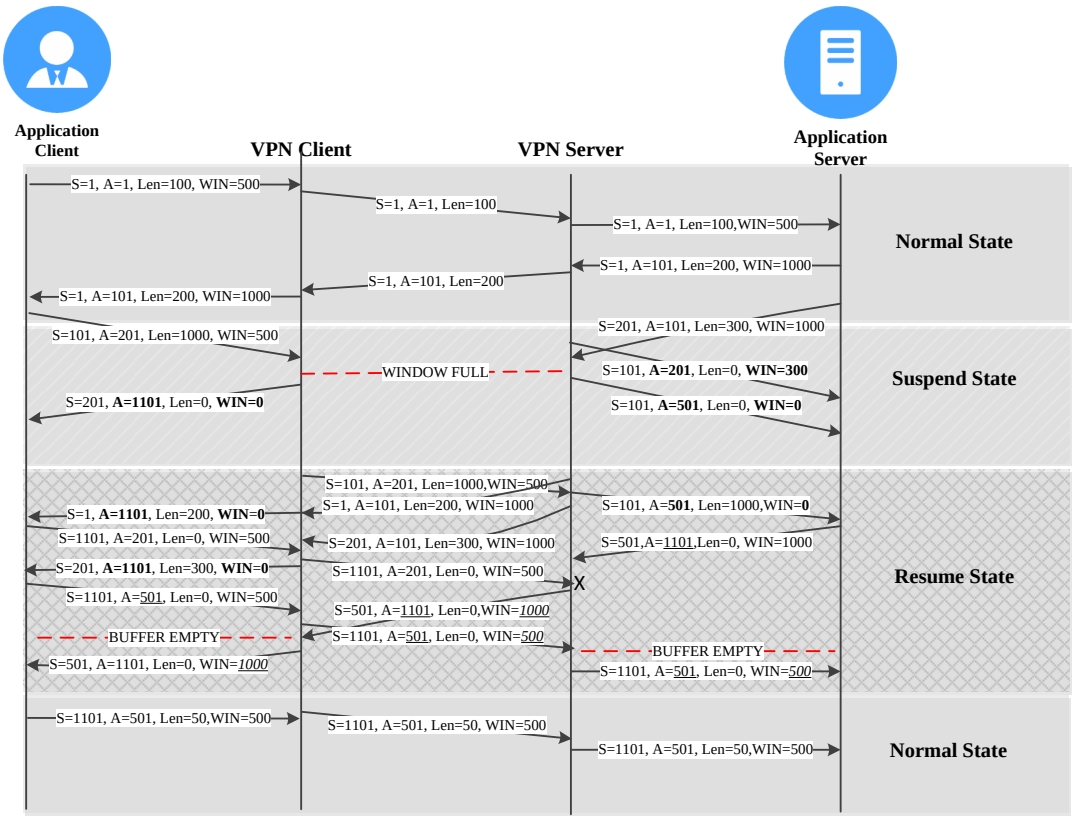

Figure 4.11: Timing Diagram for Sample Scenario. Both Application Client and Server Are Suspended and Resumed with Buffering Option.

remote MobiVPN and are returned to the Forwarding module of the local MobiVPN in a similar fashion.

The Forwarding module would forward the packet to the verification module, which would then forward the packet onto the local application after updating the flow information in the flow profile table and clearing the acknowledged packets from the buffer. Since the VPN tunnel is healthy and the application sessions are behaving normally, MobiVPN behaves as OpenVPN normally would, with the added robustness of packet buffering.

The buffer feature in MobiVPN helps mitigate scenarios where the tunnel break-down happens after a packet has been forwarded by the Forwarding module to the

remote MobiVPN. If the packet is not acknowledged, the local MobiVPN can retransmit it.

The data channel encryption and authentication keys are stored in the MobiVPN session context.

Suspend State

MobiVPN is in the Suspend state when the local application sends data to the remote application, but the connection monitor module detects that the VPN tunnel is broken. If the packet belongs to a non-buffered flow, MobiVPN immediately suspends the application by sending an acknowledgment with the window set to 0 and the `ACK` field set to the last seen ACK from this flow as stored in the flow profile table. If the packet belongs to a buffered flow, the packet will be cached, acknowledged, encrypted and compressed. When the number of buffered packets from a certain flow reaches the maximum we allow to buffer according to Equation 4.8, the flow is suspended. This ensures that the sending rate of the flow is maintained or recovered depending whether or not an RTO has been triggered during the Suspend state.

The `Mark` field of these packets is set to 1. Since the buffering module knows that the VPN tunnel is down, it does not forward the packet to the Forwarding module. In-flight packets from non-buffered flows are forwarded to the Forwarding module which may store them in its send buffer or drop them if the buffer is full.

While in the suspend state, traffic between the application server and the application client has halted, but the TCP session between them is still active. All window probe messages and/or keep-alive messages sent from a suspended TCP flow are answered in this state by the buffering module in order to prevent the termination of the TCP session.

Resume State

When the connection monitor module detects that the network connectivity is restored, the VPN tunnel is resumed by the tunnel management module. This allows the use of the original encryption keys , which allows the remote MobiVPN to decrypt the buffered packet. This starts the resume phase.

At this point, the tunnel management module triggers both the packet resending module and the S&R module. The S&R module goes through the flow profile table, and sends each resume packet of the non-buffered flows three times to trigger a fast retransmission. The packet resending module retrieves packets from the buffer, and forwards them to the Forwarding module, until the buffer is emptied from marked packets. The forwarding module bypasses the compression and encryption for these packets.

The acknowledgments received by the forwarding module from the remote MobiVPN are forwarded to the verification module, which determines whether to discard it if it had no data and was not acknowledging the packet with the highest sequence number in the buffer for the corresponding application, or forward it to the local application with the TCP window set to 0 if there were data piggybacked. A received packet from the remote end is forwarded as-is to the local application if the `ACK` field acknowledges the buffered packet with the highest sequence number of that application.

Forwarding the packet as-is means the remote window will be whatever the remote application was advertising, which means that traffic exchange may resume. All acknowledged cached packets are flushed from the buffer. When the buffer is empty from marked packets, MobiVPN enters the normal operation state.

If the verification module does not receive acknowledgments for packets that were sent out in the suspend state, the buffered packets will be retransmitted by the packet resending module. After a predetermined number of unsuccessful retransmissions, the packet resending module would send a TCP RST to the application client to terminate the suspended flow as this indicates the remote application is unreachable due not to the VPN tunnel being broken but, more likely, because the connection between the remote VPN and the remote application is disconnected.

4.7 Implementation

Our implementation of MobiVPN is based on the implementation of OpenVPN 2.2.2. We implemented our modules after analyzing and understanding how the source code of OpenVPN works. Before discussing our implementation, some of the fundamentals in OpenVPN implementation are described.

OpenVPN has four major functions that deals with incoming and outing packets. Packets read from the vNIC are processed by *process_incoming_tun*. This function basically processes packets coming from local applications and compresses and encrypts them. After that, the packet is written to the *to_link* buffer which gets picked up by *process_outgoing_link*, which writes the packet in the link's socket that is established with the remote VPN. The packet is then delivered to the IP layer which sends it to the remote end.

Incoming packets that are delivered by the TCP/IP stack to the VPN's socket are processed by *process_incoming_link*. This function decrypts and decompresses incoming packets and writes them in the *to_tun* buffer. At this point, *process_outgoing_tun* is executed to pick up the packet, and write it to the vNIC so the TCP/IP stack can deliver it to the local application.

MobiVPN modules are implemented to perform their functions with respect to the workflow described above. Below, more details are provided about how each module is implemented in relation to the original OpenVPN implementation.

4.7.1 Packet Caching Module

This module is called inside the *process_incoming_tun* function to buffer the input packet, which is placed in the outgoing buffer of OpenVPN's internal multiplexer (i.e., packets coming from mobile applications). These packets are cached in a hash table, named $mbuf$, whose structure is shown in Figure 4.8. We maintained another hash table, named f_tbl, to store information about every TCP flow using the structure shown in Figure 4.9. The buffering is performed according to Algorithm 2.

In lines 3-13, we create a flow profile if there is not one, and decides whether or not to buffer this flow. Lines 14-18 updates the state of the TCP flow according to (Postel, 1981), and cleanup the buffer and the flow table from this flow's entries if its TCP state is CLOSED. Lines 19-25 deals with the packet if it is from a non-buffered flow.

Line 22 calls the method *send_to_link*, which writes the packet into the *to_link* buffer and calls *process_outgoing_link* to deliver out the packet. Line 22 invokes the S&R module to suspend the flow if the VPN in a SUSPEND state.

Lines 26-59 treat the packet as it belongs to a buffered flow. It adds the packet to the MobiVPN buffer in line 31 before sending it to the forwarding module if the VPN is up. The returned value $f.r\_pak\_seq$ will be either 0 if this is a new packet or the packet's *seq#* if it is a retransmitted packet. This will assist in calculate *bShare_size* in Algorithm 3.

Lines 38-59 decide what is to be done during the SUSPEND state. A packet is either acknowledged with the advertised window updated or the flow is suspended.

Algorithm 2 Packet Buffering

```
1: procedure CACHE_PACKET(c : context, p : packet)
2:     f ← LOOKUP_FLOW_PROFILE(c.f_tbl, p)
3:     if f = NULL then
4:         f ← CREATE_FLOW_PROFILE(p)
5:         bShare ← CALCP(f, 1)
6:         if c.mbuf.rsize ≥ bShare and c.buffering then
7:             f.buffered ← true;  c.mbuf.rsize -= bShare
8:         else
9:             f.buffered ← false
10:        end if
11:        ADD_FLOW_PROFILE(f, f_tbl)
12:    end if
13:    flow_state ← UPDATE_FLOW_STATE(f, p)
14:    if flow_state = CLOSED then
15:        REMOVE_FLOW(c.mbuf, c.f_tbl, f); return                ▷ ends the algorithm
16:    end if
17:    if ¬f.buffered then
18:        SEND_TO_LINK(c, p)
19:        if c.vpn_state = SUSPEND then
20:            SUSPEND_FLOW(c, p, null, f)
21:        end if
22:        return
23:    end if
24:    be.src_ip ← p.src_ip                                       ▷ be is a Buffer_Entry
25:    be.dst_ip ← p.dst_ip; be.src_port ← p.src_port
26:    be.dst_port ← p.dst_port; be.seq ← p.seq
27:    be.exp_ack ← GET_PAYLOAD_LEN(p) +p.seq;  be.mark ← 0 ;  be.packet ← p
28:    f.r_pak_seq ← BUFFER_ADD(c.mbuf, be)
29:    if f.r_pak_seq = 0 then
30:        f.srate + +
31:    end if
32:    if c.vpn_state = NORMAL or (f.is_sync and c.vpn_state = RESUME) then
33:        SEND_TO_LINK(c, p)
34:    else
35:        if c.vpn_state = SUSPEND then
36:            t ← current_time() + f.RTT;    rem_packets ← f.bShare_size −
    f.total_acked
37:            if rem_packets > f.srate then
38:                w ← f.remote_win
39:            else
40:                w ← rem_packets ∗ MSS
41:            end if
42:            if w ≤ 0 then
43:                SUSPEND_FLOW(c, p, be, f)
44:            else
```

Algorithm 2: Packet Buffering (Continued)

```
45:                 ADD_ACK_QUEUE(&be, w, t);  f.total_acked++
46:             end if
47:             if f.last_ack < p.exp_ack then
48:                 f.last_ack ← p.exp_ack
49:             end if
50:             if f.last_seq < p.seq then
51:                 f.last_seq ← p.seq
52:             end if
53:         end if
54:     end if
55: end procedure
```

As soon as MobiVPN enters the Suspend state, it starts by acknowledging all buffered packets as shown in Algorithm 3.

We go through the *ack_queue* and send an acknowledgment for every entry only if the current time is larger or equal to t. An `ACK` packet is constructed such to make it appear that it is originating from the remote end by swapping IP addresses, ports and SEQ/ACK numbers. Once an application is suspended, this module responds to window probe messages as well as keep-alive messages in order to prevent TCP from timing out.

4.7.2 Suspension & Resumption Module

This module implements two main methods; one suspends TCP flows and the other resumes them. The suspension method sends a ZWM as shown in Algorithm 4, which also updates the connection profile to reflect that the flow is suspended. The method called in line 11 writes the packet into the *to_tun* buffer and calls *process_outgoing_tun* which sends the packet to a local application. Algorithm 5 is used to resume a suspended flow.

Algorithm 3 Suspension Phase Kickoff

```
1: procedure START_SUSPENSION_PHASE(c : context)
2:     for all f ∈ f.f_tbl do
3:         if f.buffered then
4:             f.flight_size ← 0; flight_acked = 0
5:             f.total_acked ← 0; f.synchronized ← false
6:             if f.r_pak_seq ≠ 0 then
7:                 f.bShare_size ← CALCP(f, 1)
8:             else
9:                 f.bShare_size ← CALCP(f, 0)
10:            end if
11:        end if
12:    end for
13:    for all be ∈ f.mbuf do
14:        f ← LOOKUP_FLOW_PROFILE(c.f_tbl, be)
15:        t ← current_time() + f.RTT
16:        rem_packets ← f.bShare_size − f.total_acked
17:        if rem_packets > f.srate then
18:            w ← f.remote_win
19:        else
20:            w ← rem_packets ∗ MSS
21:        end if
22:        if w ≤ 0 then
23:            SUSPEND_FLOW(c, p, be, f)
24:        else
25:            ADD_ACK_QUEUE(&be, w, t)
26:            f.total_acked + +
27:        end if
28:        if f.last_ack < p.exp_ack then
29:            f.last_ack ← p.exp_ack
30:        end if
31:    end for
32: end procedure
```

4.7.3 Connection Monitor Module

This module was implemented in the same way we described in Section 3.6.1. However, the use of *NETLINK* socket to detect network switching was only employed in the VPN client as we assumed in this chapter that the mobility or change of IP address happens at the mobile client.

Algorithm 4 Flow Suspension

```
1: procedure SUSPEND_FLOW (c : context, p : packet, be : buffer_entry, f : flow_profile)
2:     zwp ← CREATE_ZWP(p)
3:     zwp.window = 0
4:     if f.buffered then
5:         zwp.ack ← GET_PAYLOAD_LEN(p) + p.seq
6:         be.mark ← 1
7:         if zwp.ack > f.last_ack then
8:             f.last_ack ← zwp.ack
9:         end if
10:    end if
11:    SEND_THROUGH_TUN(c, zwp)
12:    f.suspended ← true
13: end procedure
```

Algorithm 5 Flow Resumption

```
1: procedure RESUME_FLOW (c : context, p : packet, f : flow_profile)
2:     if f.buffered then
3:         if p.ack ≥ f.last_ack then
4:             send_buf_tun(c, p)
5:             f.suspended ← false
6:         end if
7:     else
8:         for i ← 1, 3 do
9:             send_through_tun(c, f.resume_packet)
10:        end for
11:        f.suspended ← false
12:    end if
13: end procedure
```

4.7.4 Tunnel Management Module

This module implements Algorithm 6 which is a modified version of Algorithm 1, which is called by the network monitor module when a network event occurs. Similarly, the algorithm handles four cases. The first case (lines 2-10) is where the network is connected but not switched. Here, there is no need to resume the VPN session as the mobile device did not acquire a new physical IP. Therefore, the VPN state is changed to RESUME and the S&R module is asked to resume non-buffered flows, and the Packet Resending module is triggered. In the second case (lines 11-22), the

mobile has switched to a different network; similar reaction to the first case takes place except that we call the VPN resumption function whose details were described in Section 3.6.2.

The third and fourth cases occur when the network is disconnected. This enters the VPN into the Suspend state unless the *persistence_ timeout* has expired, at which time the VPN tunnel is terminated.

