UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science Curriculum

Märt Bakhoff

Integrating VDE into the F2F framework

Master's Thesis (30 ECTS)

Supervisor:	Artjom Lind, MSc
Supervisor:	Prof. Eero Vainikko

Tartu 2015

Integrating VDE into the F2F framework

Abstract:
In this thesis the Virtual Distributed Ethernet (VDE) extension is introduced for the Friend-to-Friend (F2F) framework. F2F is an existing framework for creating Peer-to-Peer (P2P) private computing clouds developed by the Distributed Systems group at University of Tartu. F2F works by bootstrapping a P2P network from the social networks or instant-messenger networks in order to configure direct connections between the joined peers. These connections are not fault tolerant and can be used only through the F2F API. The new VDE extension we discuss here builds a high performance Virtual Ethernet topology, which adds fault recovery features and Spanning Tree Protocol (STP). The extension is designed to seamlessly integrate with existing virtualization tools that are used in F2F cloud computing. The implementation combines tools from the vde2 project (developed at University of Bologna) with non-blocking input/output libraries and peer communication API provided by the F2F framework. The improved framework provides a fully functional Ethernet network between the F2F peers, which is essential for the F2F client applications running on QEMU virtualization platform.

Keywords: p2p, vde, networking, cloud computing

VDE integreerimine F2F raamistikku

Lühikokkuvõte:
See lõputöö kirjeldab sõpradevahelise arvutuste raamistikule (*F2F framework*) uue laienduse lisamist, mis võimaldab virtuaalsete hajusate võrguühenduste (*VDE*) loomist. F2F raamistik on Tartu Ülikooli hajussüsteemide grupis arendatav tarkvara, mis võimaldab luua privaatseid pilvelahendusi kasutades võrdõiguslike võrgusõlmede (*P2P network*) võrgustikke. F2F kasutab oma võrgustike loomiseks olemasolevad sotsiaalvõrgustikke ja sõnumivahetusprotokolle, mis võimaldavad sõlmedevahelist andmevahetust ja täiendavate ühenduste loomist. Sellised ühendused ei ole veakindlad ja on kättesaadavad ainult läbi F2F programmeerimisliidese. Siin kirjeldatav lahendus kasutab VDE tehnoloogiat, et F2F võrgustikule hea jõudluse, veakindluse ja STP protokolli toega virtuaalne Ethernet võrk lisada. Pakutav virtuaalne võrk on disainitud hõlpsalt F2F raamistikus kasutatavate virtualiseerimistehnoloogiatega integreeruma. Realisatsioon kombineerib Bologna Ülikoolis arendatud vde2 projekti tarkvara, mitteblokeeruva sisend-väljundi mudeli ja F2F raamistiku sõlmedevahelised ühendused. Lõpptulemusena on F2F raamistiku kasutajatel võimalik üles seada täisfunktsionaalne virtuaalne Ethernet võrk, mille saab liidestada erinevate virtuaalmasinatega.

Võtmesõnad: p2p, vde, sidevõrgud, pilvearvutused

Contents

1 Introduction

In this thesis a new network support plugin is introduced for the Friend-to-Friend (F2F) framework developed by the Distributed Systems group at University of Tartu.

The F2F framework is a framework for building distributed applications and setting up private clouds. For communication the F2F framework relies on ad-hoc peer-to-peer networks. The framework simplifies the setup of the peer-to-peer network by taking advantage of the existing social networks and instant messenger protocols when bootstrapping the network. The framework also aims to provide tools for establishing efficient direct connections between the peers after the initial bootstrapping process. Last but not least, the framework provides means to run the user's application using the collective computation power of the peers in the F2F cloud. The current solution relies on running the code directly on the host machine, but a more secure solution would be to set up a virtual machine on each of the F2F cloud member's machine. This way the application would be fully isolated from the host operating system and the members of the F2F cloud would have to worry less about security.

Until now the use of virtual machines in the F2F framework has been limited. Using the widespread approach of creating an operating system level virtual network interface for bridging the F2F network and the virtual machine's network would require the F2F framework to run with root access to the host machine. This would increase the risks[6] to the host machine and restrict the use of F2F framework in high security environments. The solution provided in this thesis leverages the Virtual Distributed Ethernet (VDE) technology. VDE enables building fully functional lightweight virtual Ethernet networks. The resulting F2F network can be attached to a virtual machine without requiring root access. The solution is usable with different virtual machines out of the box. It enables code running in the virtual machines to communicate with other F2F instances using standard network tools.

This thesis covers the extensive practical work that was done to integrate the VDE technology into the F2F framework. The practical work includes designing and building a new plugin for the F2F framework that can bridge the F2F framework's peer-to-peer connections with the VDE virtual network, along with the significant modifications to the existing F2F code to enable the integration. The thesis analyses the system requirements of both the F2F network layer and the F2F VDE plugin. The design choices, alternatives and implementation details are well documented in the corresponding chapters of the thesis.

Chapter 2 of the thesis starts with an overview of F2F computing, the existing functionality and explains the problems the VDE integration will solve. The overview also introduces the existing solutions that try to solve similar problems. Chapter 3 discusses the architectural changes to the F2F framework's network layer needed to enable the VDE integration. Finally, Chapter 4 discusses the implementation details of the new VDE plugin and shows the performance of the resulting solution.

2 Overview

2.1 Introduction to F2F computing

Most of the online services and data processing tools have long been using a centralized client-server model. The main processing is done in large data centers with dedicated servers that use special enterprise level hardware. At the same time, countless computers in people's homes and pockets are spending most of their time idle, even though their combined processing power is comparable to the dedicated data centers. The main reason for this is the connectivity issues between the devices and the lack of tooling support to allow easy access to the unused resources.

The Friend-to-Friend (F2F) framework was created to address these problems and provide intuitive tools for creating distributed applications that can be run on commodity hardware spread around the entire planet. The F2F framework provides the session and presentation layer of the **OSI model**[35], a conceptual model explaining the communication networks and their layers.

The main aim of the F2F framework is to handle connection establishment between the devices in the F2F cloud and to provide a way to distribute and execute applications on the participating devices.

The connection establishment is complicated by the issues with direct connectivity and network address translation as described in Section 2.5.3. The F2F framework uses the existing social networks and instant messenger protocols (such as Google Talk and Facebook chat) to find the members for the F2F cloud and do the initial bootstrapping of the peer-to-peer network (network where device communicate directly with each other without using a central server as an intermediary). Using the existing channels enables the use of advanced connection establishment algorithms such as STUN and TURN (Section 2.5.3).

Once the connections have been established, the F2F framework provides the tools to execute distributed applications on the members of the F2F cloud. Prototypes have been built to experiment with different ways of running the code remotely. The issue of platform native code was solved by using platform neutral languages (such as Python and Java), but this restricted the libraries and languages that were usable with the F2F framework. Experiments with LLVM (low level virtual machine) improved the usability of native code with the F2F framework, but increased the security risks from executing arbitrary code on the F2F cloud members.

The best solution to the code execution problems is using full platform virtualization (for example using the QEMU virtual machine). The distributed code would be packaged into a virtual machine image and distributed over the cloud. Each F2F peer could then run the code without exposing the host machine itself. The application could make more assumptions about the environment and the underlying operating system.

The problem solved in this thesis is integrating the virtual machines with the peer-to-peer connections provided by the F2F network. The solution uses VDE networking (as described in the next Section) which enables creating a virtual network interface on the virtual machine without requiring root access to the host machine or modifications to the virtual machine source code.

Running without root access is an important step for the F2F framework because it significantly reduces the risks for the host machine. The F2F framework will become useful even in high security environments where root access is not possible. Additionally the security risks of running the process as root are avoided - the principle of least privilege is followed, any bugs in the distributed application, VM image or the F2F code cannot harm the system and in case of a security breach the damage is more isolated.

2.2 Introduction to Virtual Distributed Ethernet

Virtualsquare VDE[24] (Virtual Distributed Ethernet) is a project by University of Bologna. The aim of the project is to build an Ethernet compliant virtual network able to interconnect virtual machines even if they are running on different host machines. The project also provides the tools to directly connect to the virtual network, to make the network visible via the operating system level interfaces and to connect the network with different virtual machine technologies.

A VDE network consists of one or many virtual network switches running on different machines and the virtual connections between them. A virtual switch is a program which emulates a physical Ethernet switch (see Section 2.5 for introduction to Ethernet networks). The emulated switch provides all the functionality of a real physical switch: connecting multiple machines into a single network and forwarding packets between them.