4.7.5 Packet Resending Module

This is implemented by sequentially going through MobiVPN's buffer (*mbuf*), and sending cached packets out using the Forwarding module. The module retries retransmitting the cached packets according to equation 4.10. Every time a packet is sent the Mark field is increased by 1. Once Mark >m (configured by the user) , the resending module gives up and flushes the connection from the cache, and send a RST packet to the local application. This is to prevent the resending module from going into an infinite loop. Algorithm 7 is the heart and sole of this module. Line 19 reduces the flow's (*srate*) by half when we retransmit a packet. The verification module increases the *srate* by one, once an *srate* number of packets has been acknowledged.

4.7.6 Packet Verification Module

This module is implemented to monitor the packets coming from remote applications. It observes the `ACK` field and compares it with `Buffer_Entry.exp_ack` to decide whether to remove a cached packet from the buffer or not. This module also updates the state of a TCP flow according to the received packet. If a flow's state changes to `CLOSED`, it removes the connection from *c.f_ tbl* and removes the buffered packets of that flow from *c.mbuf*.

Algorithm 6 Tunnel Manager

```
1: procedure MANAGE_VPN_TUNNEL(c : context)
2:     if ¬c.network_switched and c.network_connected then
3:         c.persis_timer ← 0; c.tunnel_state ← UP
4:         c.vpn_state = RESUME
5:         c.buffering ← TRUE
6:         for all f ∈ f.flowlbl do
7:             if ¬f.buffered then
8:                 RESUME_FLOW(c, f.resume_packet, null, f)
9:             end if
10:        end for
11:        PACKET_RESENDING(c)
12:    else if c.network_switched and c.network_connected then
13:        c.persist_timer ← 0
14:        RESUME_VPN(c)
15:        c.oldRTT = c.newRTT
16:        c.newRTT = MEASURE_RTT( )
17:        c.vpn_state = RESUME
18:        c.buffering = BUFFERING(c.newRTT)                ▷ Equation:4.5
19:        for all f ∈ f.flowlbl do
20:            if ¬f.buffered then
21:                RESUME_FLOW(c, f.resume_packet, null, f)
22:            end if
23:        end for
24:        PACKET_RESENDING(c)
25:    else if ¬c.network_connected and c.tunnel_state = UP then
26:        c.tunnel_state ← DOWN
27:        c.vpn_state ← SUSPEND
28:        START_SUSPENSION_PHASE(c)
29:    else if ¬c.network_connected and c.persist_timer < c.persist_timeout
   then
30:        c.tunnel_state ← DOWN
31:        c.vpn_state = TERMINATE
32:        TERMINATE_VPN(C)
33:    end if
34: end procedure
```

If a ZWM was sent during the disconnection, this module modifies the replies coming from the destination by setting the ACK to the highest seen ACK, `last_ACK`, and setting the window to zero. Upon receiving an ACK equal to or higher than the highest seen ACK saved in the flow profile, this module delivers this ACK packet

Algorithm 7 Resending Cached Packets

```
1: procedure PACKETS_RESENDING(c : context)
2:     init_time = current_time()
3:     while CONTAINS_MARKED_PACKETS(c.mbuf) and c.tunnel_state = UP
   do
4:         be =GET_NEXT_PACKET(c.mbuf)
5:         f ← LOOKUP_FLOW_PROFILE(c.f_tbl, be)
6:         if f = null then
7:             BUFFER_REMOVE(c.mbuf, be)
8:             Continue
9:         end if
10:        if (f.flight_size ≥ f.srate) then
11:            be.mark ← 1
12:            Continue
13:        end if
14:        t = (f.RTT + c.newRTT − c.oldRTT) × 2 × (be.mark − 1)
15:        if current_time() ≥ init_time + t then
16:            if (be.mark > 1) then
17:                x ← f.flight_size + f.flight_acked
18:                if x ≥ srate then
19:                    f.srate ← f.srate/2
20:                end if
21:            end if
22:            if (be.mark > m) then
23:                rst ← CREATE_RST(be.packet)
24:                SEND_TO_TUN(rst)
25:                REMOVE_FLOW(c.mbuf,c.f_tbl, f)
26:                BUFFER_REMOVE(c.mbuf, be)
27:                Continue
28:            end if
29:            SEND_TO_LINK(be.packet)
30:            f.next_seq ← be.exp_ack
31:            f.flight_size + +; be.mark + +
32:        end if
33:    end while
34: end procedure
```

as-is to enable the application to resume data sending. Algorithm 8 is the soul of this module.

Algorithm 8 Acknowledgments Verification

```
1: procedure VERIFY_ACK(c : context, p : packet)
2:     max_seq = 0, stop = false, f = NULL
3:     while ≠ stop do
4:         key.src_ip = p.dst_ip; key.dst_ip = p.src_ip
5:         key.src_port = p.dst_port; key.dst_port = p.src_port
6:         key.exp_ack = p.ack
7:         be ← HASH_LOOKUP(c.mbuf, key)
8:         if be ≠ NULL then
9:             if f = NULL then
10:                f ← LOOKUP_FLOW_PROFILE(c.mbuf, p)
11:            end if
12:            if c.vpn_state = RESUME then
13:                f.flight_size − −;  f.w_acked + +
14:                if f.w_acked ≥ f.srate then
15:                    f.srate + + ; f.w_acked ← 0
16:                end if
17:            else if c.vpn_state = NORMAL then
18:                f.srate − −
19:                if f.r_pak_seq = be.seq then
20:                    f.r_pak_seq ← 0
21:                end if
22:            end if
23:            key.exp_ack = be.seq
24:            BUFFER_REMOVE(c.mbuf, be)
25:        else
26:            stop = true
27:        end if
28:    end while
29:    if UPDATE_FLOW_STATE(f, p)= CLOSED then
30:        REMOVE_FLOW(c.mbuf, c.f_tbl, f)
31:        return
32:    end if
33:    if c.vpn_state = RESUME then
34:        if p.ack < f.last_ackand¬f.is_sync then
35:            p.ack = f.last_ack ;  p.window = 0
36:        else
37:            f.is_sync = true
38:            if IS_ALL_FLOWS_SYNCHRONIZED(c.f_tbl)  then
39:                c.vpn_state = NORMAL
40:            end if
41:        end if
42:    end if
43:    if f.is_sync = 0||p.data.len > 0 then
44:        SEND_TO_TUN(c, p)
45:    end if
46: end procedure
```

4.8 Evaluation

In this section, the features that MobiVPN adds to OpenVPN are evaluated. For every testing scenario we compare how MobiVPN performs as opposed to OpenVPN. The evaluated features are: 1) the persistence of TCP sessions during network disruption events, 2) the improvement of TCP throughput when TCP flows are buffered or not in out system.

4.8.1 Testbed Setup

To conduct our experiment we setup the testbed illustrated in figure 4.12.The VPN server and the application server are installed in virtual machines running on Parallels which is hosted on a Macbook. These VMs run Ubuntu 16.04 with 2GB RAM. The VM that has the VPN client and the application client is running on VMWare and hosted on a separate Macbook. This VM runs Ubuntu 12.04 with 2GB RAM.

The VPN server is always connected to two networks; a private network (10.0.100.0/24) along with the application server, and another network (192.168.100.0/24) accessible via Internet. In the local setup, the client VM and the VPN server are on the same 192 network. In the distant setup the client VM connects to the VPN server via an Internet-connected WiFi network.

4.8.2 TCP Sessions Persistence

Applications can lose their underlying TCP sessions when: 1) TCP does not receive an ACK for a data packet that has been retransmitted `tcp_retries2` times, or 2) when TCP keep-alive messages are not answered within (`tcp_keepalive_time` + [`tcp_keepalive_intvl` × `tcp_keepalive_probes`]) seconds. The TCP protocol

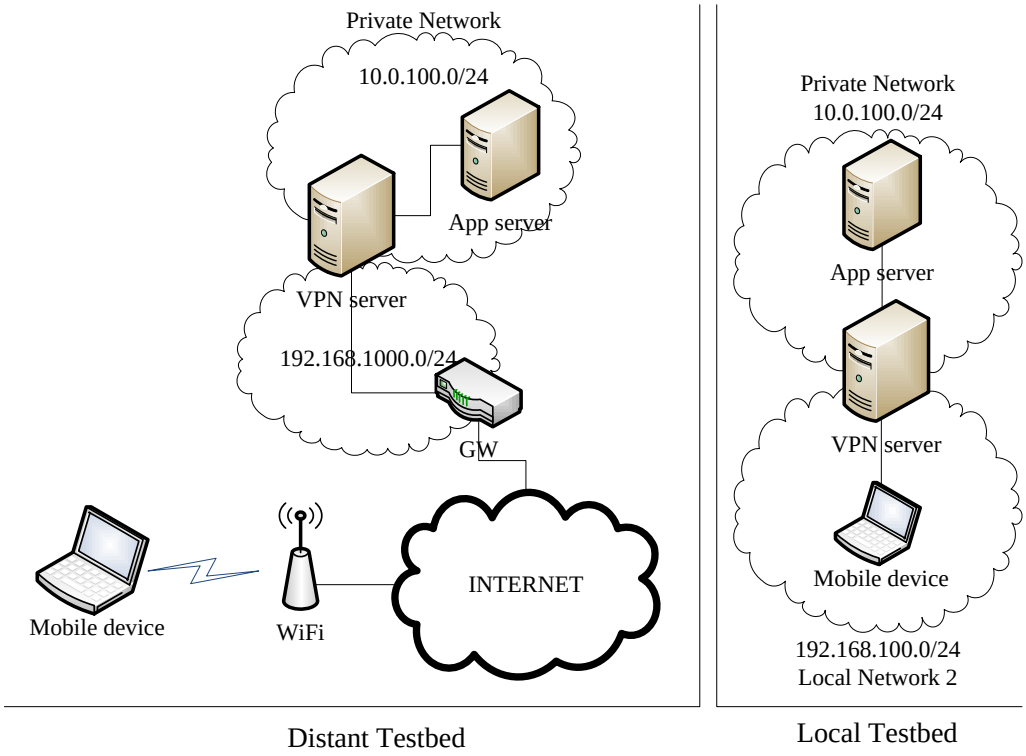

Figure 4.12: Evaluation Testbed Setup.

of the mobile device or the application server may drop the session based on their respective TCP configuration as mentioned above. Notice, even if the mobile client sets high values for these options to avoid losing the TCP session while being out of coverage, it has no control over the TCP settings of the application server, hence the TCP session persistence is not guaranteed. MobiVPN overcomes this problem.

To test this feature, we conducted an experiment over the local testbed. We wrote a basic file transfer application that sends 1 GB text file from the client to the application server and vice versa, using a TCP socket defined as *socket(AF_INET, SOCK_STREAM, 0)*, and in every scenario we alternated OpenVPN and MobiVPN. We performed four scenarios where we change the configuration of the application server's TCP settings, and disconnect the client's network interface for varying lengths as shown in Table 4.1.

Table 4.1: TCP Socket Persistence.

Transfer direction	Server's TCP settings	Disconnection length	Transfer Completed?	
			OpenVPN	MobiVPN
S → C	*tcp_retries2 = 6*	10 seconds	✓	✓
		45 seconds	✗	✓
C → S	*tcp_keepalive_time=10s* *tcp_keepalive_intvl=5s* *tcp_keepalive_probes=3*	10 seconds	✓	✓
		45 seconds	✗	✓

In all cases, using MobiVPN the file was transfered completely, whereas using OpenVPN the file transfer failed twice when the disconnection length (45 seconds) was long enough for the application server to drop the TCP session after 6 failed retransmission in one case, and after the keep-alive timeout was triggered after 25 seconds in the other case.

4.8.3 TCP Performance

We used *iperf* to measure the throughput of OpenVPN vs. MobiVPN by streaming data (100MB for distant testbed, and 200MB for local testbed) from the client to the application server using various scenarios by varying the number of disconnections, the length of a disconnection period and the testbed. In order to prevent the experiment from ending prematurely, we performed the first disconnection after 5 seconds, and then we set the length of intermediate connection periods to be 8 seconds. The disconnections are done by disabling/enabling the physical network interface using a shell script. For every measurement, we averaged out the results of five runs. In

this experiment, we do not perform any network handover eliminate the effect of late VPN resumption in OpenVPN.

With MobiVPN we performed the experiment with two cases: 1) with buffering enabled for the TCP flow (MobiVPN-B), and 2) we set the size of MobiVPN buffer to zero so that the TCP flow will be treated as a non-buffered flow (MobiVPN-NB).

Figure 4.13 shows TCP throughput obtained after carrying our experiment over the distant testbed. OpenVPN slightly outperformed MobiVPN when there are no disconnections. The degradation of the throughput is due to the overhead of monitoring the state of the TCP connection in MobiVPN. MobiVPN-NB has less overhead as there is no buffer management required. The TCP throughput gain increases with the increase of number of disconnections and the length of disconnections. For example, in the case with three 15-second long disconnections, the TCP throughput increased by 54% in MobiVPN-B, and 45% in MobiVPN-NB. TCP behavior in one of the trials of this testing case is presented in Figure 4.16. During the disconnection periods, TCP was idle in the OpenVPN case; waiting for a retransmission timeout to trigger.

In short disconnections (1 second), while OpenVPN still was performing better, TCP performance in MobiVPN was brought to a very close range. In longer disconnections, MobiVPN-B and MobiVPN-NB always yield better throughput because in both cases, retransmission timeouts are avoided, and so is TCP slow start. MobiVPN-B had the edge for two reasons. 1) it preserves the sending rate of TCP by preventing the window from dropping half its value as is the case in MobiVPN-NB. TCP in the MobiVPN-NB case, resumes in the congestion avoidance phase if the suspension happened before an RTO is triggered, otherwise, it starts in the slow-start phase. 2) MobiVPN-B utilizes the disconnection period buy encrypting and compressing the packets in its buffer so they are ready for immediate transmission upon reconnection.

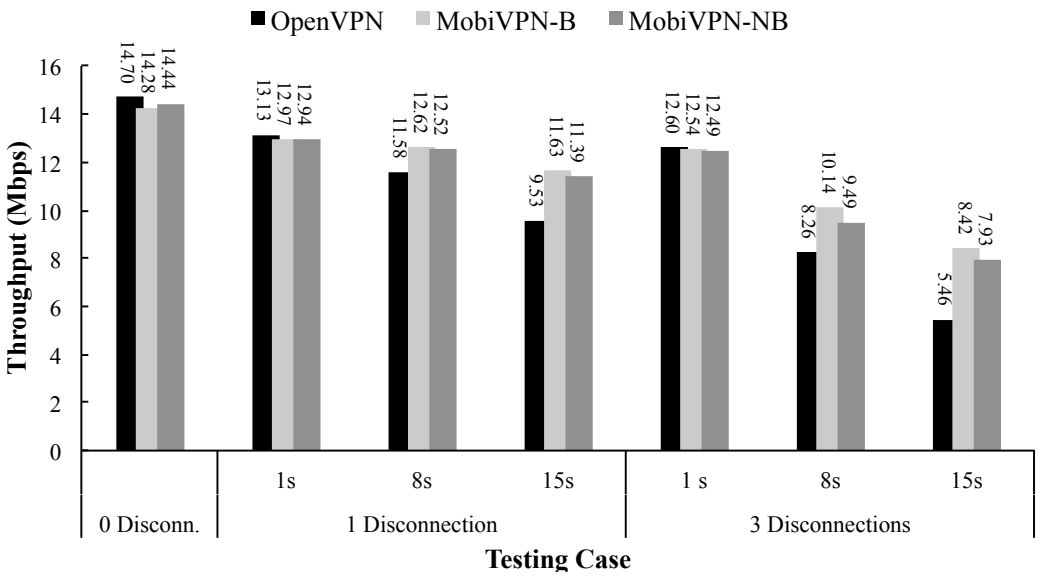

Figure 4.13: TCP Throughput Measurements in Distant Testbed: MobiVPN Vs. OpenVPN.

As shown in Figure 4.14, performing the same experiment on the local testbed showed similar observations except that MobiVPN-NB was performing almost as good as MobiVPN-B. This is due to the very short RTT in this testbed as TCP can recover its sending rate on its own in a very short time, enabling it from utilizing the connection period almost equally.

In both experiments, we aimed that all testing cases experience the same number of disconnection events. This is not always the case in reality. Therefore, we conducted our last experiment using the distant testbed. 100 MB of data were streamed with alternating connection and disconnection periods of 5 and 7 seconds long, respectively. Figure 4.15 shows three cases with different maximum number of disconnection events. In all cases, MobiVPN-B outperformed the rest while MobiVPN-NB always outperformed OpenVPN. This was very noticeable in the third scenario where the MobiVPN-B finished the data transfer while experiencing 10 disconnections. MobiVPN-NB experienced 16 disconnections, whereas OpenVPN experienced all of the 20 disconnections.

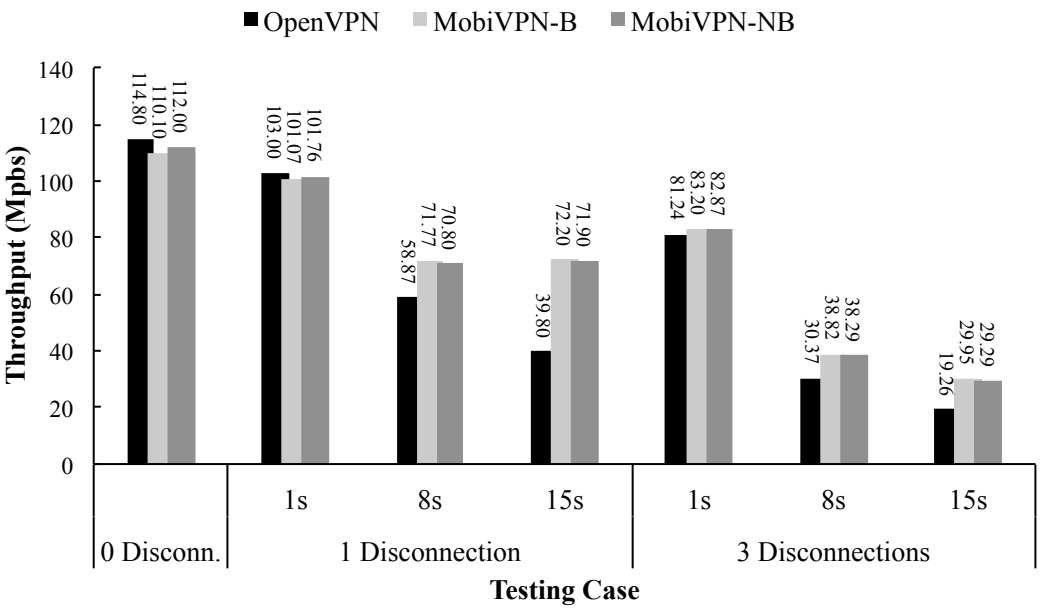

Figure 4.14: TCP Throughput Measurements in Local Testbed: MobiVPN Vs. OpenVPN.

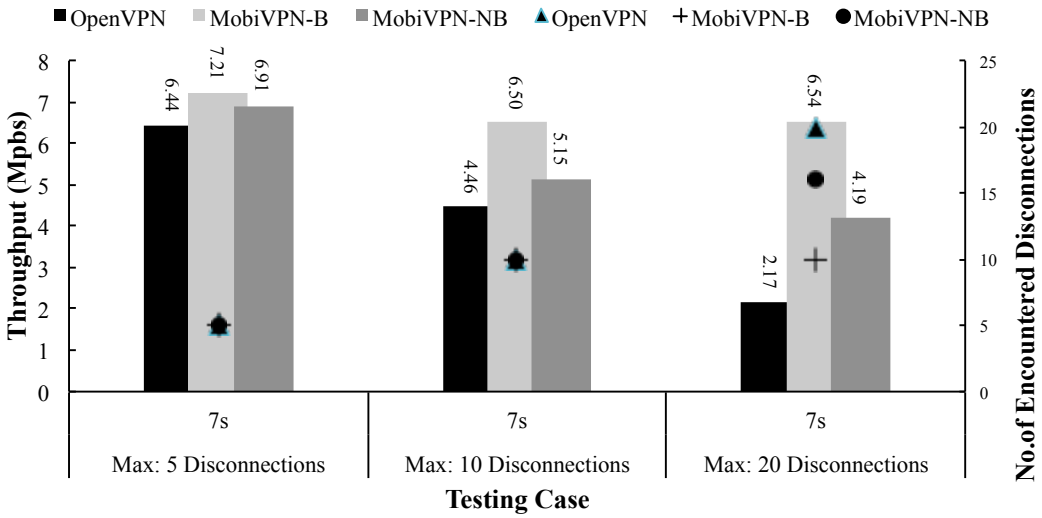

Figure 4.15: TCP Throughput Measurements in Distant Testbed with Frequent Disconnections: MobiVPN Vs. OpenVPN.

4.9 Conclusion

We have developed a robust system design for caching, TCP session resumption and packet retransmitting which hides the breakdown of the VPN connection from the application. OpenVPN source code has been modified to work in accordance with

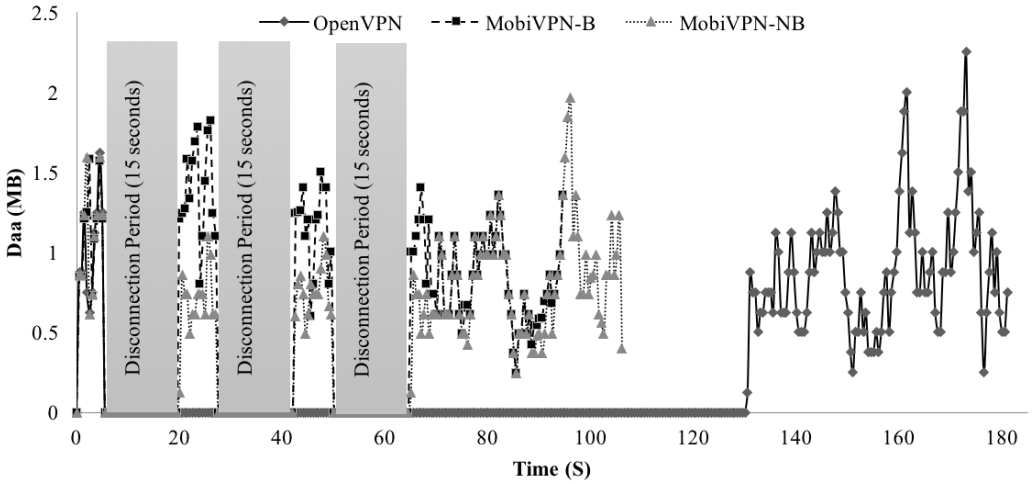

Figure 4.16: TCP Sending Rate in One of the Distant Testbed Trials, with Three 15-Second Long Disconnections : MobiVPN Vs. OpenVPN.

said design. TCP sessions tunneled through MobiVPN can be kept alive no matter how long it takes to recover the VPN tunnel.

In addition, our performance results shows the great motivation for a mobile VPN to provide persistence to TCP flows that are tunneled through it. MobiVPN solution built on our design, we believe will usher the mobility constrained VPN into the mobile age.

Chapter 5

FLOW-BASED ADAPTIVE COMPRESSION

5.1 Introduction

Compression is a technique used to reduce the size of data for several reasons, such as reducing disk space when saving files, or reducing network traffic to allow more data to be sent out, so that the effective throughput is increased. This is desired especially when a network is congested.

Some of the benefits of reducing the size of transmitted data for mobile devices include:

- It reduces data charges. Nowadays, mobile network providers charges their users based on how much data is consumed by the user. This is a motivation for mobile users to reduce their transmitted data so they do not consume their entire monthly data allowance prematurely. Although service providers may advertise for unlimited data plans, the truth is that data plans restrict the provided network speed based on a predetermined data allowance. Mobile users who consume their monthly data allowance will be switched from a high speed mobile network, such as a 4G LTE network, to a lower speed mobile network, such as a 2G network.

- Effective throughput increases when the same original data is transmitted in lesser amounts. Consider this illustrative example where the network can transmit 1 MB per second, but 2 MB of compressible data have to be sent. Without compression, it will take 2 seconds to transmit the data. If this data can be

compressed, for instance to 0.5 MB, the time to transmit the data will be 0.5 seconds in addition to the compression time, say 0.1 seconds. The effective throughput in the former case would be 1 MB/second, where in the latter case it would be 3.33 MB/second.

With all the benefits of compression, an always-compress strategy may not be ideal. For example, when we have incompressible data, the compression will be a waste of system resources, which is not desirable, especially in a mobile environment. In addition, compression may not be desirable if the CPU becomes a bottle-neck causing under-utilization of available network bandwidth. Therefore, an online adaptive compression scheme can be employed to only perform compression only when it is feasible, and abort it when it is infeasible.

In this chapter, an online flow-based adaptive compression scheme for MobiVPN, where compression will be enabled for compressible flows and disabled for incompressible flows, is introduced.

This approach will utilize the mobile device resources better, as compression of incompressible data is eliminated. It also allows the VPN to blend compressed and uncompressed packets, which allows for better network bandwidth utilization when the compression produces compressed packets at a lower rate that the network bandwidth can handle.

5.2 Related Work

(Knutsson and Björkman, 1999) introduced one of the earliest adaptive compression solutions. They altered the TCP implementation in Linux kernel by allowing TCP endpoints to negotiate their willingness to use compression during a TCP handshake. When compression is used, the adaptive mechanism decides on the level of compression by monitoring the TCP_write_queue. This queue is where the applica-

tion data is buffered before TCP segments them and moves them to the send buffer. They used zlib as a compression algorithm with different levels in which level 0 is no compression and level 11 is the highest level, with more compression ratio and requiring more CPU processing. When the length of the TCP_write_queue increases, this indicates that the network is being the bottleneck and, thus, the compression level is increased. When it shrinks, this means the network can handle more than what the compressor is producing and, therefore, the compression level is decreased to allow for more TCP segments to go to the send buffer. (Jeannot *et al.*, 2002) improved the work of (Knutsson and Björkman, 1999) by implementing the AC algorithm as a user-space library which can be used by applications. They used two threads, one for compressing and the other for sending, with a shared FIFO buffer. The compressing thread writes packets to the FIFO buffer, while the sending thread reads from it. The level of compression is changed based on the rate of change of the FIFO buffer size.

(Krintz and Sucu, 2006) introduced an adaptive compression scheme called ACE to switch the compression on or off to avoid the critical bottleneck of the system. ACE is implemented by modifying a Java Runtime Machine to intercept TCP/IP socket calls made by Java programs. A compression is turned on when the bandwidth is saturated to utilize more bandwidth. When the CPU becomes the bottleneck, the compression is turned off and data is sent uncompressed. Therefore, with this scheme at any given time, data are either all-compressed or non-compressed. Bandwidth and CPU statistics are obtained using the Network Weather Service. Data blocks of a size less than 32KB are sent uncompressed.

(Motgi and Mukherjee, 2001) developed a network conscious system for compressing text files that can be integrated with application servers that serve text files like HTML, email, news, and so forth. Based on threshold values for the files' sizes, server load, number of connecting clients, line speed, and bandwidth, the system will

choose whether or not to compress the text file and what compression method to use. The paper does not specify how the network module obtains the line speed and the bandwidth.

(Xu *et al.*, 2003) studied the impact, when downloading data from a proxy server over a wireless LAN by handheld devices, of data compression on reducing the battery consumption. The energy of communication while downloading uncompressed data is compared with the energy of communication while downloading compressed data in addition to the energy consumed while decompressing the data. The experiments showed that downloading uncompressed files consumed less energy. Therefore, they came up with an adaptive algorithm in which the proxy split the data into blocks, compressed them with zlib if they were larger than 3900 bytes, and then decided, based on a precalculated threshold obtained from equations they defined, either to send the compressed block or the uncompressed version of it. They assumed a known fixed bandwidth.

(Wiseman *et al.*, 2005) integrated a dynamically configurable compression scheme with ECho middle-ware for Grid computing. They relied on the middle-ware to provide network bandwidth information and observed the CPU load by monitoring the reduction in the speed of compression. The dynamic configuration was performed according to precalculated thresholds obtained via analyzing the statistics of the compression methods used in the study namely Huffman coding, Lempel-Zif, arithmetic coding, and Burrow Wheeler transformation after they had been applied on their dataset. Also included in the decision making was the result of the compression ratio of a 4KB sample of every 128KB block.

(Pu and Singaravelu, 2005) showed that the ACE scheme suffers from an oscillation behavior and, thus, proposed a mechanism to mix compressing some data, while not compressing other data simultaneously. Their goal was to compress as many blocks

as possible to fully utilize the CPU, while, at same time, other blocks were sent uncompressed in order to fully utilize the available bandwidth. The bandwidth and CPU utilization metrics were obtained through internal probing.

(Maddah and Sharafeddine, 2006) proposed an adaptive compression scheme for mobile-to-mobile communication, which is built into a file-sharing application. In order to reduce battery consumption, the mobile device decides whether or not to compress blocks of data based on wireless signal strength. When the signal is weak (under a certain threshold), blocks are sent compressed, otherwise they are sent uncompressed.

(Politopoulos *et al.*, 2008) studied the efficacy of using compression in DiMAPI, a distributed remote network monitoring system. When a client requests a network flow from a remote sensor, the system buffers the packets into a 64KB block and compress them using LZO. To address the delay problem resulting from waiting for the buffer to be filled up before compressing, they suggested the use of a dynamically calculated timer using packet inter-arrival rate and average throughput.

(Chen *et al.*, 2008) introduced a suite of algorithms to compress IP traffic at ISPs. Their idea was that, given a training set of network traces from different flows, the goal was to identify intra- and inter-packet correlations and find the best reordering of the packets (including a byte-based reordering of the fields of the packet header) to increase the compression ratio. According to their paper, finding a near-optimal compression plan requires huge computation which can be suitable for data centers. Once obtained, the compression plan can be fed to online and offline compressing algorithms. The compression ratio of these algorithms is monitored, and, if it reduces due to changes in network flow characteristics, a new compression plan can be reproduced with a new training set from the current network traffic. This work was motivated by saving storage space of Internet flows required by some governments.

Therefore, spending resources to find a near optimal compression plan made sense for this purpose, but, in my view, it is not applicable for resource-constrained mobile devices or even personal computers.

(Shimamura *et al.*, 2010) introduced an adaptive online packet compression scheme applied on network internal relay nodes rather than end hosts. When a relay node receives a packet, it compresses it only if its waiting time in the queue is going to be more than the time it requires to compress it. The essence of this scheme is that every relay node in the core network compresses a subset of the packets, so when a packet passes through multiple congested advanced relay nodes, chances of it getting compressed by one of them is increased. The effectiveness of this scheme was evaluated by simulations.

(Yoshino *et al.*, 2014) improved on the work of (Shimamura *et al.*, 2010) by using flow information (Source IP, Destination IP, source port, destination port, protocol number) to reorder packets in the queue of the relay node so as to compress packets from the same flow together. This work was also evaluated through simulation.

(Xiao *et al.*, 2010) introduced a framework for energy-aware lossless compression in mobile services. They proposed the use of a performance enhancing proxy (PEP) that can communicate with the mobile client to adaptively compress files to be sent to the mobile client if it would reduce the energy consumption by the mobile device. Upon requesting a file from the server, the mobile device sends a message to the PEP containing current values of battery level, signal-to-noise ratio, and available compression algorithms. For every file to be sent to the mobile client, the PEP computes the compression effectiveness of every compression algorithm using the information obtained from the mobile device, along with precomputed values of compression ratios of the supported algorithms on predefined file types, and precomputed energy cost for different data transfer rates. The adaptation algorithm in PEP then compresses

the file with the algorithm with highest compression effectiveness if it is higher 1 and if the battery level is above a threshold.