The virtual connections between the VDE switches can be implemented in several ways. The switch process maintains virtual ports (in the physical switch analogue, this is where the cables would connect to). A network application can use the ports to send and receive raw Ethernet frames (explained in Section 2.5) and then forward them to another machine where another virtual switch is running. This is also what the F2F VDE plugin does, as seen in Figure 1.

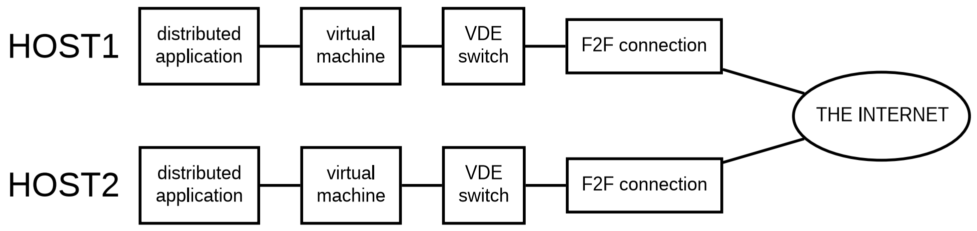

Figure 1: Data flow using the F2F VDE plugin

The main features[25] of the VDE project are:

- consistent behavior with real Ethernet networks
- connectivity between virtual machines and applications
- ability to run without administrative privileges
- support for Spanning Tree Protocol for handling loops in network topology

Most virtual network solutions use a technology called **TUN/TAP devices**[8] to make the virtual network visible to the applications running on the host machine. Tun/tap works by having the operating system create a pseudo network interface. The virtual network solution can then emulate a real network using the created pseudo interface. To the applications, the pseudo interface looks like a regular network interface. To create a TUN/TAP interface, root access to the operating system is required.

The VDE virtual switches and connections do not require root access to start and run, because it they do not rely on TUN/TAP interfaces. Many programs, including the QEMU and VirtualBox virtual machines have built-in support for directly connecting to the VDE switch, bypassing the pseudo interface. This setup is much more secure and can have improved performance due to the reduced overhead.

The possibility of running without root access is one of the main reasons the F2F framework is being integrated with the VDE technology. As a result the entire stack can run without root permissions.

2.3 Requirements for the F2F network layer

The Friend-to-Friend framework has many responsibilities: connection management, the wire protocol, packet assembly/disassembly, dispatching packets to the F2F components, security and job management. In this chapter we will be focusing on the high level requirements for the connection management and packet dispatching.

The main requirement is to support different types of connections between the peers. Different peers can be reachable by different means, for example one pair of peers may be connected by a direct connection while another pair may be inhibited from using a direct connection due to a firewall. In the latter case the peers must use another way of communicating, usually by using a third party as a proxy.

To support different mechanisms the peers are using to communicate with each other, the I/O (input/output) provider abstraction is added to the network layer. An I/O provider consists of an actual direct or indirect connection between a pair of peers and the software implementation in the F2F framework that wraps the connection. The wrapper hides the implementation details of the wrapped connection and makes it usable to other components of the F2F framework (mainly the packet assembly and dispatch components). The different types of connections are explained in depth in Section 2.5.3. The wrapper abstraction is discussed in more detail in Section 3.3.5.

Having different I/O providers means that the packet dispatch component of the F2F framework must support runtime selection of the I/O providers. Runtime selection of the provider means that the F2F framework does not know which I/O providers will be used at compile time and must be able to activate the correct provider later when the program is already running. Incoming connections can dictate the use of any of the supported I/O provider. At the same time, outgoing connections can try to use any of the I/O providers but due to connectivity issues some of the providers may fail to establish a connection. The framework must be able to handle these failures gracefully and retry connecting with another provider. The runtime selection capability is added to the framework as part of adding the I/O provider abstraction (Section 3.3.5).

In addition to being able to retry establishing outgoing connections with different I/O providers, the framework should also be able to handle failures in already established connections. When an already established connection fails then the framework should use all available means to detect the failure and notify other components (and the framework clients) of the failure. The framework should also try to reestablish the connection by either using the same I/O provider or falling back to another provider. The notifications are implemented as part of the F2F core event API (Section 3.5).

The second major requirement for the network layer is cross platform compatibility. The F2F framework aims to be usable on all major platforms, including Linux, Mac, Windows and Android. This means that the core code should only use standard cross platform APIs and language constructs. This requirement is the most difficult to abide by in practice since the C programming language does not include a networking API. All platform specific code must be carefully separated and hidden behind platform neutral abstractions so that it could be easily ported to all required platforms. The I/O provider abstractions partially solve this problem by providing a natural abstraction boundary for platform specific code. At the same time creating platform specific I/O providers severely reduces their reusability.

Using third party libraries (such as SDL[20]) to solve the cross platform networking issues was considered, but it became apparent that this would only shift the issue to another part of the code as integrating the F2F framework with the social networks that it uses to bootstrap the network would still be a platform specific issue. The third party tools would also introduce extra complexity and overhead into the I/O providers that could be avoided by a custom implementation.

The third requirement is safety. As the F2F framework is opening and accepting remote connections it is very important to keep the attack surface minimal and avoid compromising the host machine. The F2F framework must be able to run without administrative/root access. Any inputs received over the network must be reasonably validated. Remote code should only be executed in a highly controlled environment such as a virtual machine instance. The root access requirement will be removed by the VDE network support.

The forth requirement for the network layer is extensibility. Even though there are many ways to connect two machines, the resulting F2F cloud only cares that the data gets transmitted. The developers using the F2F framework should be able to easily develop and use new I/O providers to get the best performance out of their devices and connections. For these reasons, the new I/O provider abstraction should be as simple as possible.

Finally, the F2F framework should have overall good performance. The older versions of F2F that are implemented in a mix of python and C have demonstrated significant latency in lab conditions. Implementing VDE support should take every opportunity to reduce the latency and performance bottlenecks.

2.4 Existing solutions for virtual networks

The idea of building virtual networks is not new. Several solutions exist that can join local area networks over the internet or tunnel Ethernet packets over a secure connection. This chapter refers to the existing solutions that best meet the F2F framework's requirements. Each of them has its strengths and weaknesses. Unfortunately none of them are perfect for the F2F framework in their current state of development.

2.4.1 LogMeIn Hamachi

LogMeIn Hamachi[10] is a well known commercial tool for quickly creating virtual networks for connecting devices in remote locations. It describes itself as follows:

> LogMeIn Hamachi is a hosted VPN service that lets you securely extend LAN-like networks to distributed teams, mobile workers and your gamer friends alike. In minutes.

The Hamachi network supports several modes of operation (Figure 2): mesh, hub-and-spoke and gateway. In mesh mode, each peer is directly connected to every other peer in the network. No central server is used and a new mesh network can be created at any time by any user.

In hub-and-spoke mode, some of the nodes are designated as hubs and act similar to routers. The rest of the nodes (called spokes) connect to the hub nodes. Two spokes never establish a direct connection, so the hubs must relay data between them. Designating all nodes as hubs essentially creates a mesh.

In gateway mode, one node acts as a master gateway. Clients can connect directly to the gateway, but only a single gateway can be active. Additionally, the clients can reach other devices from the network the gateway is connected to without these devices having to run Hamachi.

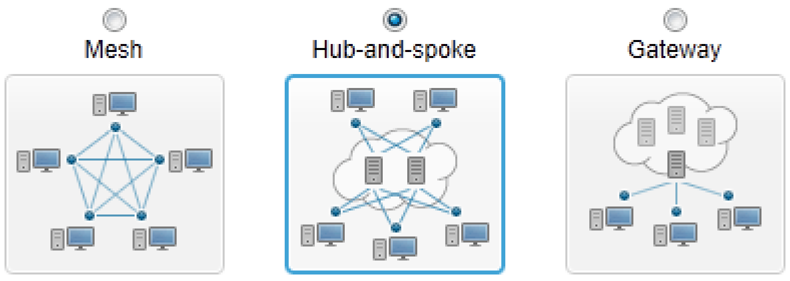

Figure 2: Hamachi network modes

The mesh model would be very suitable for the needs of the F2F framework, because the peers could efficiently communicate with each other and Hamachi would take care of the connection setup. Unfortunately there are several issues that make Hamachi unusable with the F2F framework:

- The network uses central authentication by Hamachi servers. The public keys of the peers are managed by Hamachi. This makes it very difficult to ensure the security of the connections or link the Hamachi connections to the social network accounts that are used to bootstrap the F2F network.
- The free version of Hamachi allows only up to five peers in a network. The F2F framework is free software and aims to support clouds with hundreds of peers.
- The service depends on Hamachi central servers. It is not possible to host the central server in your own environment.
- The connections and interfaces are exclusively managed by Hamachi. There is no API and mixing the Hamachi connections with other types of connections would require specific support from the operating system (bridging the tap devices etc).
- Hamachi is closed source and details of how it sets up the mesh network (among other things) are secret.

2.4.2 OpenVPN

OpenVPN[19] is one of the most popular open source virtual private network (VPN) solutions. It has good cross-platform support and a mature security model. OpenVPN uses the client-server model which allows several clients to access the subnets attached to the OpenVPN server and optionally also provides connectivity between the peers.

OpenVPN would be better suited for use in the F2F framework than Hamachi because it is open source, has no restrictions on the connection count and provides a public plugin API which could be used to integrate it closely with the F2F framework. OpenVPN does not support mesh mode, so the necessary network topology would be constructed by running the OpenVPN server on multiple peers.

The main issues preventing the use of OpenVPN with the F2F network are:

- OpenVPN is built to run as a centralized service. The server must have an open port with a public IP address to be able to receive connections. There is no built-in NAT traversal and it would require modifications to the OpenVPN code to get it to support some of the common NAT traversal techniques such as STUN/TURN.
- OpenVPN relies heavily on low level operating system networking capabilities. The connections are made visible using TUN/TAP devices and the packets are routed using the kernel level routing tables. All this configuration requires root access to the host operating system. The F2F framework tries hard to avoid requiring root access.
- While OpenVPN has some limited support for connection redundancy, it doesn't handle dropped connections very well. The configuration enables specifying alternative server addresses so that if one of them fails, the next one can be tried. However there is no easy way to get OpenVPN to reconnect to a connection that failed but was later restored.
- OpenVPN doesn't have a mechanism to deal with loops in the network topology. As the F2F framework tries to establish a mesh of connections between its peers, the network loops will almost certainly occur. This can cause broadcast storms and completely kill the peer-to-peer network. Fixing the issues caused by loops would be error prone and almost certainly require using operating system specific functionality.

2.4.3 IPOP project (socialvpn)

Of the currently existing P2P virtual private network solutions, the IPOP (IP-over-P2P) project[22] is the most similar to the F2F framework. IPOP has a clear focus on tunneling IP packets[21] between peers without using a central server. It uses modern NAT traversal techniques to bypass NATs and firewalls. The connections are negotiated and managed using existing XMPP (Extensible Messaging and Presence Protocol) servers.

The F2F framework is different from IPOP in several ways:

- IPOP is tightly coupled to the XMPP protocol while the F2F framework can use any social network that has a public API for sending instant messages.
- IPOP only deals with the network traffic between peers while the F2F framework aims to provide a complete suite of tools required for setting up a P2P private computing cloud.
- IPOP optimizes for use cases where the IP packets are generated by the host machine. The VDE integration for the F2F framework is designed for use by the virtual

machine that is running the F2F computations (but accessing the VDE network is also possible from the host machine).

Even though the aim of the IPOP project is different from that of the F2F framework, it is still an useful piece of software. In its current form it cannot fulfill all the needs of the F2F framework for reasons similar to OpenVPN: setting up the network interfaces that IPOP uses requires root access to the host machine and it doesn't handle loops out of the box. However unlike OpenVPN, the IPOP code is much more modular.

This creates some interesting possibilities. For example it would be possible to integrate parts of the IPOP code into the F2F framework to provide an alternative way of creating connections between the peers. The connections provided by the IPOP code could then be used in the F2F VDE network the same way the native F2F framework connections are used.

2.5 Introduction to network protocols and algorithms

This chapter introduces the main building blocks of modern networks and tools for building P2P virtual networks. The chapter starts with basic network technologies. Later, more advanced topics such as network topologies and NAT traversal tools are covered.

2.5.1 Brief overview of IP networking

The internet is a massive computer network that works on the packet switching principle[23]. Machines on the network can communicate by sending small discrete packets of information. In many cases the source and destination machine are not directly connected, in which case the network must support routing the packet through intermediary machines to the destination. This section explains the packet structure and routing of IP networks running on Ethernet (and WiFi) links.

IP (Internet Protocol, OSI layer 3) is a the main protocol that is used to identify machines connected to the internet and to address packets that are sent between these machines. Each machine is assigned an unique address from the address space (32bits in case of IPv4). The address space is split into contiguous blocks called subnets and machines in the same physical location are often grouped into the same subnet (assigned addresses from the same subnet).

Each packet using the IP addressing scheme contains the IP packet header. The header contains information required to efficiently route the packet to its destination, including the source IP address and the destination IP address.

The IP addresses in the packets alone are not sufficient to successfully route a packet to its destination. A machine can have multiple network interfaces, each of which is connected to a different subnet of the network. It would not make sense to forward the packets through the interfaces that do not lead to the destination. To solve this issue, each machine maintains a separate routing table to help with the routing decisions[15].

A routing table is a lookup table that has ranges of IP addresses associated with names of the network interfaces on that machine. For addresses that are directly connected to the machine, only the interface name is associated with the address range. For ranges that are known to be reachable (directly or indirectly) through another machine, the range is associated to the IP address of that concrete router machine.

In the most simple use case the machine has only a single subnet directly connected to it in which case its routing table contains only two entries: one entry for the directly connected network and another entry that works as a ”catch-all” for everything else. The catch-all part is sent to some other machine on the network that is connected to the other segments of the network (or the internet). This is called the default gateway. The gateway must be directly connected so that packets can be sent to it using the first routing table entry. It is also possible that the gateway doesn't exist - in this case, only the directly connected machines are reachable.

2.5.2 Network links and topology

The most popular technology for physically connecting devices on home and office networks is the Ethernet[31] protocol. Ethernet is the low level protocol which encapsulates each packet moving over the physical wire and enables the network interfaces to ”talk” to each other.

IP is not the one and only protocol that is sent over Ethernet networks. In fact, two versions of it are in active use (IPv4 and IPv6) in addition to completely different protocols such as ARP (address resolution protocol), IPX etc. It would be impractical for the network hardware to implement parsing for all the current and possible future protocols just to be able to determine the packets destination. For this reason, the packets going over Ethernet networks use another addressing scheme for link layer communications (sending packets between directly connected devices).

The link layer (OSI layer 2) addresses are called MAC (media access control) addresses. Each network interface has a unique MAC address that is usually assigned to it by the manufacturer. All data sent over an Ethernet network is wrapped in Ethernet frames that start with the source and destination MAC addresses. The rest of the frame contains the payload which can be a packet of one of the higher level protocols such as IP. This way the software of Ethernet network interfaces must only ”speak” Ethernet frames and don't need to be modified if a new higher level protocol is invented.

The examples so far have been focusing on direct connections between the addressable machines. In most office or home networks there are many machines that are communicating with both each other and the internet. Running dedicated cables between each pair of machines would be very costly. A more efficient solution would be sharing the cables between multiple machines.

Hubs were invented to solve the issue of directly connecting multiple devices without requiring several network interfaces on each of them. A hub is a device with a large number of ports. Each device in the network has only a single physical connection to the hub, which then replicaates all the incoming Ethernet frames to the other connected devices. Hubs can be connected to each other which allows building complex network topologies. The devices connected via hubs can exchange packets directly without going through the full network stack of the gateway device or some other intermediary.

The issue with hubs is that the Ethernet frames are replicated to all ports, even the ones that don't lead to the destination machine. The unnecessary transmissions and the resulting collisions severely limit[32] the throughput of the network. Each machine must compete with all others for the network resources because the ”send channel” is shared. This fact makes it much slower than connecting all devices with separate links so that everyone can send at the same time. This was solved by inventing switches - a special kind of hubs that inspect Ethernet frames going through the ports and recording the source MAC addresses. The generated lookup table of MAC address to ports allows the switch to send the frames only to the relevant ports, greatly reducing the unnecessary transmissions.