(Park and Park, 2011) developed an adaptive compression scheme for TLS. They did so by introducing three changes to the implementation of OpenSSL. First, they separated the computation routines from the network routines which allows them to blend compressed packets and uncompressed packets in TLS transmissions. Second, in order to address the dynamic differences in capabilities (network bandwidth, computation power) between the sender and receiver, they introduced a floating scale mechanism which calculates a computing index (CI) for each encoding scheme (compression + encryption). CI's of the sender and the receiver are then exchanged. The best encoding scheme is selected by the shortest Euclidean distance between the current network bandwidth and the encoding scheme. Third, they improved the memory management when allocating buffers during the switching of the encoding scheme.

(Hovestadt *et al.*, 2011) investigated the use of adaptive compression schemes to mitigate the negative effects of shared I/O in IaaS clouds. Motivated by their observation that CPU utilization and I/O bandwidth are inaccurately reported in Virtual Machines, they devised their AC scheme based on regularly changing the compression algorithm and observing its effect on the application data rate. Their AC module is placed between the applications and their respective I/O layer.

The adaptive compression solutions discussed in this section show the importance of adaptive compression. However, none can be applied directly to what we wanted to achieve in our MobiVPN. Most of the schemes handle data input coming from a single application as they are located between the application and the transport layer, either through a middle-ware e.g., (Krintz and Sucu, 2006), a library e.g., (Jeannot *et al.*, 2002), modifying a transport protocol e.g., (Knutsson and Björkman, 1999) or

modifying a session protocol e.g., (Park and Park, 2011). Therefore, the heterogeneity of the input data is reduced and data can also be compressed as blocks.

In the VPN scenario, it intercepts every packet from the virtual interface and compress them individually and not in bulks. The reason for that is: 1) a packet may not have any relation to the next packet in line since it may belong to a different flow and, thus, have a different application, 2) packets compressed together as a block, may be at risk if the results of the compression have to be sent out to the other end fragmented. The receiver may not be able to decompress it if one of the fragments did not arrive.

(Shimamura *et al.*, 2010) and,(Yoshino *et al.*, 2014) addressed adaptive compression for IP packet-based systems but assumed the existence of multiple relay nodes that performs the compression. This benefits the network but does not reduce the size of packets transmitted by the mobile device.

The use of adaptive compression for mobile devices has been looked at from an energy-saving prospective such as in the works of (Maddah and Sharafeddine, 2006), (Xiao *et al.*, 2010), and (Xu *et al.*, 2003). While this is an important factor, our adaptive compression is focused on improving the effective throughput with data reduction with the least number of compression operations. The energy-consumption factor is left for future work..

5.3 Background

In this section, the compression options that OpenVPN provides are discussed, and details of OpenVPN's adaptive compression scheme are provided.

OpenVPN provides three options for traffic compression: 1) no compression, 2) always-on compression and 3) adaptive compression. For compression, it uses LZO, developed by (Oberhumer, 2008), which is applied on each IP packet individually.

LZO is a lossless data compression algorithm that is designed to provide very high speed of compression and decompression at the expense of lower compression ratio. However, according to (Oberhumer, 2008), it achieves a quite competitive compression quality compared to other lower speed compression algorithms. This made LZO is an ideal option for compression in OpenVPN, as it has the goal to disseminate packets as quickly as possible.

It is obvious that a no-compression option introduces zero computation overhead, but leaves out the opportunity for compressing the compressible data packets. An always-on compression guarantees that compressible data will be compressed. However, it introduces unnecessary computation overhead to compress incompressible data packets.

Incompressible data packets when compressed with LZO, introduces an increase of packet length by up to 10 bytes. In that case, OpenVPN chooses to ignore the result of compression and send out the uncompressed packet. This, however, takes place after the compression has already been performed. For this reason, OpenVPN introduces the adaptive compression option which is described in the following section.

Finally, OpenVPN applies a minimum packet length policy to perform compression. Packets that are less than 100 bytes are not compressed, regardless of whether the always-compress strategy or the adaptive compression are used.

5.3.1 Adaptive Compression in OpenVPN

OpenVPN documentation provides little information on how its adaptive compression works. Therefore, the source code of OpenVPN 2.3.10 was analyzed to understand how the adaptive compression works.

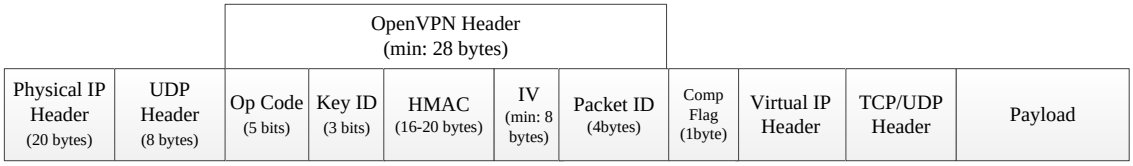

Figure 5.1: Data Packet Format in OpenVPN.

Our source code analysis showed that the adaptive compression in OpenVPN is designed with basic sampling at 2-second intervals. The flowchart of the adaptive compression is presented in Figure 5.2.

At first, compression is turned on and packets are compressed using LZO. After a packet is compressed, the size of the resulting packet is compared with the original packet. If there is a reduction in size, the compressed packet is transmitted, otherwise, the original packet is transmitted instead. A one-byte flag is then perpended to the packet to indicate whether or not it had been compressed, as shown in Figure 5.1.

After that, OpenVPN checks whether or not the packet qualifies as a sampling packet. A data packet would qualify if its original length is greater than 1000 bytes. The total size of original packets is recorded as well as the total size of transmitted packets after compression. At the conclusion of the 2-second sampling period, the tunnel compression ratio (TCR) is calculated according to the following equation:

$$TCR = (1 - \frac{Total\ size\ of\ transmitted\ packets\ after\ compression}{Total\ size\ of\ original\ packets}) \times 100 \qquad (5.1)$$

The decision whether to leave the compression turned on, or turn it off, is based on the following model that determines the tunnel compression state (TCS):

$$TCS = \left\{ \begin{array}{ll} ON, & if\ TCR \geq 5\% \\ OFF, & if\ TCR < 5\% \end{array} \right\} \qquad (5.2)$$

When the compression state is changed to OFF, it remains in that state for 60 seconds before the compression is automatically switched back on for another 2 seconds.

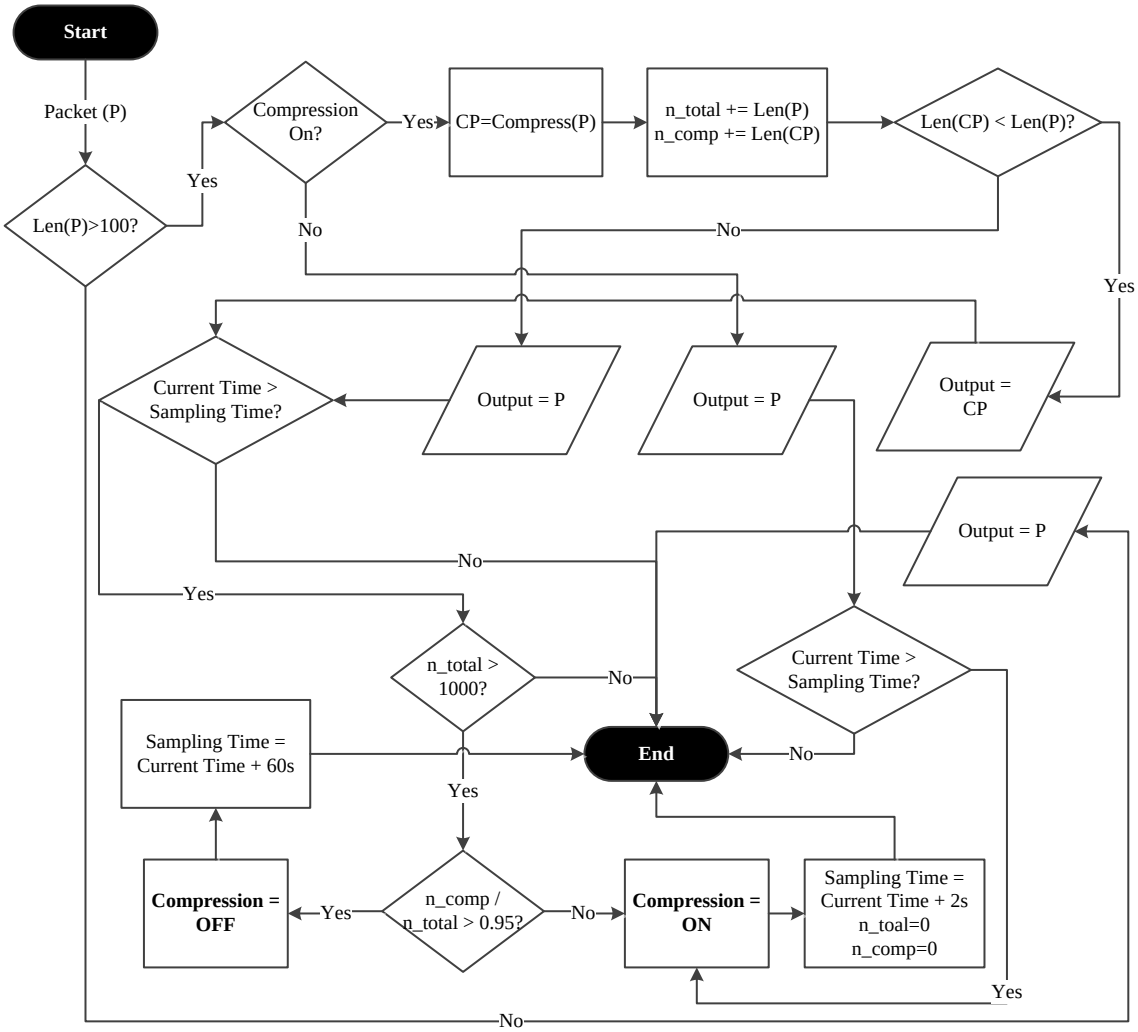

Figure 5.2: Adaptive Compression in OpenVPN.

Therefore, OpenVPN does not mix the sending of compressed and uncompressed packets. The compression decision is applied on the entire tunneled traffic except for packets that have less than 100 bytes as we discussed earlier.

5.3.2 Packet Processing in OpenVPN

In this section, how the OpenVPN processes and transmits packets will be discussed as we made some modifications to the way the packets are processed in MobiVPN. Since the goal was to perform the compression adaptively, only OpenVP-

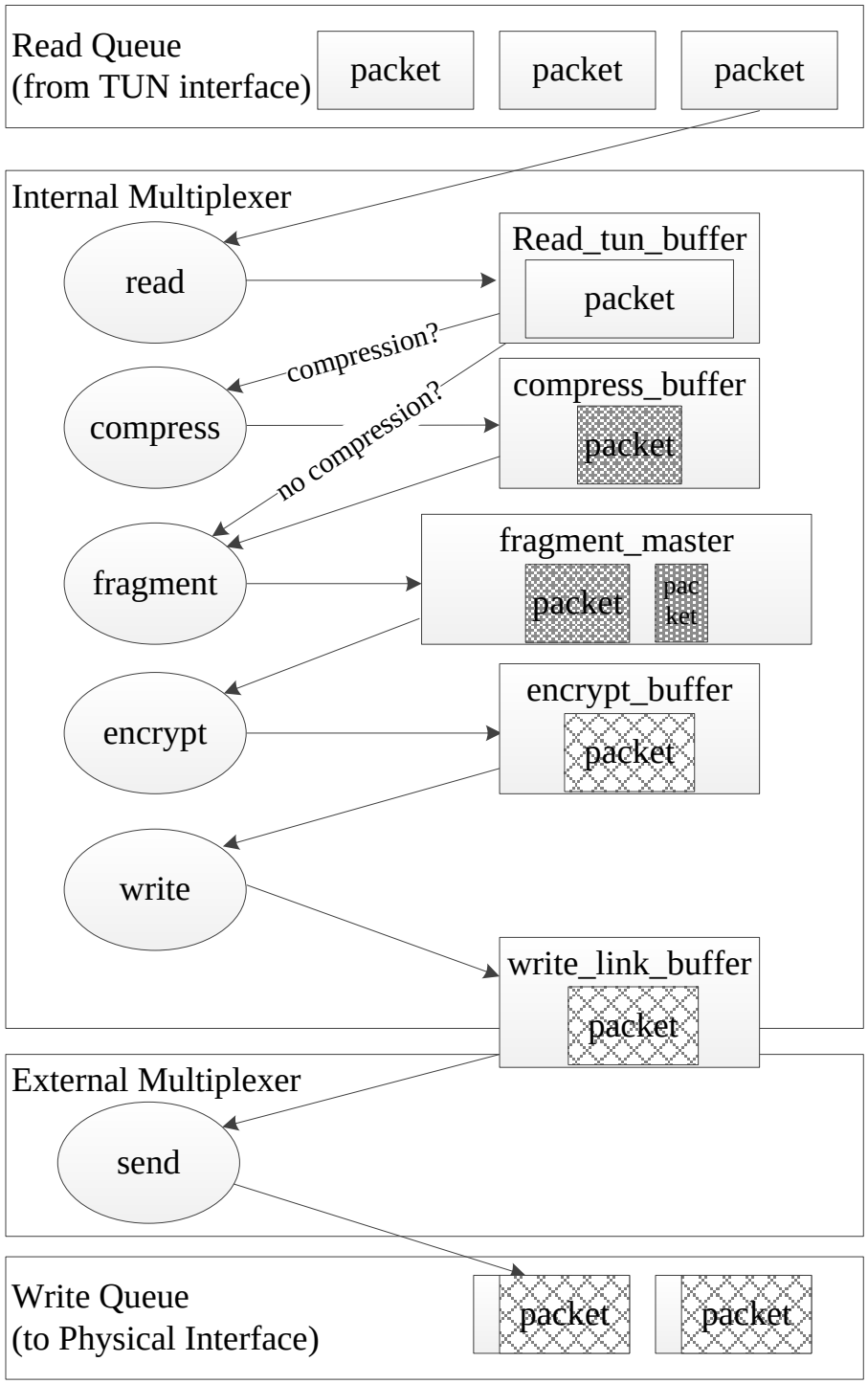

Figure 5.3: Packet Processing in OpenVPN.

NâĂŹs processing when sending data will be discussed. The process is illustrated in Figure 5.3.

OpenVPN processes packets atomically one at a time. The internal multiplexer reads a packet from the TUN interface and places in the `read_tun_buffer` which is big enough to hold one packet. It then compresses the packet, if enabled, and places the outcome in the `compress_buffer`. If the size of the packet is greater than the VPN's MTU, the fragmentation module splits the packet into smaller fragments and processes the fragments individually after adding a fragmentation header that tells the receiver how to reassemble the fragmented packet. The packet(s) are then encrypted and placed in the `encrypt_buffer`. After that, the internal multiplexer writes the packet into the `write_link_buffer` after which, it invokes the external multiplexer to send it. The external multiplexer reads the packet and sends it out through the VPN's UDP socket, which transmits the packet through the physical interface.

An important observation was that OpenVPN encapsulates each compressed packet in its own IP packet. This behavior was confirmed by inspecting captured VPN traffic through Wireshark. A highly compressible file was sent out through the tunnel, and it was observed that, while captured packets on the TUN interface were of the size of 1407 bytes, the same number of packets were sent out through the physical interface with a size equal to 183 bytes. Sending small packets increases the bandwidth overhead as each small packet will have an additional IP and UDP headers.

5.4 Motivation

Figure 5.4 shows the adaptive compression strategy of OpenVPN. The compression state alternates between ON and OFF depending on the compression ratio of a sampling period from all flows in the tunnel. The decision to whether or not enable compression is applied on the entire tunnel. Flows that can be compressed will not be compressed during the OFF state when most flows are not compressible. Similarly,

during the ON state, incompressible flows will be compressed unnecessarily which wastes computation resources.

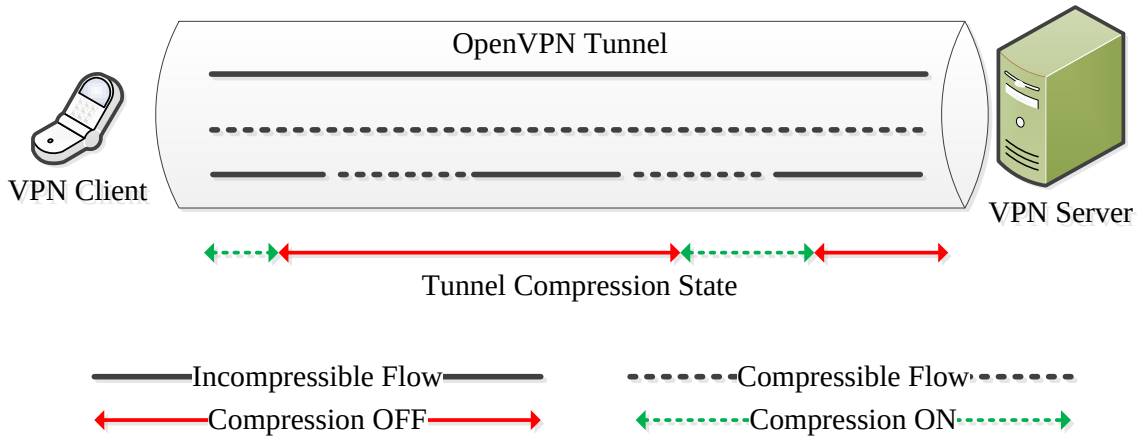

Figure 5.4: Adaptive Compression Scheme in OpenVPN.

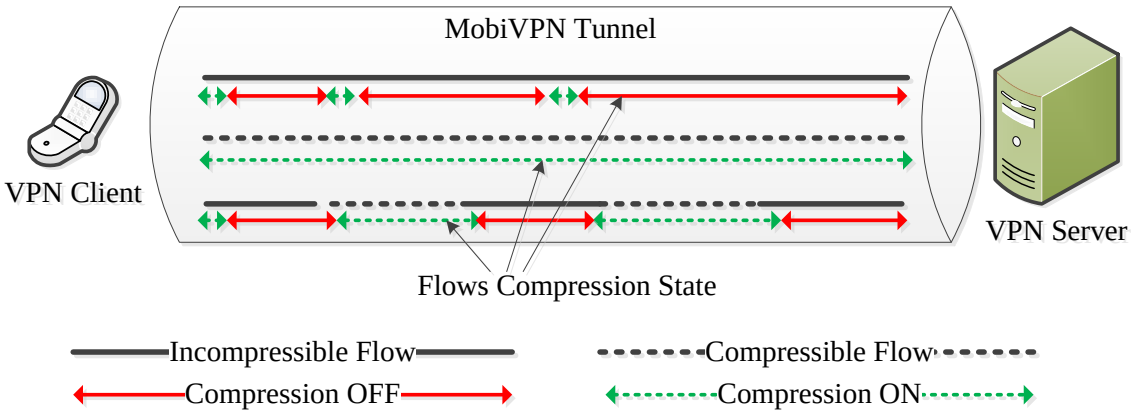

Figure 5.5: Flow-Based Adaptive Compression Scheme in MobiVPN.

The motivation in this work was to overcome the shortcomings of OpenVPN's adaptive compression. Functions that are applied in the application layer are performed on all packets from the same flow before they are intercepted by the VPN. Some application protocols will compress and/or encrypt their data before they are handed over to lower layers. For example, HTTP compresses its data when compression is enabled. Also, protocols such as HTTPS, SSH and SSL encrypt their data

before they are delivered to the VPN. A better adaptive compression strategy would be more fine-grained, and would assign a separate compression state for each flow.

Figure 5.5 illustrates how adaptive compression in MobiVPN is carried out. Three flows are illustrated in this example where the compression state for each flow is independent from the other. This would allow MobiVPN to turn off the compression on the first flow most of the time as compression has to be turned on for short sampling periods. The compression is always turned on for the second flow. The third flow resembles a flow that contains compressible and incompressible packets such as an FTP application that sends files where some are compressible and some are not. The compression for such flow will be turned on and off alternately depending on the outcome of the sampling periods.

Another advantage of doing the compression in a flow-based manner is that it gives the mobile VPN the ability to send out compressed and uncompressed packets at the same time. This is very useful to utilize the bandwidth when the CPU becomes a bottle-neck as the VPN can still send out uncompressed packets whereas if the compression was turned on for the entire tunnel, all packets would have to be compressed by the CPU before they were sent out. The bandwidth utilization in this case will be bound by the rate at which the CPU performs packet compression.

5.5 Design of Adaptive Compression in MobiVPN

MobiVPN's adaptive compression is designed with two goals in mind:

- The design of a fine-grained, flow-based adaptive compression which includes:
  - Each flow will have its own compression state, and its own sampling.

- The compression sampling periods will be much smaller than that of OpenVPN. This is because we will be sampling from one flow instead instead of from the entire tunnel.
- When turning off the compression, the OFF period will start small and duplicates every time the sampling confirms that the flow is still incompressible.

- The compressed packet will be aggregated in a buffer that can hold one MTU-long packet. This reduces the overhead resulting from sending small packets individually.

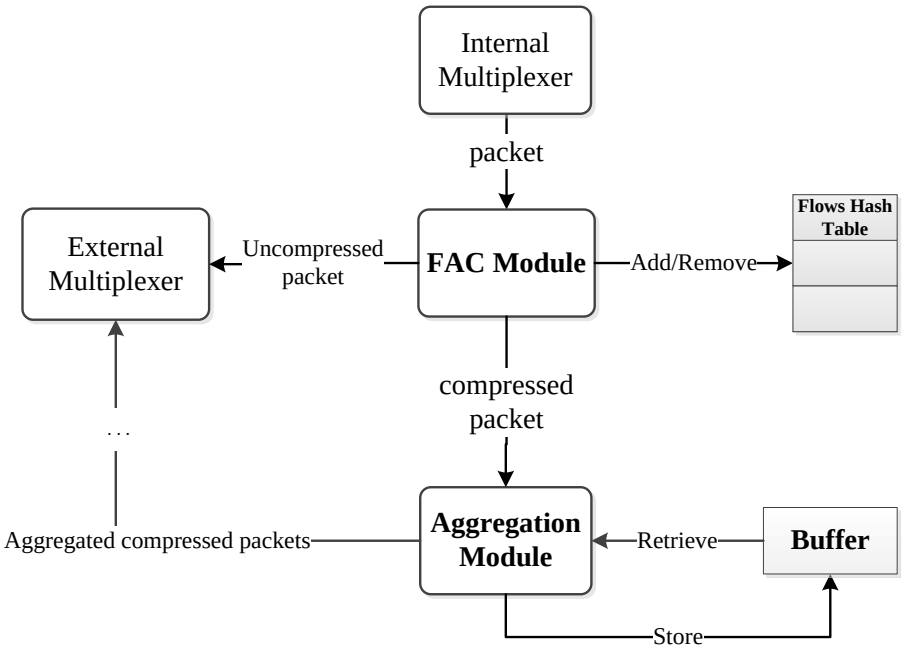

Figure 5.6: Modules Design of Flow-Based Adaptive Compression in MobiVPN.

Figure 5.6 shows the modules that were introduced and how they interact with OpenVPN's internal and external multiplexers in order to accomplish the aforementioned goals. In the following sections, the design of each module and its functionalities are described.

5.5.1 Flow-based Adaptive Compression (FAC) Module

In order to perform our flow-based adaptive compression, we introduced a hash table for the tunneled flows, which is consulted by the adaptive compression module, was introduced. The hash table contains one entry per flow. Each entry of the hash table is composed of a key and a value. The address of the flow was used as the key. The address contains: the source IP address, the destination IP address, the source port and the destination port. The value part of the hash table contains a record that hold the following fields:

- Compression state; which is a flag that indicates whether the compression is turned on or off for this flow.
- Sampling Time, which is used to determine whether or not compression should be turned on to collect a new compression sample.
- Increment; which is used to increase the sampling time of the next sample when the compression is to be turned off.
- N_total: the number of bytes from this flow during a sampling period seen by the VPN before compression is applied.
- N_comp: the number of bytes from this flow resulted after compression during a sampling period.

Table 5.1 shows an example of a flow hash table. It is important to note that before the flow hash table reaches its full capacity, one entry was added to represent any new flow. This makes MobiVPN treats these flows just like how OpenVPN does with one difference. The maximum time for an OFF state was set to 16 seconds instead of OpenVPNâĂŹs 60 seconds. This is considered a fail-safe strategy that

Table 5.1: An Example of a Populated Flow Hash Table.

Key				**Value**				
Src IP	*Dst IP*	*Src Port*	*Dst Port*	*Comp State*	*Sampling Time*	*inc*	*n_total*	*n_comp*
10.0.8.6	10.0.5.3	1900	25	ON	10:54:11:850	1	50000	4000
10.0.8.6	10.0.5.3	1901	443	OFF	10:54:12:001	4	0	0
10.0.8.6	10.0.5.15	1902	22	OFF	10:54:18:500	16	0	0
10.0.8.6	10.0.5.30	1903	23	ON	10:54:12:005	1	6000	4500
*	*	*	*	ON	10:54:14:005	16	1000	800

MobiVPN may not need to encounter since it periodically removes the idle flows and terminated TCP flows, which allows for new flows to be added. A hash table with a size of 65535 entries ensures that all flows will have an entry, as this is the maximum number for source ports.

Figure 5.7 shows the flowchart of the FAC module. When a packet is received by the FAC module, it checks whether or not it belongs to a flow that has been seen before. If it is not, a new flow entry is added to the flow table and compression is turned on for 200ms. The packet is then compressed, and compression statistics are recorded. If the packet belongs to a flow that has an entry in the flow table, it was decided whether or not to compress it based on the current compression state of the flow. Just like OpenVPN, the compressed packet is sent only if there is a reduction in size. After that, whether or not the flowâĂŹs sampling time has passed is checked. If it has passed, the flow compression ratio (FCR) is checked as follows:

$$FCR(f) = (1 - \frac{f.n\_comp}{f.n\_total}) \times 100 \tag{5.3}$$

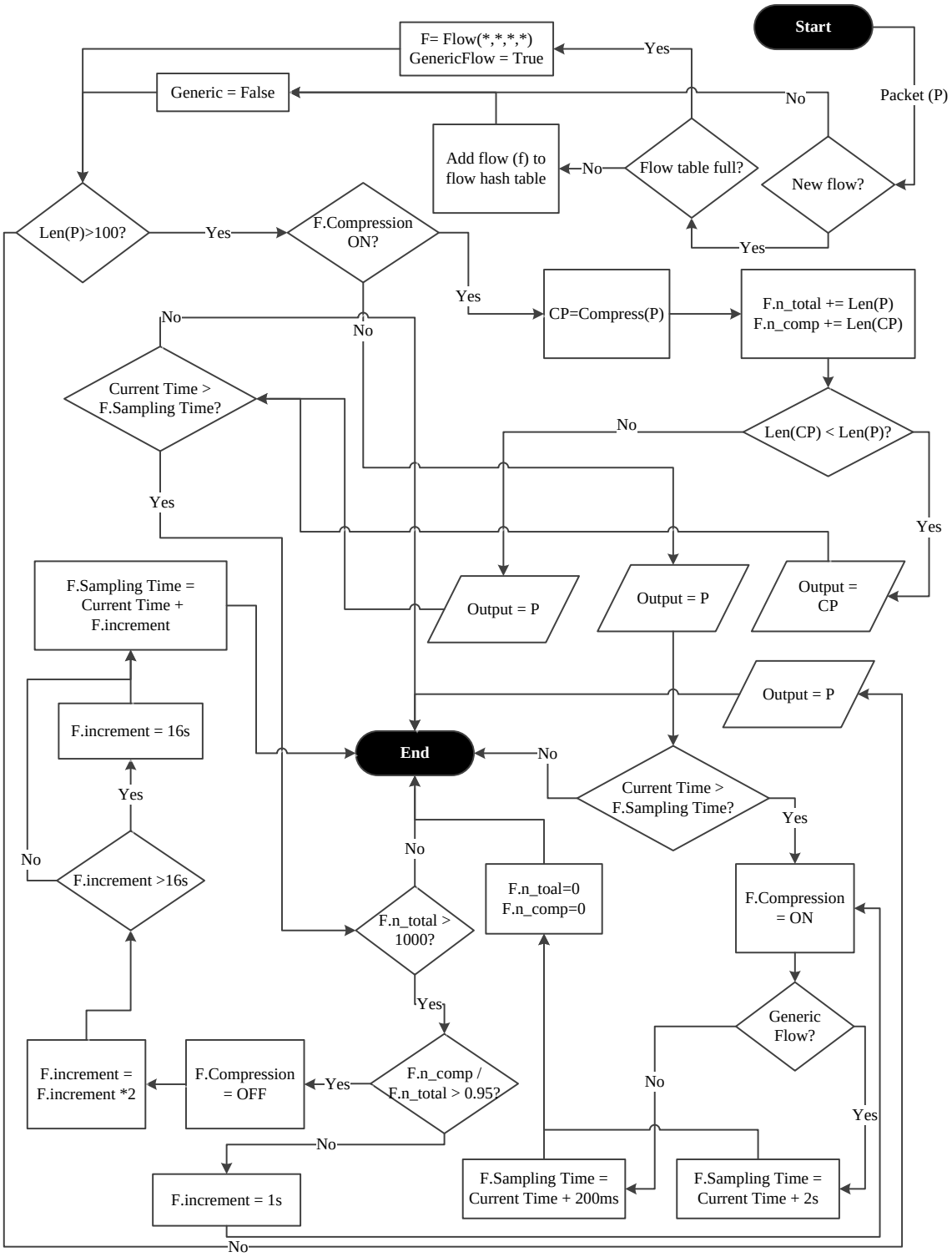

Figure 5.7: Adaptive Compression in MobiVPN.

After that, a decision, similar to that of OpenVPN, to determine the flow's compression state (FCS) can be made based on this model:

$$FCS(f) = \left\{ \begin{array}{ll} ON, & \textit{if } FCR(f) \geq 5\% \\ OFF, & \textit{if } TCR(f) < 5\% \end{array} \right\} \tag{5.4}$$

If, after a sampling period, a decision is made to turn the compression off, it will remain off for 2 seconds before another sampling period is started. If it is decided that the compression i to be turned off again, it will remain off for twice the previous OFF period, with the maximum set to 16 seconds. The choice was also made to reset the length of the OFF period to 2 seconds once the result of a sampling period indicates the compression should stay on. The reason this is done is to limit the sampling periods for incompressible flows. However, when the flow contains a mixture of compressible and incompressible packets, the long OFF periods will result in missed compression opportunities. Hence, in such cases, we reset the length of the OFF period.

5.5.2 *Compressed Packets Aggregation Module*

The goal of this module was to aggregate compressed packets and send them in one packet which reduces the overhead of sending small packets each with an external IP header, UDP header and OpenVPN header.

Figure 5.8 shows the format of an aggregated packet. Each compressed packet is perpended with a 2-byte filed that contains the length of the compressed packet before they are concatenated to form the payload of the aggregated packet.

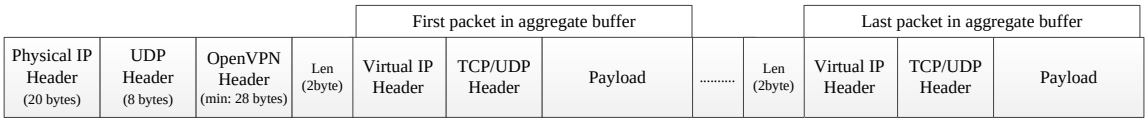

Figure 5.8: Data Packet Format in MobiVPN.