Chaining switches allows different network topologies to be formed. Most of the office networks are built as a tree topology (Figure 3). The tree is easy to set up and provides connectivity between all the machines. Machines that are physically near each other can be attached to the same switch and packets for nearby devices rarely have to travel long distances back and forth to a central switch or router. Also, the length of the cables (and thus the cost) can be minimized. Another popular topology is the star, where all switches connect to one central switch. This is the most simple topology which also limits the number of intermediary switches between two communicating devices. As a downside, the total length of the cables in longer because each switch must connect directly to the center. Also the central switch must be able to handle a heavy load as most packets will be passing through it.

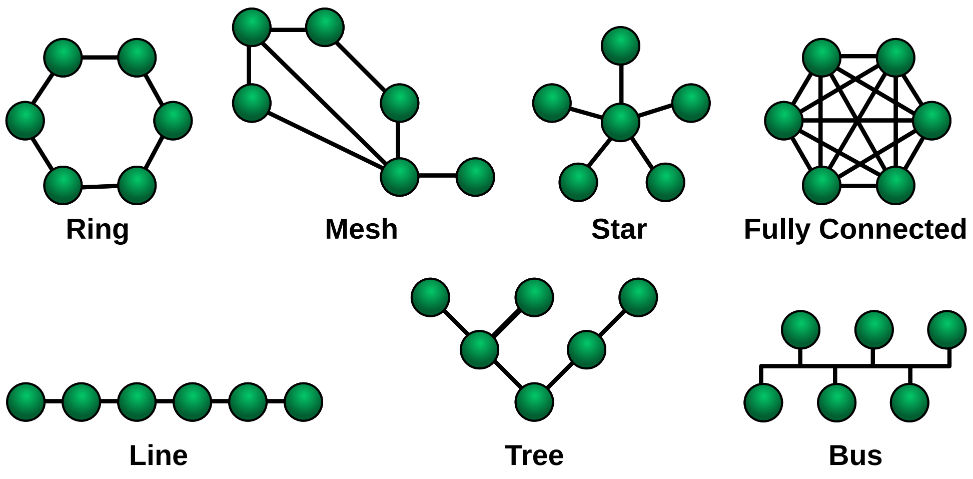

Figure 3: Different switch topologies[11]

The tree-like topologies do not contain loops or backup links. It is also possible to build topologies with loops such as rings or fully connected meshes. In this case special considerations must be taken into account. While most of the packets are unicast packets with concrete destination machines, some packets are broadcast over the entire network. To make a broadcast, the source machine must use a special MAC address as the destination address when sending out the packet. Each machine that receives the broadcast packet (including switches) will then forward the packet out of all its interfaces, except where the packet was received from. This will not work too well in a topology with loops because the packets will keep looping around infinitely and use up all the resources. Even so, protocols exist that can enable use of backup links between the switches.

2.5.3 Network address translation

One of the IP addressing scheme's strong points is that it can be used to create end-to-end connections between the connected machines. Both the source and destination are addressable which enables routing packets directly from the source to the destination using the endpoints' actual IP addresses. These direct connections are also the foundation of all peer to peer networks. Unfortunately IPv4 addresses have a length of 32 bits which can represent only 4 294 967 296 unique addresses (not all of which are usable).

The shortage of IPv4 addresses has forced the wide scale adoption of techniques called network address translation (NAT, RFC 2663) and private network addresses (RFC 1918). Private network addresses are blocks of IPv4 addresses that are not routed (by policy) on the public internet. The private addresses are often used in local area networks in homes and offices that don't need to be reachable from any machine on the internet. This allows the private address blocks to be reused in different networks, which alleviates the shortage of IPv4 addresses.

The machines that are using the private IP addresses cannot directly communicate with other machines on the internet because the responses cannot be routed back to them. NAT can be used to overcome this restriction. The gateway of the private network must be assigned a public IP address. The packets from the machines in the private network reaching the gateway are then modified by the gateway so that all packets seem to be originating from the gateway. This process is called IP masquerading. Any packets going out of the private network have their source IP address changed to the public IP address of the gateway. Any response packets coming into the gateway have their destination IP changed to the original machine's private IP address. As a result, only a single public IP address is required per private network.

While NAT works great for many common use cases (using services where the client initiates connections to access a server) it severely restricts the way packets can be sent. One significant change compared to a NAT-free network is that end-to-end connections are no longer possible, because not all machines are directly addressable. Another difference is that not all IP protocols can be used. NAT usually only works with TCP and UDP protocols (OSI layer 4), because the port numbers used in these protocols make it easy for the gateway to match the response packets to the source machine. Protocols without port numbers cannot be used at all without special configuration in the gateway.

The restrictions imposed by NAT make it practically impossible to create connections between two machines in different networks that are both masqueraded. Regardless, direct connections are required to be able to build many useful applications and protocols, such as GRE (Generic Routing Encapsulation), VoIP (Voice over IP), videoconferencing applications, peer-to-peer file sharing and also the F2F framework.

Over time, different protocols and techniques have been invented to work around the restrictions of NAT or remove the underlying cause altogether. In the long run the best fix would be to start using IPv6 which would solve the IP address shortage. IPv6 has been implemented in all major operating system since 2011 but the internet service providers have been reluctant to upgrade their infrastructure to support it. This is slowly starting to change but at the time of writing this thesis, only 5 countries are showing any significant (over 10% of requests) IPv6 traffic[1] according to Akamai content distribution network.

Other solutions for bypassing NAT work by asking or tricking the gateway into forwarding the necessary ports to the masqueraded machine. Protocols like UPnP (Universial Plug and Play) and NAT-PMP (NAT Port Mapping Protocol) can be used to send a request to the gateway to forward some ports to the requesting machine. The gateway must support the protocols and be willing to forward the requested ports. This works reasonably well in small home networks where the user has control over the gateway. At the same time other environments like office networks often disable UPnP and NAT-PMP for security reasons.

Finally, more "forceful" solutions have been invented to bypass the gateways that do not support port mapping. STUN (Session Traversal Utilities for NAT) uses the property of NATs that creating an outgoing connection from the masqueraded machine automatically allocates a port mapping to allow responses back to the outgoing packets. The STUN protocol can be used to connect to an external server and learn the gateway's public IP address and the port that was allocated for the STUN query. The same address and port can then be used for connecting to the masqueraded machine. Some NAT implementations are more strict and only allow incoming packets from the IP address where the "port opening" packet was sent to. In this case, STUN is useless and an actual relay must be used. TURN (Traversal Using Relays around NAT) is a popular protocol for setting up relay connections to bypass NAT. As a downside, the extra hop increases the latency of the connection and the bandwidth costs for the relay may get expensive (especially if it used for file sharing).

The F2F framework is essentially a peer-to-peer network. The networking layer of the F2F framework must be flexible enough to support all the NAT traversal solutions described in this section. Additionally the complexity should be hidden from the higher level components and the framework's users. This is solved by the I/O provider abstraction layer which is described in the later chapters.

3 Rebuilding the F2F network layer to support VDE integration

Adding support for VDE turned out to require significant changes to the existing F2F framework code. At the start of the integration the framework code was mostly a proof of concept with a lot of implementation details missing. During the integration many issues were fixed and a large part of the code was rewritten to accommodate the VDE module. Here we discuss the changes that were implemented during the practical part of the thesis.

3.1 Overview of the pre-VDE network layer

The initial version of the F2F framework code (”old code”, from the f2f-qemu prototype branch) consisted of a mix of python and C code. The C code was responsible for queueing send/receive buffers, managing the list of peers in the computing cloud, group memberships and job scheduling. The python code initialized the entire system and handled the data connections.

The pseudocode in Figure 4 illustrates the input/output main loop in the old code.

```
// F2F C API (with SWIG bindings for python)
char* f2f_get_outgoing();
bool f2f_has_outgoing();
void f2f_handle_incoming(char* message);

// F2F XMPP python API
def xmpp_get_next()
def xmpp_has_next()
def xmpp_send_message(message_to_send)

// F2F python control loop
while running:
  sleep(small_constant)
  while f2f_has_outgoing():
    xmpp_send_message(f2f_get_outgoing())
  while xmpp_has_next()
    f2f_handle_incoming(xmpp_get_next())
```

Figure 4: Main loop of the old code

While the code works and has proven itself extremely useful for testing and as a proof of concept, it has some considerable disadvantages:

- The F2F core has no way of throttling incoming messages and is forced to create an unbounded buffer. This can cause unpredictable spikes in latency and memory usage. Using a queue that is too long can cause packets to arrive late, causing retransmissions for packets that are actually not lost with both duplicates being delivered.

- The F2F components don't know when the connection provider is ready to accept new messages without blocking. Non-blocking I/O is not possible because the connection provider doesn't emit any events when the buffers become empty.
- Low level components - input/output and moving around most of the data - is implemented in python while the high level code - policy and logic - is implemented in C. It would be better to have the low level code components in C instead, so that they can benefit from the speed of native code.
- The control loop contains a sleep() call which kills latency and throughput. However, with the current architecture removing it would cause high CPU usage due to the loop spinning without anything to do.
- The API uses global static state which makes starting multiple instances of the framework impossible. This limits the use of the F2F framework in use cases where a single instance is not enough or the instances must be cleanly separated.
- The control loop is hard coded to use a single connection provider (in this case XMPP). This makes adding new connection providers a pain because the control loop must be modified for each of them at compile time (or write time, in our python case).
- Mixing python and C complicates the build system and can become a portability issue when support for a new platform must be added.

3.2 Design goals for modifying the F2F network layer

Preparation of the VDE integration provided an excellent opportunity to redesign parts of the F2F networking layer and amend some of the issues that were present in the earlier design.

As part of the redesign, control of the input/output was inverted so that the F2F core would be handling the reading and writing of the messages. This fixes the unbounded queue issue described above and gives the F2F packet assembly component better control over the incoming data. As a result, the number of times the incoming data has to be copied around is reduced and the logic is simplified.

Since the F2F framework is using different connection types, a common abstraction had to be created to hide the differences between the connection types. The idea is that the F2F core should not have to know the internal implementation details of the connection as it only needs to send and receive data through it. The abstraction was already introduced in Chapter 2.3 under the name of I/O providers. In this chapter we discuss the details of the abstraction and how it enables the runtime selection of the I/O provider.

The design changes also aim to create a more intuitive API for both the I/O provider implementers as well as F2F framework's API users. In the new design, the relevant API is focused into cleanly separated header files with minimal dependencies on the internal details of the F2F framework. The API methods are thread-safe and non-blocking, where possible, to avoid race conditions and random hangs in case of network issues.

Finally, the implementation was updated to have an internal event dispatch mechanism to notify the framework components and also any interested framework API users of the events happening in the network layer (new connections, availability of new packets etc).

3.3 Implementing event based input/output abstractions in F2F

In the center of all the data going through the F2F framework are the I/O providers. The F2F cloud is aiming to be scalable up to several hundred peers and moving data between the peers must be as efficient as possible. While the total bandwidth of each peer is expected to be well under 100Mbps for most use cases, the number of connections is going to increase as peers join the F2F cloud.

3.3.1 Issues with blocking input/output

To illustrate the issue of growing number of active connections, two common connection handling patterns will be compared: blocking I/O with thread-per-stream (used in the old code) and non-blocking I/O with multiple streams per thread (used in the improved code).

In the first pattern, reading and writing from the streams (both input and output) is blocking, meaning that the write and receive methods on the stream will not return until at least some data is written or received. Because no peer is guaranteed to send any data, the thread waiting on the receive method can be indefinitely blocked and be unable to do any other work. The same issue can manifest with the write method if the connection is slow or the receive buffer is full on the remote end.

Due to the unpredictable blocking of this pattern, most programs utilizing it will allocate a dedicated thread for each direction for each stream. This enables the main thread to continue working even while data is being transferred over the network. The cost of the pattern is that the number of threads allocated equals the number of remote hosts times two.

In most cases the blocking I/O pattern works very well and is somewhat easier to understand. The threads that are blocked are not using any CPU time and modern operating systems are capable of handling a large number of threads efficiently. Issues arise when the number of connections starts growing. In case of a F2F cloud with 50 peers, the blocking pattern would have to allocate about 100 threads in each peer just for the connections.

If most of these connections are also actively transferring data then the thread context switching costs[27] can start to show up. The context switches hurt the most on less powerful systems (for example Android systems with a single core CPU). Even so, the context switch cost would not affect the performance significantly - it would only increase CPU utilization (and with that, the power consumption).

The bigger issue with large number of threads is memory usage. While creating additional threads is a lot cheaper than creating new processes[16] because the threads can share the open resources (file handles, loaded libraries etc), there is still some per-thread overhead. In particular, each new thread must allocate its stack space for the running function calls. In a modern linux system, the default stack size for a thread is 8192K[7]. In the F2F case, the memory usage of the blocking model can easily grow into hundreds of megabytes as the number of connections increases.

Configuring the stack size to be smaller is possible[29] but the exact size of the required stack size is not known ahead of time. Setting the stack size to insufficient size will lead to a stack overflow followed by a segmentation fault crash. Even with a perfectly optimized threads the memory usage would still stay unreasonably high and would be highly dependent on the number of connections. For these reasons it was decided against using blocking I/O pattern in the F2F network layer.

3.3.2 Using non-blocking input/output

The disadvantages of blocking I/O have been known for a long time and several solutions exist to provide a more scalable model. The main features of non blocking input/output are:

- API methods will never block. If data is available for reading, then the method returns it without delay. If the buffer is empty, then no data is returned and the status will be indicated with the corresponding return value. The same applies for writing data. In particular, a call to write is not guaranteed to write the entire buffer passed to the method if the network socket's output buffer cannot fit it. The return value will indicate how many bytes were actually written, if any.
- The operating system emits events for socket state changes. A low level API is provided that can be used to receive events from the operating system. The events are emitted when a socket becomes available for reading or writing, an error occurs or a new connection is ready to be accepted from a listening server socket. This makes it possible to efficiently use the non blocking API without constantly polling each socket to find out if it is ready to be processed.
- A single thread can poll for events from multiple sockets at the same time. This removes the need to create a separate thread for each socket and the need to loop over the sockets separately. A single thread will choose the sockets it is interested in receiving events for and call a special selector method. The selector method can be configured to either block until any socket becomes ready for processing or until a timeout expires.

In practice there are several APIs available for using non blocking input/output in different operating systems including some cross platform libraries for socket I/O. In the following section some examples of these libraries and APIs are given to introduce the concepts used in later chapters.

3.3.3 Non-blocking example: libevent

Libevent is a popular lightweight library[18] by Nick Mathewson and Niels Provos. It is used by many programs including the Chromium browser and memcached. The libevent project describes itself as follows:

> The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also support callbacks due to signals or regular timeouts.

The code snippet in Figure 5 (based on tutorial[12]) shows how one could wait for a pair of sockets to become available for reading or writing.

```
void cb_readable(evutil_socket_t fd, short what, void *user) {
  printf("socket %d is now ready to read\n", (int) fd);
}

void cb_writable(evutil_socket_t fd, short what, void *user) {
  printf("socket %d is now ready to write\n", (int) fd);
}

void main_loop(evutil_socket_t fd1, evutil_socket_t fd2) {
  struct event_base *base = event_base_new();

  struct event *ev1, *ev2;
  ev1 = event_new(base, fd1, EV_READ|EV_PERSIST, cb_readable, NULL);
  ev2 = event_new(base, fd2, EV_WRITE|EV_PERSIST, cb_writable, NULL);

  event_add(ev1, NULL);
  event_add(ev2, NULL);

  event_base_dispatch(base);
}
```

Figure 5: libevent code example

The methods `cb_readable` and `cb_writable` contain user code that will be executed by libevent once it determines that a socket is ready to read or write some data. Libevent will also pass some useful arguments to the callback methods: the socket on which the event happened, a bit field that contains information about what happened with the socket and also a user defined argument (not used in this example).

The method `main_loop` sets up libevent to watch for events on the two sockets passed in as parameters. First, an `event_base` object is created that acts as a "registry" of sockets that libevent is supposed to watch. Next, we create an event object for both sockets. Each event contains the socket that shall be watched for events, a callback method that should be executed when an interesting event is detected and a bitmask that specifies the conditions on which the callback should be executed. In the example, `cb_readable` will be called when data becomes available to read from fd1 and `cb_writable` will be called when fd2 is ready to write some data. Next, both events ev1 and ev2 are marked as pending (ready to act on socket state changes) by calling `event_add`. Finally, `event_base_dispatch` is called which starts the libevent main loop.