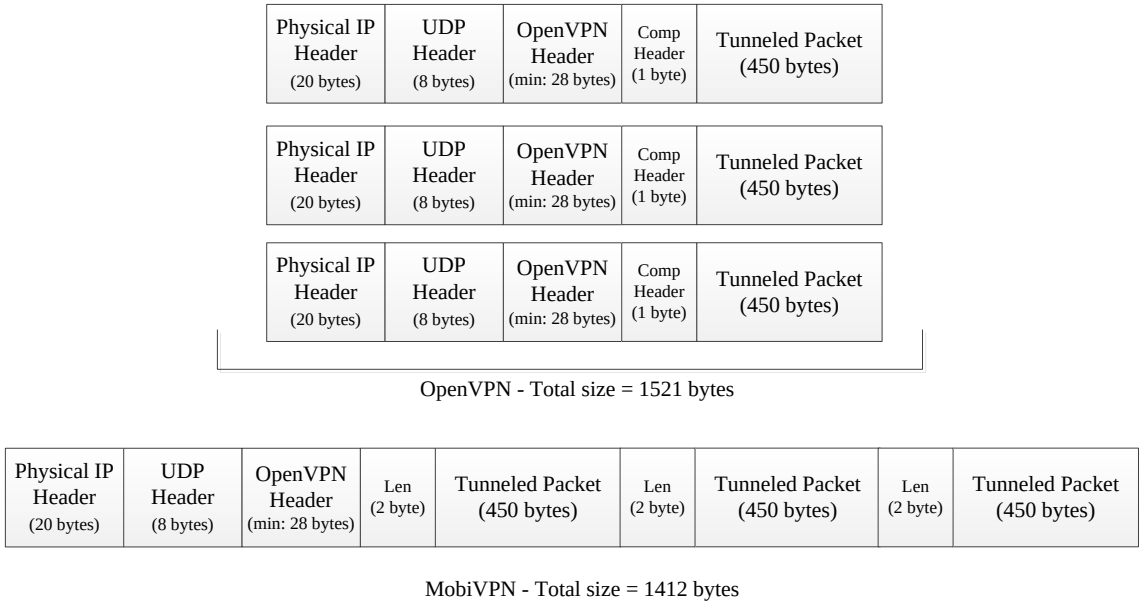

Figure 5.9: Example of Three Compressed Packets in OpenVPN Vs. MobiVPN.

Figure 5.9 shows a motivating example for aggregation. In this example, three compressed packets of the size of 450 bytes were to be sent. OpenVPN would send each packet individually, adding an IP header, UDP head and an OpenVPN header

to each packet. Also, the payload is perpended with a 1-byte field to indicate whether or not the packet is compressed. This results in a total of 1521 bytes to be sent. In MobiVPN case, the same three packets are sent in one packet of 1412 bytes long, which is equal to a 7.17% reduction in size. The more packets that are aggregated, the less data to be transferred.

Assuming n packets are aggregated, the percentage of the minimum size reduction by aggregation can be calculated as:

$$Reduction\% = \frac{56 + \Sigma_{i=1}^{n} 2 + Packet_i.len}{\Sigma_{i=1}^{n} 57 + Packet_i.len} \tag{5.5}$$

The aggregation function takes place between the compression and the fragmentation. Figure 5.10 shows the packet processing sequence in MobiVPN. A packet belongs to a flow where compression is turned off will be forwarded immediately to the fragmentation module, skipping both compression and aggregation. However, if the packet is to be compressed, it will compressed and then stored in the `aggregate_buffer` by the aggregation module. This buffer is big enough to hold one full-size packet. As soon as a compressed packet is stored in the buffer, a timer is initialized to make sure that aggregated packets are emitted after a certain `aggregate_threshold` of time has passed. Therefore, the aggregation module emits an aggregated packet if one of the following two conditions hold:

- A new compressed packet cannot fit in the `aggregate_buffer`.
- The aggregation timer is triggered. At this point, no more delay is tolerated and the aggregated packet is sent to the fragmentation module even if it contains just one compressed packet.

While Figure 5.10 shows the process of sending a packet, it was noted that the reverse of this process is performed when receiving packets. The external multiplexer

reads a packet from the VPN UDP socket, decrypts it , assembles fragmented packets, disaggregates compressed packets, decompresses and finally writes the packet to a buffer where the internal multiplexer can pick it and send it through the TUN interface to the local application.

The disaggregation process is carried out by splitting the payload into several packets using the perpended packet length.

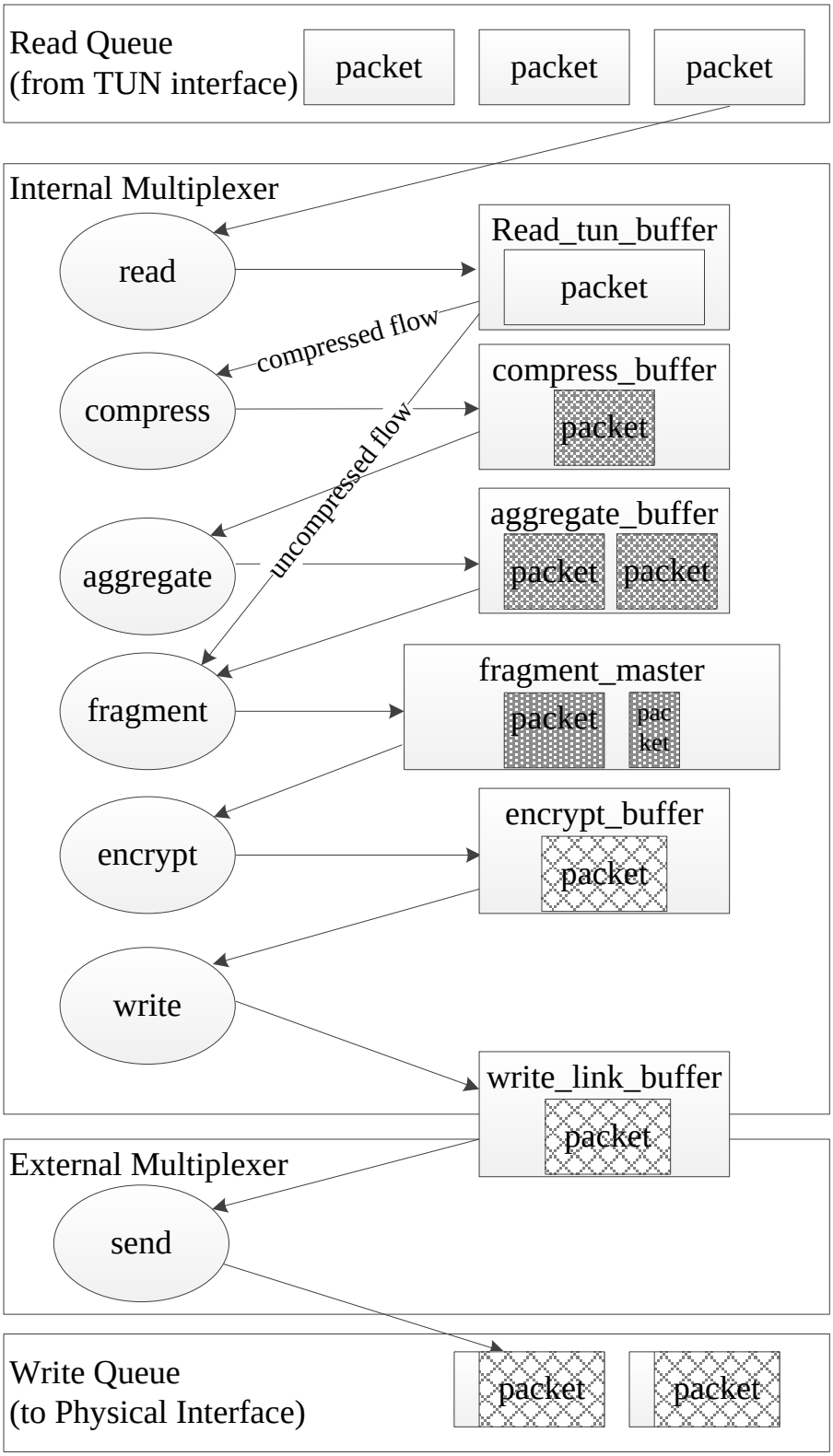

Figure 5.10: Packet Processing in MobiVPN.

5.6 Implementation

Our flow-based adaptive compression was implemented by modifying OpenVPN 2.3.10. Presented in the following sections are the algorithms that were developed for the two modules presented in our design.

5.6.1 FAC Module

This module is called from within the internal multiplexer. In particular, it is called by the method *process_incoming_tun* which is responsible for processing packets read from the TUN interface. Basically, the module receives a packet stored in `read_tun_buffer` and decides whether or not to compress it. The module also decides whether or not compression should continue based on how well the compression did during the sampling period. Algorithm 9 is the soul of this module. The flows information as shown in Table 5.1 are stored in a hash table, named $flows\_tbl$.

5.6.2 Compressed Packets Aggregation Module

This module implements two algorithms: 1) the aggregation module, which is called within the internal multiplexer to aggregate compressed packets before they are dispatched to the fragmentation module, 2) the disaggregation module, which is called by the external multiplexer when it receives packets from the remote VPN. This module reverses the work of the aggregation module. Algorithm 10 implements the aggregation module functionality, while Algorithm 10 implements the functionality of the disaggregation module. The aggregation module uses Linux's *clock_gettime* with `CLOCK_MONOTONIC` to measure the compressed packets stays in the `aggregate_buffer`. This function was used since it provides a nano-second accuracy.

Algorithm 9 Flow-based Adaptive Compression

```
1: procedure COMPRESS_PACKET (c : context, p : packet)
2:     f ← LOOKUP_FLOW_PROFILE(c.flows_tbl, p)
3:     generic = false
4:     if f = NULL then
5:         f ← CREATE_FLOW_PROFILE(p)
6:         if f = NULL then
7:             f ← GET_GENERIC_FLOW_PROFILE()
8:             generic = true
9:         else
10:            ADD_FLOW_PROFILE(f, f_tbl)
11:        end if
12:    end if
13:    if p.len ≤ 100 then
14:        c.buf ← PERPEND(0x00,p); return                    ▷ end procedure
15:    end if
16:    if f.comp_state = ON then
17:        cp ← LZO_COMPRESS(p); ; f.n_total+ = p.len; f.n_comp+ = cp.len
18:        if cp.len < p.len then
19:            cp ← PERPEND(cp.len, cp)
20:            AGGREGATE_PACKET(c, cp)
21:        else
22:            c.buf ←PERPEND(0x00,p)
23:        end if
24:    else
25:        c.buf ← PERPEND(0x00,p)
26:    end if
27:    if now > f.sampling_time then
28:        f.n_total ← 0; ; f.n_comp ← 0
29:        if f.comp_state = OFF then
30:            f.comp_state ← ON
31:            SET_SAMPLING_TIME(f, ON, generic)
32:        else
33:            if FCR(f) > 5% then SET_SAMPLING_TIME(f, ON, generic)
34:                f.increment ← 1s
35:            elseSET_SAMPLING_TIME(f, OFF, generic)
36:            end if
37:        end if
38:    end if
39: end procedure
```

Algorithm 9: Flow-based Adaptive Compression (Continued)

```
40: procedure SET_SAMPLING_TIME(flow, state, generic)
41:     if state = ON then
42:         if generic then
43:             flow.sampling_time ← now + 200ms
44:         else
45:             flow.sampling_time ← now + 2s
46:         end if
47:     else
48:         flow.increment = MAX(flow.increment × 2, 16)
49:         flow.sampling_time ← now + flow.increment
50:     end if
51: end procedure
```

Algorithm 10 Packets Aggregation

```
 1: procedure AGGREGATE_PACKET(c : context, cp : packet)
 2:     if c.aggregate_buf.len = 0 then
 3:         c.aggregate_buf ← cp
 4:         t ← CLOCK_GETTIME(CLOCK_MONOTONIC, c.aggregate_timer)
 5:         t ← t + 10ms
 6:         RESET_AGGREGATION_TIMER(c, t)
 7:     else
 8:         if c.aggregate_buf.capacity < cp.len then
 9:             c.buf ← c.aggregate_buf
10:             CLEAR_BUFFER(c.aggregate_buf)
11:             c.aggregate_buf ← cp
12:             t ← CLOCK_GETTIME(CLOCK_MONOTONIC, c.aggregate_timer)
13:             t ← t + 10ms
14:             RESET_AGGREGATION_TIMER(c, t)
15:         else
16:             APPEND(c.c.aggregate_buf, cp)
17:             c.c.aggregate_buf.capacity− = cp.len
18:         end if
19:     end if
20: end procedure
21: procedure AGGREGATATION_TIMER_WAKEUP(c : context)
22:     c.buf ← c.aggregate_buf
23:     c.aggregate_buf.capacity+ = c.aggregate_buf.len
24:     ENCRYPT_SIGN(c, c.buf)
25:     PROCESS_OUTGOING_LINK(c)
26: end procedure
```

Algorithm 11 Packets Disaggregation

```
1: procedure DISAGGREGATE_PACKET(c : context, cp : packet)
2:     stop ← false
3:     while ¬stop do
4:         len ← GET_LENGTH(c.aggregate_buf,c.aggregate_buf.data)
5:         if len = 0 then
6:             c.buf =BUFFER_ADVANCE(c.aggregate_buf, 2)        ▷ uncompressed packet
7:         else
8:             tmp_buf = COPY_BUFFER(c.aggregate_buf, c.aggregate_buf.data+2, len)
9:             c.buf = tmp_buf
10:            PROCESS_OUTGOING_TUN(c)
11:            c.aggregate_buf =BUFFER_ADVANCE(c.aggregate_buf, len + 2)
12:        end if
13:        if c.aggregate_buf.data = NULL then
14:            stop = true
15:            CLEAR_BUFFER(c.buf)
16:        end if
17:    end while
18: end procedure
```

5.7 Performance Evaluation

Our evaluation is performed in a local testbed which was described in Section 5.7.1. In the evaluation, the performance of the flow-based adaptive compression (MobiVPN-FAC) and OpenVPNâĂŹs compression options were compared: Adaptive Compression (OpenVPN-AC), Compress-all (OpenVPN-C) and no compression (OpenVPN-NC). The evaluation was performed on two datasets: the artificial dataset and a real mobile traffic dataset.

Three criteria were considered in our evaluation: the time it takes to transmit the data, which impacts the effective throughput; the amount of bytes transmitted; and, finally, the number of compression operations performed. A good compression scheme is a one that increases the effective throughput while decreasing the amount of bytes transmitted and the number of compression operations. In the following sections, the outcome of the evaluation is presented.

5.7.1 Testbed Setup

The testbed was setup in a local environment, as shown in Figure 5.11.The VPN server and the application server were installed in virtual machines running on VMware Fusion for Macbook. These VMs ran Ubuntu 16.04 with 2GB RAM. The client VM was hosted on the same Macbook and ran Ubuntu 12.04 with 2GB RAM.

The VPN server was connected to two networks, a private network (10.0.100.0/24) along with the application server. The VPN server was also connected to another local network (192.168.100.0/24) to which the client VM was also connected. The application server ran two application servers: Apache 2.2.20 and iperf 2.0.5. The Apache server was configured with default settings except that we turned off its compression functionality.

Finally, OpenVPN's can print out compression statistics through the management interface. OpenVPNâĂŹs source code was modified so that the number of compression operations was printed out along with the compression statistics. The time measurements were recorded using Linux's `time` command. The measurements reported the average of four trials where the order of tested compression option was alternated with each trial.