3.3.4 Non-blocking example: select

Another API for non-blocking input/output is the *select* function. It is a rather old API available both on windows (the *winsock2* API [13]) and UNIX (POSIX *select*[30]) with minor differences. The API is simpler than libevent but it has some clunky setup code that must be run before each invocation. The performance of *select* is comparable with that of libevent. In fact, libevent can use *select* internally (depending on the platform). For the F2F framework, *select* provides all the functionality that is required to implement the network layer efficiently.

The code snippet in Figure 6 demonstrates the use of the *select* function and implements the same functionality as the libevent sample did. The code is based on a tutorial from [14].

```
void cb_readable(SOCKET s, void *user) {
  printf("socket is now ready to read\n");
}

void cb_writable(SOCKET s, void *user) {
  printf("socket is now ready to write\n");
}

void main_loop(SOCKET s1, SOCKET s2, void *user) {
  fd_set readfds, writefds;
  while (running) {
    FD_ZERO(&readfds);
    FD_SET(s1, &readfds);

    FD_ZERO(&writefds);
    FD_SET(s2, &writefds);

    if (select(0, &readfds, &writefds, NULL, NULL) > 0) {
      if (FD_ISSET(s1, &readfds))
        cb_readable(s1, user);
      if (FD_ISSET(s2, &writefds))
        cb_writable(s2, user);
    }
  }
}
```

Figure 6: winsock select code example

In the sample code, methods `cb_readable` and `cb_writable` work the same way as they did in libevent sample. The only difference is the function parameters. Because we are calling the callback functions manually, we can pass in any arguments we need. In case of libevent the function signature was fixed and only a single user parameter was allowed.

The main loop with *select* is somewhat more verbose than libevent but it is easier to understand because control is not passed to an external dispatch loop.

The method starts by declaring two sets to hold the sockets that must be monitored - the first set contains the sockets we are interested in reading from and the other set contains the sockets we are interested in writing to. The loop starts by clearing the sets and adding the relevant sockets to the sets. Next, the *select* function is called. The function will wait until any of the sockets in any of the sets becomes ready for processing (or until a specified timeout is elapsed, in this case NULL). Before returning, the *select* clears all the socket sets that were passed to it and uses the same sets to return the sockets that are ready to be processed (that is the reason the sets must be reinitialized at the beginning of each loop).

Finally the caller of *select* must check the socket sets for sockets that are ready to be processed and decide what to do next. Usually this is where the reading and writing is done.

3.3.5 I/O provider API in F2F

This section discusses the implementation details of the new F2F I/O provider API. The I/O provider API has two components: the comm abstraction and the I/O notification API.

The comm abstraction consists of a comm data structure and the related helper methods. As explained in the earlier chapters, the objective of the comm abstraction is to hide the implementation details of different connection types from the F2F network layer users. Users of the network layer (for example the packet assembly component) should not have any source code dependencies on different connection implementations such as TCP or XMPP. Only minimal required functionality should be exposed to the comm user.

The abstraction works by using advanced C programming techniques such as opaque pointers and using function pointers to hide[34] the implementation details. The comm data structure contains a function pointer for each method that the abstraction provides ("virtual methods"): reading bytes, writing bytes, closing the connection and enabling/disabling events. Additionally an opaque pointer to the connection specific data structure is held.

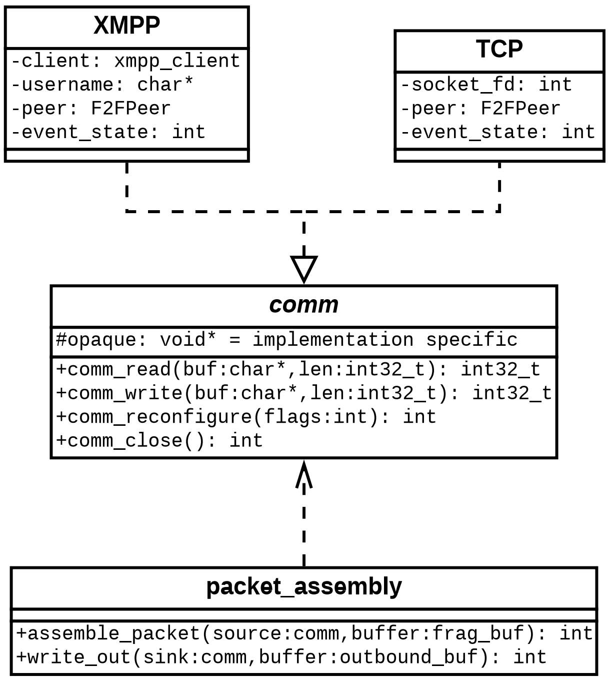
XMPP
-client: xmpp_client
-username: char*
-peer: F2FPeer
-event_state: int
TCP
-socket_fd: int
-peer: F2FPeer
-event_state: int
comm
#opaque: void* = implementation specific
+comm_read(buf:char*,len:int32_t): int32_t
+comm_write(buf:char*,len:int32_t): int32_t
+comm_reconfigure(flags:int): int
+comm_close(): int
packet_assembly
+assemble_packet(source:comm,buffer:frag_buf): int
+write_out(sink:comm,buffer:outbound_buf): int

Figure 7: Dependencies in the comm abstraction

The comm data structure is initialized by the connection type specific code ("connection implementation"). The connection implementation must implement each of the virtual methods. When a new connection is established by the connection implementation, it must initialize a new comm data structure and assign the function pointers in the comm to its implementation of the virtual methods. For example, an XMPP connection implementation would point `comm.read_fn` (Figure 7) to its `xmpp_read` function. If the connection implementation doesn't map cleanly to the comm abstraction then it is the connection implementation's responsibility to implement the required adapter code so that the comm abstraction can stay simple and connection neutral.

The connection implementation will likely need to hold some state during its lifetime. It cannot add its data to the comm structure because that is shared between all the connection implementations. Instead, the comm abstraction reserves an opaque pointer for the connection implementation that can point to a connection implementation specific data structure that holds the necessary state.

An opaque pointer is simply a pointer that has its type hidden. It is useful for hiding implementation details and ensuring encapsulation - using an opaque pointer guarantees that the connection implementation's internal state isn't accessed from an alien method and also avoids the compile time dependency on the internals of the connection implementation. The connection implementation is free to manage the opaque pointer and add/remove fields from the encapsulated data structure without forcing the recompilation of rest of the F2F code.

Finally the comm abstraction provides a set of helper functions for using the virtual methods. Each helper function checks that the comm hasn't been closed and calls the corresponding virtual method using the correct function pointer and the correct opaque pointer argument. The helper functions do not contain any complicated logic by themselves and are meant for convenient usage by the F2F comm abstraction users.

The main declaration of the comm abstraction can be seen from Figure 8.

```
struct comm {
    int is_closed;
    void *opaque;
    int32_t (*read_fn)(void *opaque, char *buf, int32_t len);
    int32_t (*write_fn)(void *opaque, char *buf, int32_t len);
    int (*reconfigure_fn)(void *opaque, int flags);
    int (*close_fn)(void *opaque);
};

int32_t comm_read(struct comm *comm, char *buf, int32_t len);
int32_t comm_write(struct comm *comm, char *buf, int32_t len);
int comm_reconfigure(struct comm *comm, int flags);
int comm_close(struct comm *comm);
```

Figure 8: comm/comm.h

The virtual methods in the comm abstraction are specified as follows:

- `comm_read` is used to read bytes from the connection. The arguments specify a read buffer where the read bytes should be copied to and the maximum number of bytes the read buffer can accept. The method is non-blocking - it will return immediately if there are no bytes to read. The return value contain a non-negative number with the number of bytes read or a negative error code.

- `comm_write` is used to write bytes to the connection. The arguments specify a pointer to the byte array that should be written and the number of bytes in the pointed array. The method is non-blocking - it will return immediately if the connection's output buffer is already full. The method will return the number of bytes written from the passed buffer or a negative status code if an error occurs.

- `comm_reconfigure` is used to notify the connection implementation when the F2F framework is ready to read or write bytes to the connection. This configuration controls when and how the connection implementation calls the F2F I/O notification API (explained in the next section).

- `comm_close` requests the connection provider to stop forwarding data and clean up any resources it may have allocated.

The second part of the I/O providers is the addition of I/O notification API. This is a set of function in the F2F network layer that are called by the connection implementations. The functions are used to notify the F2F framework of different events happening with the connections linked to its peers.

The minimal set of notification functions the connection implementation is required to support is shown in Figure 9.

```
int f2f_handle_comm_io(struct F2FPeer *peer);
int f2f_handle_comm_connecting(struct F2FPeer *peer);
int f2f_handle_comm_disconnected(struct F2FPeer *peer);
int f2f_handle_comm_incoming(struct F2FCore *core,
                             struct comm **comm_out,
                             struct F2FPeer **peer_out);
```

Figure 9: f2f_io_backend.h

The functions are specified as follows:

- `comm_io` is used to notify F2F when the connection is ready to read or write data

- `comm_connecting` is used to notify F2F when a connection has been established with a peer

- `comm_disconnected` is used to notify F2F when a peer has disconnected or the connection has failed

- `comm_incoming` is used when a new incoming connection is detected. The method lets the F2F core allocate resources for the new peer and run any other preparations before the peer is marked as connected.

The notification API isn't directly used to move any data but it is essential for efficient use of the comm abstraction's non-blocking methods. The F2F network layer relies on the notification API to know when a read or write should be attempted on a peer and as a result doesn't have to poll all the peers in a loop or allocate dedicated threads to process each peer's connections. The notifications are also used to run cleanup for the peers that have been disconnected without having to run a cleanup loop to find stale resources.

The connection implementation's use of the notification API can be configured using the `comm_reconfigure` virtual method. This mainly affects the triggering of `comm_io` notifications. The F2F network layer maintains its own send and receive buffers for each peer. When the read buffer is full, then it isn't interested in information about the connection being readable. Likewise, when the write buffer is empty, then it isn't interested about the connection being ready to write data. The reconfigure method is used to simply disable or enable the notifications to match the state of the F2F internal buffers.

3.3.6 TCP reference implementation in F2F

This section describes a sample I/O provider implementation. The TCP I/O provider now included in the F2F framework is useful for benchmarking and regression testing. Due to its simplicity it's also suitable for use in tutorials on using the comm and notification API as well as for example code.

The TCP provider consists of two components - the connection group (tcp_group) and the connections (tcp_conn). Each tcp_group can have multiple tcp_conns as its members. To use the TCP provider, the user must first create a tcp_group structure. When creating or accepting new connections, the connections must be associated with one of the groups. Closing (freeing) a group also closes all the member connections.

The tcp_group is responsible for maintaining the resources that are shared between the connections. This includes the synchronization primitives, the list of member connections and a shared worker thread for handling the input/output events on the member connections.

The tcp_conn structure groups all the necessary information for handling the input/event on that event: the associated peer, tcp_group, configuration for invoking the notification API and the actual TCP socket handle. A pointer to tcp_conn is stored in both the tcp_group and in the opaque pointer of the comm abstraction.

In the center of the TCP provider is the tcp_group's worker thread and its event loop. The loop starts by fetching the notification API configuration (whether to look for read or write readiness) from each of its members and then waits for a one or more socket to become ready. If one of the sockets has failed then the corresponding tcp_conn is closed. For each failed or ready connection, the corresponding notification API functions are called.

The worker thread uses the POSIX *poll* system call to wait for a socket to become ready. The *poll* syscall is almost identical to the *select* syscall (covered in Section 3.3.4), but the setup code is a little shorter. Both *poll* and *select* are blocking syscalls. To support changing the connections' configuration (by calling `comm_reconfigure`) while the poll is blocked, the "self pipe trick"[3] is used. This adds a dummy socket to the list of polled sockets and marks it readable when the syscall needs to be "interrupted". The tcp_conn configuration can then be refreshed.

3.4 Implementing packet assembly and multiplexing in F2F

The F2F framework peers communicate with each other using discrete packets of data (as opposed to streams). As part of the redesign the F2F packet format was simplified and the packet assembly component was rewritten to support the non-blocking APIs. The packets of data ("F2F packets") consist of a header containing the packet type and length, followed by the data itself. The format of the header is fixed: the first octet contains an unsigned 8bit integer denoting the type of the packet. The next two octets form an unsigned 16bit integer denoting the length of the packet (excluding the header). The length bytes are transmitted in network byte order.

3.4.1 Packet multiplexing

The packets are used to support multiplexing data from different components of the F2F framework so that data from different components can be sent over a single connection. When a component needs to send data to another peer then it can schedule a packet to be sent via the network layer. The network layer maintains only a single connection between a pair of peers, because in many cases it is not even possible to establish multiple connections. For example if data is transmitted over an instant messenger channel, the instant messenger protocol might not support multiple "channels". Thus the existing connection must be shared between all components. Without the packets, the F2F network layer would be unable to distinguish the start and end of communication between the different components and would not be able to dispatch the incoming data to the correct component.

Sharing the connection between different components is brings about a new set of problems. Suppose the client code needs to send a big file transfer to another peer which takes several minutes to complete. If the F2F framework needs to send command messages to the peer at the same time (for example send network control messages for the VDE component) the command messages could be delayed for a long time and cause confusing timeouts in other components. Creating an "out of band" communication channel for these messages might not be possible for reasons explained above, so a solution had to be found to use the shared connection efficiently.

The packet assembly component was changed to impose two restrictions to fight the long delay issue. First, the packets are restricted to blocks of maximum of 64K bytes. This enables the network layer to interleave the outgoing packets so several F2F components can send data at the same time with relatively small delays. This still leaves the danger of a low priority or bulk data transmission overwhelming the connection by filling the outgoing queue with its packets. To avoid that, the packet assembly component does not queue more than a single outgoing packet and uses a simple priority system for the components so that the high priority data can always be sent at the earliest opportunity. As a result the single connection can support a wide range of use cases and keep the latency near the theoretical minimum (interrupting the bulk transfers would be the only thing left to do to reduce latency, but this would be difficult and impractical for all but the most demanding use cases).

3.4.2 The packet API

The new packet API (Figure 10) provided to the higher level F2F components and the F2F client code follows the same principles as the I/O provider API.

- The API functions are thread safe. Any function can be called from any thread without using external synchronization.
- The API is designed so that the functions are self sufficient - the parameters of the function contain all information to complete the task.

- The API is safe to use - no function leaves the packet assembly component in a unsafe or incomplete state (for example, lock some internal state with one function call and require another function to be called to release the lock).
- The functions are non-blocking. The functions only block for the bare minimum of keeping the packet assembly internal state consistent. The caller does not have to wait for input/output to complete before the function returns. Calling the read function will return data only if the entire packet is ready. Otherwise a status code is returned to indicate that the packet is incomplete. The write function will queue the data for sending out if the packet assembly buffer is empty. Otherwise the function signals the caller to try again later.

As with the I/O providers, non-blocking functions are inefficient if the caller does not know when it's appropriate to call them. Chapter 3.5 will cover the event system that addresses this issue.

```
int peer_is_writable(struct F2FPeer *peer);
int peer_is_readable(struct F2FPeer *peer, uint8_t expected_type);

int peer_atomic_read(struct F2FPeer *peer,
                     uint8_t expected_type,
                     char **payload_out,
                     uint16_t *payload_len_out);

int peer_atomic_write(struct F2FPeer *peer,
                      uint8_t type,
                      char *payload,
                      uint16_t payload_len,
                      int free_after_send);
```

Figure 10: f2f_packets.h

3.4.3 Event based packet assembly/disassembly

Packet assembly deals with assembling incoming bytes into complete F2F packets. Packet disassembly deals with pushing data out over the wire in the correct format. The code must be reasonably optimized since all the data moving through the F2F network layer passes through it. Additionally, the packet assembly component has to be properly synchronized for use by multiple parallel thread.

The main complexity of the component is caused by the number of states (Figure 11) and transitions the code must support. All the methods used in the state machine are non-blocking and the entry points are called from multiple threads and in undetermined order. The state machine must properly respond to events from the I/O provider notification API and at the same time generate its own events for the users of the packet assembly API.

The packet assembly process begins as soon as the I/O provider signals the packet assembly component (using the notification API function f2f_handle_comm_io) indicating that some data is available to read from the connection.