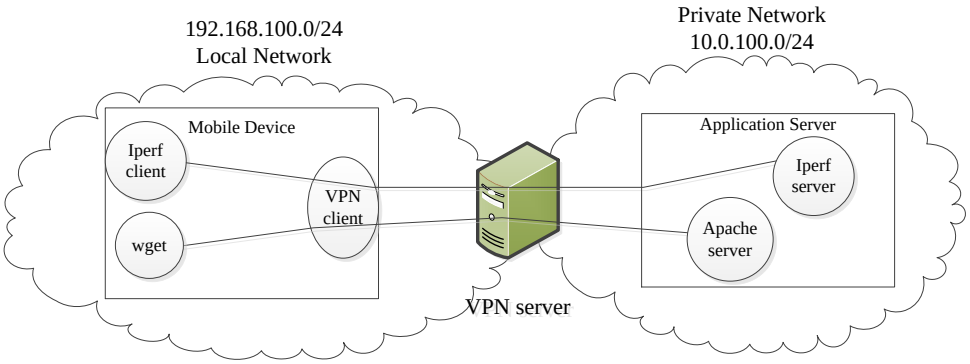

Figure 5.11: Compression Testbed.

5.7.2 Evaluation with Artificial Dataset

The reason for this evaluation was to have a baseline for the performance of the four compression options. A data set, similar to the artificial dataset from the Canterbury corpus by (Powel, 2001), was used. The difference was that the current study's dataset contained larger files than that of Canterbury corpus.

For this evaluation, two files of the size 75×10^6 bytes each (approximately 71.53 MB) were used. One file was highly compressible and contained a long string of a repeated letter "a", and named "*compressible.txt*". The second file was incompressible and of the same length and named "*incompressible.bin*". It was generated using the following Linux command: `head -c 75MB </dev/urandom >incompressible.bin`

A shell-script was used that calls *iperf* client to send out these files to *iperf* server. When transmitting multiple files at the same time, multiple *iperf* commands were used so that each file was sent in an independent flow. The results of the experiments of this dataset are presented in the following sections.

Transmission of One File in One Flow

In this testing case, the files *compressible.txt* and *incompressible.bin* were transmitted separately. Each file was transmitted in a separate VPN session. This was done so that the transmission of one file does not affect the adaptive compression decision when sending the next file.

Figure 5.12 shows the results of sending the file *compressible.txt* with the four compression options. Figure 5.12-A shows that OpenVPN-NC was had the worst effective throughput, as it did not perform any compression. Although, OpenVPN-C and OpenVPN-AC performed almost the same number of compression operations as seen in Figure 5.12-C, and both sent the same amount of data as seen in Figure 5.12-B, OpenVPN-C yielded slightly better effective throughput due to the absence of the computation overhead that exists in OpenVPN-AC.

MobiVPN-FAC yielded the best effective throughput, as it sent less data than both OpenVPN-C and OpenVPN-AC. The data savings for OpenVPN-C, OpenVPN-AC and MobiVPN-FAC were 91.9%, 91.9% and 94.44%, respectively. The additional data savings in MobiVPN-FAC were due to the aggregation process where the average packet size was 1354 bytes compared to 159 bytes in both OpenVPN-C and OpenVPN-AC, as indicated in Figure 5.12-D.

Figure 5.13 shows the results of sending the file *incompressible.bin*. In this case, OpenVPN-NC yielded the best effective throughput due to the absence of any compression operations. MobiVPN-FAC's effective throughput was only 0.46% less than

OpenVPN-NC. This is because it only performed 3288 compression operations, while OpenVPN-C and OpenVPN-AC performed 54,908 and 9206 compression operations, respectively.

Figure 5.12: Performance Measurements When Sending the File "compressible.txt".

Figure 5.13: Performance Measurements When Sending the File "incompressible.bin".

Transmission of Two Files in Two Flows

In this testing case, the two files "compressible.txt" and "incompressible.bin" were sent in two flows. This case showed a more significant impact of our flow-based adaptive compression.

MobiVPN-FAC yielded the most effective throughput, as illustrated in Figure 5.14-A. This was due to the fact that, just like OpenVPN-C, it was able to compress the packet of the "compressible.txt" flow. However, it did 46.86% fewer compression operations than OpenVPN-C. It also was able to send the least amount of data (75.36 MB), compared to 81.79 MB sent by OpenVPN-C. The reason for this data saving was twofold: 1) the packet aggregation process, 2) LZO yielded a better compression ratio with MobiVPN-FAC, as it did not attempt to compress the file "incompressible.bin" except for the sampling periods. This made LZO's window-based dictionary more effective in finding repeated strings.

An important observation we noticed during the experiment trials was that MobiVPN-FAC was more consistent in its adaptive behavior than OpenVPN-AC. It always aborted compression of "incompressible.bin", while OpenVPN-AC, would continue the compression of the tunnel in some trials while turning it off in others.

5.7.3 Evaluation with Mobile Traffic Dataset

The goal in this experiment was to evaluate the effectiveness of the different compression options with real mobile traffic. In order to collect a good representative sample of mobile traffic data, the mobile usage statistical study presented in (Lella *et al.*, 2015) was used. In this study, the top used mobile applications were reported in terms of the time spent by mobile users using these applications. Figure 5.15 shows

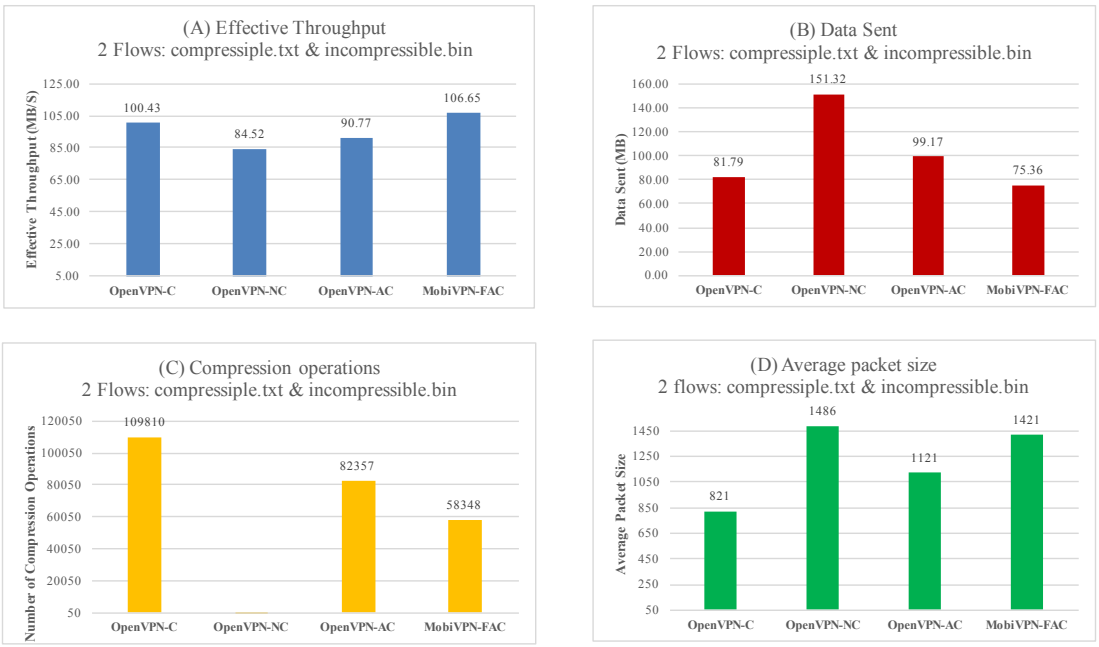

Figure 5.14: Performance Measurements When Sending the Two File "compressible.txt" and "incompressible.bin" in Separate Flows.

the percentage of the time spent on mobile applications by the mobile users in the study.

Mobile Traffic Collection Methodology

In this section, how the mobile traffic data were collected is explained. The following steps were performed to collect the data:

1. An iPhone 6 was used as the mobile client.
2. Both Wifi and cellular interfaces in the mobile device were disabled .
3. USB tethering was used between the mobile device and a Macbook. The Macbook's Internet connection was shared with the mobile device.
4. Wireshark was run and captured all the packets originated from or destined to the mobile device.

5. Mobile applications in the mobile device from the categories in Figure 5.15 were used for a total of 15 minutes. The time spent on each category matches their percentage in Figure 5.15. The applications we used and the time we spent collecting data from each category are shown in Table 5.2.

6. After the packet capturing concluded, the resulting pcap file was fed to a tool named "Split-Cap" that was developed by (Hjelmvik, 2017).

7. With Split-Cap, the payload of each packet was extracted, and each flow was saved in one file with the payload of the flow's packet is concatenated.

8. This process resulted in 447 files in which each file contained data sent from one flow.

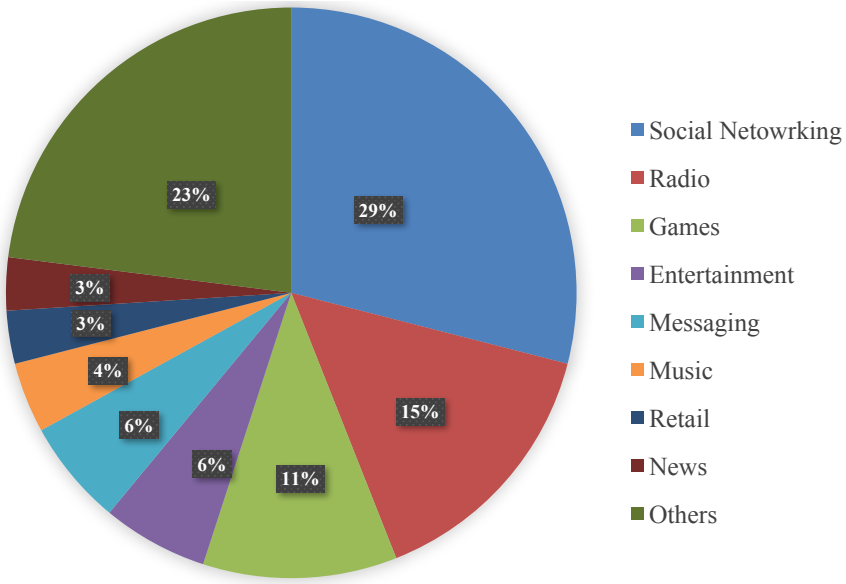

Figure 5.15: Time Spent on Mobile Applications. "Source: ComScore Media Metrix MP and Mobile Metrix, U.S., 2015".

Table 5.2: Time Spent During Data Collection of Mobile Applications.

Category	**Time Spent (M)**	**Used Applications**
Social Netowrking	4:21	Twitter/ Facebook/ Snapchat
Radio	2:15	Pandora
Games	1:39	Zynga Poker
Entertainment	0:54	Youtube
Messaging	0:54	Whatsapp
Music	0:36	Spotify
Retail	0:27	Sears.com
News	0:27	CNN.com / CNN App
Others	3:27	App Update/ Craigslist/ FTP
Total	**15:00**	

Performing the Experiment

The 447 files were hosted in the Apache web server. Then, a shell-script was written that used *wget* to retrieve the files from the web server. Each file was fetched using a separate *wget* command, and the commands were run simultaneously. A "no-cache" option was aldo used so that no downloaded file is cached. Below is a sample of executing two *wget* commands:

```
wget --no-cache 10.0.100.2/../iphone-traffic.pcap.TCP_104-X-...bin &
wget --no-cache 10.0.100.2/../iphone-traffic.pcap.TCP_110-X-...bin &
...
wait
```

Results and Discussion

Figure 5.16 summarizes the results of this experiment. In terms of throughput effectiveness, MobiVPN-FAC was the highest with 23.71 Mbps, whereas OpenVPN-C, OpenVPN-NC, and OpenVPN-AC had an effective throughput of 22.36, 21.79, and 21.85 Mbps, respectively. MobiVPN-FAC was able to reduce 17.58% the total sent data when no compression was used. MobiVPN-C was able to reduce 17.06% of the original data. However, as Figure 5.16-C shows, the number of compression operations MobiVPN-FAC performed were 66.55% fewer than that of OpenVPN-C. The additional data saving MobiVPN-FAC gained was due to the packet aggregation process, in addition to the possibility of LZO compression string dictionary being affected by the incompressible strings from the incompressible flows.

The adaptive strategy of OpenVPN-AC was not effective as it saved only 0.83% of the sent traffic. The no compression option had the least effective throughput and the most data sent.

Finally, it is worth noting that although the data saving is high in both MobiVPN-FAC and MobiVPN-C, their effective throughput does not reflect the same ratio. This is due to the fact that the VPN does its packet processing in a per packet basis. If applications data were to be compressed in bulk before they were passed to the VPN, that data saving would have much more influence on the effective throughput.

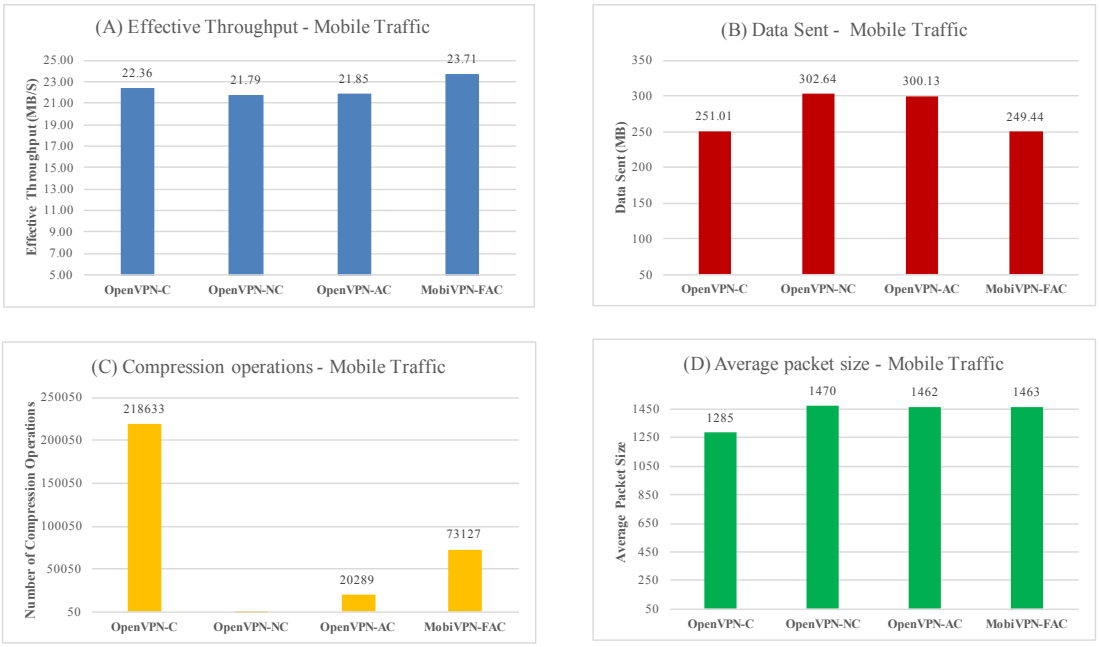

Figure 5.16: Performance Measurements When Sending The Mobile Traffic.

5.8 Conclusion

In this chapter, a flow-based adaptive compression for MobiVPN. was designed and developed. This scheme was designed to treat each tunneled flow independently. An aggregation process was also introduced to aggregate small compressed packets in order to reduce the overhead of sending them individually.

The flow-based strategy proved its feasibility through the empirical experiments. It always produced the more data savings with much fewer compression operations. For mobile devices where data consumption is costly, our system can significantly

reduce transmitted data when compression opportunities are present while performing compression operations when feasible.

Chapter 6

CONCLUSION

6.1 Contributions

This dissertations contains, a description of how MobiVPN was developed to overcome the limitations of OpenVPN in mobile environments. Three features were introduced in MobiVPN in order to satisfy the mobile VPN requirements. The contributions made in this dissertation are:

- **Fast and Lightweight VPN session resumption**:
  - A new protocol was developed for a lightweight VPN session resumption that allows both MobiVPN clients and server to resume an already-established VPN tunnel.
  - A system design and implementation were discussed and this feature of MobiVPN was evaluated compared to OpenVPN.
  - MobiVPN was able to reduce the time to resume a VPN tunnel after a mobility event by an average of 97.19%.
- **TCP-based Applications Persistence**:
  - A system design and implementation were provided to prevent TCP flows from terminating during disconnection periods.
  - The TCP flows were able to resume as soon as the VPN tunnel was resumed.

- Two options were provided to suspend and resume TCP applications, with and without buffering. The TCP sending rate can be maintained or recovered, if needed, during disconnection periods when buffering is enabled. Mathematical models were provided for how much buffering is required in order to recover the TCP sending rate.
- The evaluations showed that MobiVPN can protect a TCP socket from terminating due to TCP timeout configurations. In addition, the evaluation showed that MobiVPN was able to provide fast resumption of TCP flows after reconnection with improved TCP performance when disconnections occur with an average of 30.08% increase in throughput in the experiments when buffering is used, and an average of 20.93% of increased throughput for flows that are not buffered.

- **Flow-based Adaptive Compression**:
  - A flow-based adaptive compression algorithm for MobiVPN was designed and implemented.
  - The decision to perform compression or not was made to treat flows independently.
  - A packet aggregation step was added to packet processing. The aim of this step was to aggregate short compressed packets in order to reduce the overhead of transmitting small packets.
  - Evaluations showed that MobiVPN with flow-based adaptive compression was able to reduce the traffic and increase effective throughput while performing a lesser number of compression operations. In an experiment with real mobile traffic, OpenVPN reduced the amount of bytes transferred by 17.06%, whereas MobiVPN was able to reduce the amount of bytes sent by

17.58%, but used decreased the use of compression operations by 66.55%. OpenVPN's adaptive compression produced inadequate results as it was able only to reduces the transferred packets size by only 0.83%.

6.2 Future Work

Discussed in this section, are some of the potential directions to improve the MobiVPN in future research work. They are

- Currently, in TCP-based application persistence, MobiVPN allocates buffer capacity to flows in a first-come, first-served basis. As the goal of buffering is to recover the sending rate of TCP, it is recognized that not all TCP flows are of the same importance to the user, and the recovery of the sending rate may also not be as important to some flows as it is to others, such as the bursty flows or the short-lived flows. Therefore, a potential improvement is to provide buffering selectively based on either user input or the characteristics of the TCP flow.

- The buffering discussed in this dissertation was modeled for TCP variants that use the AIMD congestion mechanism. More TCP variants can be supported in the future, and MobiVPN will have to be able to recognize which TCP variant is used.

- In the adaptive compression subject, MobiVPN can be improved to change the acceptable compression ratio adaptively. Currently, it is set to 5%. However, this threshold can be adjusted to adapt to the state of the system resources. For example, when the CPU is the bottle-neck, the acceptable compression ratio can be raised so that a lesser number of flows are to be compressed. When the network bandwidth is the bottle-neck, more packets will be waiting to be

transmitted. Therefore, to utilize this waiting time, the acceptable compression ratio can be decreased so that more flows are compressed.

REFERENCES

"Columbitech wireless VPN technical description", White paper, Columbitech, URL `http://www.columbitech.com/img/2008/3/5/16245.pdf` (2007).

Adrangi, F. and H. Levkowetz, "Problem statement: Mobile IPv4 traversal of virtual private network (VPN) gateways", Tech. rep., RFC 4093, August (2005a).

Adrangi, F. and H. Levkowetz, "Problem statement: Mobile ipv4 traversal of virtual private network (vpn) gateways", RFC 4093 (2005b).

Ahmat, D., M. Barka and D. Magoni, "Semos: A middleware for providing secure and mobility-aware sessions over a p2p overlay network", in "8th EAI International Conference on e-Infrastructure and e-Services for Developing Countries", (2016).

Ahmat, D. and D. Magoni, "MUSeS: Mobile user secured session", in "Wireless Days (WD), 2012 IFIP", pp. 1–6 (IEEE, 2012).

Al-Ameen, M. N. and R. Hasan, "The mechanisms to decide on caching a packet on its way of transmission to a faulty node in wireless sensor networks based on the analytical models and mathematical evaluations", in "Sensing Technology, 2008. ICST 2008. 3rd International Conference on", pp. 336–341 (IEEE, 2008).

Alshalan, A., S. Pisharody and D. Huang, "Mobivpn: A mobile vpn providing persistency to applications", in "Computing, Networking and Communications (ICNC), 2016 International Conference on", pp. 1–6 (IEEE, 2016a).

Alshalan, A., S. Pisharody and D. Huang, "A survey of mobile vpn technologies", IEEE Communications Surveys & Tutorials **18**, 2, 1177–1196 (2016b).

Bakre, A. and B. Badrinath, "I-TCP: Indirect TCP for mobile hosts", in "Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on", pp. 136–143 (IEEE, 1995).

Benenati, D., P. M. Feder, N. Y. Lee, S. Martin-Leon and R. Shapira, "A seamless mobile VPN data solution for CDMA2000,* UMTS, and WLAN users", Bell Labs technical journal **7**, 2, 143–165 (2002).

Braun, T. and M. Danzeisen, "Secure mobile IP communication", in "Local Computer Networks, 2001. Proceedings. LCN 2001. 26th Annual IEEE Conference on", pp. 586–593 (IEEE, 2001).

Brown, K. and S. Singh, "M-tcp: Tcp for mobile cellular networks", ACM SIGCOMM Computer Communication Review **27**, 5, 19–43 (1997).

Byun, H. and M. Lee, "Network architecture and protocols for BGP/MPLS based mobile VPN", in "Information Networking. Towards Ubiquitous Networking and Services", pp. 244–254 (Springer, 2008).

Chen, J.-C., J.-C. Liang, S.-T. Wang, S.-Y. Pan, Y.-S. Chen and Y.-Y. Chen, "Fast handoff in mobile virtual private networks", in "Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks", pp. 548–552 (IEEE Computer Society, 2006).

Chen, S., S. Ranjan and A. Nucci, "Ipzip: A stream-aware ip compression algorithm", in "Data Compression Conference, 2008. DCC 2008", pp. 182–191 (IEEE, 2008).

Chunle, F., H. Qinggang, W. Bailing and H. Xixian, "A communication supportable generic model for mobile vpn on android os", in "Computers and Communication (ISCC), 2016 IEEE Symposium on", pp. 1039–1046 (IEEE, 2016).

Comer, D. and D. L. Stevens, *Intenetworking With Tcp/Ip* (rentice-Hall, 2003).

Data, D., "Secure mobility survey report", URL `http://www.dimensiondata.com/Global/DownloadableDocuments/SecureMobilitySurveyFindingsReport.pdf` (2014).

Devarapalli, V., R. Wakikawa, A. Petrescu and P. Thubert, "Network mobility (NEMO) basic support protocol", Tech. rep., RFC 3963, January (2005).

Dinh, H. T., C. Lee, D. Niyato and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches", Wireless communications and mobile computing **13**, 18, 1587–1611 (2013).

Dutta, A. and H. Schulzrinne, *Mobility Protocols and Handover Optimization: Design, Evaluation and Application* (John Wiley & Sons, 2014).

Dutta, A., T. Zhang, S. Madhani, K. Taniuchi, K. Fujimoto, Y. Katsube, Y. Ohba and H. Schulzrinne, "Secure universal mobility for wireless internet", ACM SIGMOBILE Mobile Computing and Communications Review **9**, 3, 45–57 (2005).

Eronen, P., "Ikev2 mobility and multihoming protocol (mobike)", URL `http://www.ietf.org/rfc/rfc4555.txt` (2006).

Feder, P., N. Lee and S. Martin-Leon, "A seamless mobile VPN data solution for UMTS and WLAN users", in "3G Mobile Communication Technologies, 2003. 3G 2003. 4th International Conference on (Conf. Publ. No. 494)", pp. 210–216 (IET, 2003).

Geneiatakis, D., T. Dagiuklas, G. Kambourakis, C. Lambrinoudakis, S. Gritzalis, S. Ehlert, D. Sisalem *et al.*, "Survey of security vulnerabilities in session initiation protocol.", IEEE Communications Surveys and Tutorials **8**, 1-4, 68–81 (2006).

Goff, T., J. Moronski, D. S. Phatak and V. Gupta, "Freeze-TCP: A true end-to-end TCP enhancement mechanism for mobile environments", in "INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE", vol. 3, pp. 1537–1545 (IEEE, 2000).

Gurtov, A., *Host identity protocol (HIP): towards the secure mobile internet*, vol. 21 (John Wiley & Sons, 2008).

Heydari, V., S.-M. Yoo and S.-i. Kim, "Secure vpn using mobile ipv6 based moving target defense", in "Global Communications Conference (GLOBECOM), 2016 IEEE", pp. 1–6 (IEEE, 2016).

Heyman, K., "A new virtual private network for today's mobile world", Computer **40**, 12, 17–19 (2007).

Hjelmvik, E., "Split-cap", URL `https://www.netresec.com/?page=SplitCap` (2017).

Hovestadt, M., O. Kao, A. Kliem and D. Warneke, "Evaluating adaptive compression to mitigate the effects of shared i/o in clouds", in "Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on", pp. 1042–1051 (IEEE, 2011).

Huang, D., X. Zhang, M. Kang and J. Luo, "MobiCloud: building secure cloud framework for mobile computing and communication", in "Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on", pp. 27–34 (IEEE, 2010).

Huang, S.-C., Z.-H. Liu and J.-C. Chen, "SIP-based mobile VPN for real-time applications", in "Wireless Communications and Networking Conference, 2005 IEEE", vol. 4, pp. 2318–2323 (IEEE, 2005).

Jeannot, E., B. Knutsson and M. Bjorkman, "Adaptive online data compression", in "High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on", pp. 379–388 (IEEE, 2002).

Knutsson, B. and M. Björkman, "Adaptive end-to-end compression for variable-bandwidth communication", Computer Networks **31**, 7, 767–779 (1999).

Koponen, T., P. Eronen, M. Särelä *et al.*, "Resilient connections for ssh and tls.", in "USENIX Annual Technical Conference, General Track", pp. 329–340 (2006).

Krintz, C. and S. Sucu, "Adaptive on-the-fly compression", Parallel and Distributed Systems, IEEE Transactions on **17**, 1, 15–24 (2006).

Lella, A., A. Lipsman and B. Martin, "The 2015 u.s. mobile app report", Online, URL `https://www.comscore.com/Insights/Presentations-and-Whitepapers/2015/The-2015-US-Mobile-App-Report` (2015).

Liu, Z.-H., J.-C. Chen and T.-C. Chen, "Design and analysis of SIP-based mobile VPN for real-time applications", Wireless Communications, IEEE Transactions on **8**, 11, 5650–5661 (2009).

Maddah, R. and S. Sharafeddine, "Energy-aware adaptive compression for mobile-to-mobile communications", in "Proc. IEEE Symposium on Spread Spectrum and Applications", (2006).

Mittal, R., N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats *et al.*, "Timely: Rtt-based congestion control for the datacenter", in "ACM SIGCOMM Computer Communication Review", vol. 45, pp. 537–550 (ACM, 2015).

Motgi, N. and A. Mukherjee, "Network conscious text compression system (nctcsys)", in "Information Technology: Coding and Computing, 2001. Proceedings. International Conference on", pp. 440–446 (IEEE, 2001).

Oberhumer, M. F., "Lzo-a real-time data compression library", URL `http://www.oberhumer.com/opensource/lzo/` (2008).

OpenVPN Technologies, "OpenVPN", URL `http://www.openvpn.net` (2011).

Park, K.-W. and K. H. Park, "Accent: Cognitive cryptography plugged compression for ssl/tls-based cloud computing services", ACM Transactions on Internet Technology (TOIT) **11**, 2, 7 (2011).

Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", Tech. Rep. 2988, URL `http://www.ietf.org/rfc/rfc2988.txt` (2000).

Politopoulos, P. I., E. P. Markatos and S. Ioannidis, "Evaluation of compression of remote network monitoring data streams", in "Network Operations and Management Symposium Workshops, 2008. NOMS Workshops 2008. IEEE", pp. 109–115 (IEEE, 2008).

Postel, J., "Transmission control protocol", STD 7, RFC Editor, URL `http://www.rfc-editor.org/rfc/rfc793.txt`, `http://www.rfc-editor.org/rfc/rfc793.txt` (1981).

Powel, M., "The canterbury corpus", URL `http://corpus.canterbury.ac.nz/index.html` (2001).

Pu, C. and L. Singaravelu, "Fine-grain adaptive compression in dynamically variable networks", in "Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on", pp. 685–694 (IEEE, 2005).

Pulkkis, G., K. Grahn, M. Mårtens and J. Mattsson, "Mobile virtual private networking", in "Future Internet-FIS 2009", pp. 57–69 (Springer, 2010).

Rosado, J. J. A., "Mobile virtual private networks", US Patent 8,544,080 (2013).

Schonwalder, J., G. Chulkov, E. Asgarov and M. Cretu, "Session resumption for the secure shell protocol", in "Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on", pp. 157–163 (IEEE, 2009).

Shimamura, M., T. Ikenaga and M. Tsuru, "Compressing packets adaptively inside networks", IEICE transactions on communications **93**, 3, 501–515 (2010).

Shneyderman, A., A. Bagasrawala and A. Casati, "Mobile VPNs for next generation GPRS and UMTS networks", URL `http://esoumoy.free.fr/telecom/tutorial/3G-VPN.pdf` (2000).

Shneyderman, A. and A. Casati, *Mobile VPN: delivering advanced services in next generation wireless systems* (John Wiley & Sons, 2003).

So-In, C., R. Jain and G. Dommety, "Pets: persistent tcp using simple freeze", in "Future Information Networks, 2009. ICFIN 2009. First International Conference on", pp. 97–102 (IEEE, 2009).

StatCounter, "Mobile and tablet internet usage exceeds desktop for first time worldwide", Online, URL `http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide` (2016).

Tiendrebeogo, T., D. Magoni and O. Sié, "Virtual internet connections over dynamic peer-to-peer overlay networks", in "INTERNET 2011, The Third International Conference on Evolving Internet", pp. 58–65 (2011).

Tzvetkov, V. D., *Virtual Private Networks for mobile environments. Development of protocol for mobile security and algorithms for location update.*, Ph.D. thesis, TU Darmstadt (2010).

Uskov, A. V., "Information security of ipsec-based mobile vpn: authentication and encryption algorithms performance", in "Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on", pp. 1042–1048 (IEEE, 2012).

Vaarala, S. and E. Klovning, "Mobile IPv4 traversal across IPsec-based VPN gateways", (2008a).

Vaarala, S. and E. Klovning, "Mobile ipv4 traversal across ipsec-based vpn gateways", RFC 5265, RFC Editor (2008b).

Wiseman, Y., K. Schwan and P. Widener, "Efficient end to end data exchange using configurable compression", ACM SIGOPS Operating Systems Review **39**, 3, 4–23 (2005).

Xiao, Y., M. Siekkinen and A. Ylä-Jääski, "Framework for energy-aware lossless compression in mobile services: The case of e-mail", in "Communications (ICC), 2010 IEEE International Conference on", pp. 1–6 (IEEE, 2010).

Xu, R., Z. Li, C. Wang and P. Ni, "Impact of data compression on energy consumption of wireless-networked handheld devices", in "Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on", pp. 302–311 (IEEE, 2003).

Yonan, J., *OpenVPN 2.2 man page*, URL `https://community.openvpn.net/openvpn/wiki/Openvpn22ManPage` (2008).

Yoshino, M., H. Koga, M. Shimamura and T. Ikenaga, "Adaptive online compressing schemes using flow information on advanced relay nodes", ICN 2014 p. 109 (2014).

Zúquete, A. and C. Frade, "Fast VPN mobility across wi-fi hotspots", in "Security and Communication Networks (IWSCN), 2010 2nd International Workshop on", pp. 1–7 (IEEE, 2010).