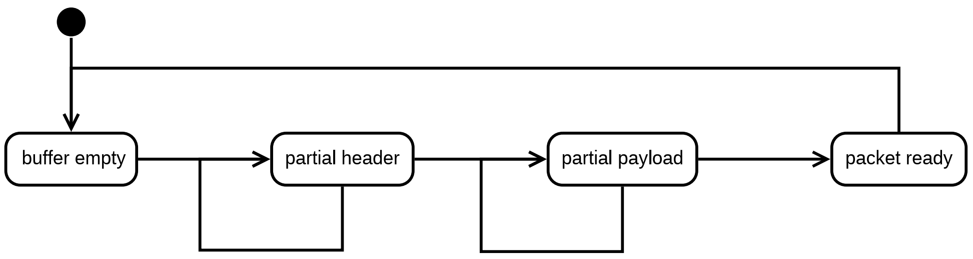

Figure 11: Packet assembly buffer states

The process begin with locking the inbound buffer. Next, the code checks the packet header state. The packet header is read into a separate buffer that can fit only the header. If the header is incomplete, then reading into the header buffer is attempted until the header is complete or the I/O provider returns a status code indicating that no bytes are available without blocking. If the header was not completed then the buffer is unlocked and the assembly process is stopped until the next signal from the I/O provider. If the header was completed, then it is parsed to discover the packet payload size.

The memory for the packet payload is allocated only after the payload size has been determined. Thus the payload buffer is separate from the header buffer. Next, the assembly code tries to read the payload similarly to the header. If the payload cannot be completed then the lock is released and the process is stopped. Otherwise, if the payload is completed. The buffer is unlocked and the packet can be dispatched to an appropriate component based on the packet type.

The choice to read the header and payload into separate buffers has several benefits. First, as the header has a fixed length it doesn't need to be dynamically allocated. This saves some memory allocation overhead. Second, collecting the payload into a dedicated buffer allows the buffer to be directly passed on to the component that claims the packet. The API is designed so that no unnecessary copies are made. For example, the only copying with the TCP I/O provider happens between the kernel level socket buffer and the payload buffer. The same strategy also applies when writing out packets to the network.

The final piece in the packet assembly logic is handling the I/O provider events. A situation may occur where a packet has been assembled but not yet claimed by any component. The I/O provider is continuously accepting new data over the network and whenever it has some new data it will signal the packet assembly component to read it. The packet assembly cannot read the new data since its buffer is full and this would cause the I/O provider to keep re-sending the same event. As a result the I/O provider thread would be running in a loop with 100% CPU usage, wasting time and energy.

To solve the issue, all I/O providers must implement a method that can be used to disable or enable the ready events. This is done separately for read and write readiness. The packet assembly component will reconfigure the I/O provider to disable the read events as soon as it finishes assembling a packet. When a packet is claimed and the buffers are clear again, the packet assembly component re-enables the read event for that peer.

The process for sending out packets is very similar but in reverse order. To send out a packet, the user of packet assembly API calls the write function which accepts the destination peer, the payload and the packet type as parameters.

As the first thing the outbound buffer is locked and the code checks if the buffer is empty (meaning it is ready to accept a new packet). If the buffer is already full then the lock is released and the write is rejected immediately. Otherwise the payload and the packet type are stored for a time when the I/O provider is ready to write. The entire payload buffer is claimed by the packet assembly component instead of making a copy of it. After that the lock is released and the packet API function returns.

After the payload has been accepted the packet assembly must wait for a signal from the I/O provider. When it is ready to write, the buffer is locked and writing starts with the header. To avoid unnecessary copying of data, the code avoids building the full packet ahead of time. This would mean allocating an appropriately sized buffer, preparing the header into the beginning of the buffer and then copying the entire payload to the region after the header.

Instead, the header is prepared into a separate fixed size buffer and sent out. Immediately after that the payload is sent out. As with the packet assembly loop, the writes need not happen at the same time. The I/O provider may declare at any time that it cannot accept more data without blocking. The write loop must be able to stop at any point and resume later. The buffer is unlocked as soon as the writing is stopped to allow other users of the API check if the buffer is ready to accept new payloads for writing.

As with the reading, the events from the I/O provider must be configured to work efficiently. When the buffer is empty and no components are queuing new packets to be sent, the I/O provider will start sending write notifications as soon as it has sent out the tail of the last packet. As there is nothing to be sent out, the event would be resent in a busy loop, wasting resources. To avoid that, the write events are disabled as soon as a packet is sent out and the outbound buffer is cleared. Likewise, the events are re-enabled right after a packet is queued to be sent out.

3.5 Implementing event based packet dispatching in F2F

The I/O providers generate low level events when raw data is received or it is ready to be sent out. This is useful for the packet assembly component but not for the other components working on the higher abstraction level. To provide useful events to the higher level components, a new event API is introduced. The new API ("core events") is essentially an implementation of the observer pattern[26]. Any component can implement an event handler method with predefined parameter types and return type and register it to receive events. When a new event becomes available, the emitter of the event loops through the list of event listener callbacks and invokes each of them.

Core event listeners are notified of all important lifecycle events of each peer. Currently the list of events covers all events that are necessary to handle communications efficiently but events can be easily added if need arises. The existing events work as follows:

- PEER_IO_RW is emitted when the packet assembly component is ready to accept a new packet to be sent out to a peer. Receiving the event does not guarantee that the receiver can queue the packet immediately. The packet assembly API can be called from any thread at any time and there is a race condition between an event handler being called after the buffer becomes free and another thread calling write on the packet assembler API.

  The receiver of the RW event can queue a packet to the outbound buffer by using the `peer_atomic_write` function (Figure 10) of the packet assembly API. The function atomically check if the buffer is free and queues the packet if possible. Additionally the `peer_is_writable` can be used to check the buffer state without queuing a packet. The buffer state is checked using appropriate synchronization but the state may change immediately after the function returns (if another thread is waiting to queue a packet at the time of the check).

  PEER_IO_RW is also emitted when a new packet is ready to be claimed from the packet assembly buffer. The packet can be claimed using the `peer_atomic_read` function (Figure 10). The function `peer_is_readable` exists for checking whether a packet is available, but again, the packet may claimed by another thread right after the check returns true to the current thread.

- PEER_IO_CONNECTING is emitted when a new connection to a peer has been successfully established. The event can be used to set up new event listeners or initialize internal data structures.

- PEER_IO_DISCONNECTED is emitted when a connection to a peer has been shut down. This can be used to clear up any pending buffers, free resources and run other cleanup processes.

One noteworthy aspect of the PEER_IO_RW event is the policy on packets that were not handled by any of the event handlers. This may be caused by either the fact that no component in the F2F framework supports the packet type or the handler for that packet type isn't interested in handling the packet. The default behavior is to discard any such packets. While it may sound extreme at first, it seems to be the best solution.

Having a packet wait in the inbound buffer blocks any other packets from being processed. For example, an unwanted or corrupt packet could block the inbound buffer, thus blocking other packets of higher priority or packets that manage the F2F cloud itself. If the packet isn't removed, it might stay there forever.

On the other hand, if the packet assembly component would queue the incoming packets, then other issues could arise. The packet assembly queue would not know how much data will be queued and for how long. The queue could easily grow very large. Since it would be unbounded, denial of service attacks would be possible by sending a lot of data with a packet type that is known to be not handled by the peer, causing an out-of-memory crash.

The default drop policy actually doesn't rule out buffering of packets completely. While the packet assembly component does not support inbound queues, any interested component can claim the interesting packets and queue them internally. That component would be in a much better position to form a reasonable queuing policy.

The actual core event registration works using the API shown in Figure 12.

The core event listeners can be added and removed using the corresponding functions in the API. As the last argument, a context pointer can be passed to the F2F framework. This pointer is stored and passed back to the event handler when an event is emitted. This is a common technique in C programming that is used to avoid the use of static variables - everything that the event handling function needs can be stored in a struct and be passed to the function using the user context pointer.

The broadcast function is used to emit events to all registered event listeners. For example the packet assembly component invokes it when a packet is ready to be claimed. By default the order in which the event listeners are invoked depends on the order of

```
typedef
  int (*peer_listener_fn)(void *ctx, struct F2FPeer *peer, int what);

int peer_listeners_broadcast(struct F2FPeer *peer, int event_type);

int peer_listeners_add(struct F2FCore *core,
                       peer_listener_fn func, void *ctx);

int peer_listeners_remove(struct F2FCore *core,
                          peer_listener_fn func, void *ctx);
```

Figure 12: f2fcore.h

registration. The event handlers that were registered earlier are also called earlier. This is useful for prioritizing some packet types over the others. The handlers for higher priority packets can be registered first, so that they will always have the chance to claim the packet and/or write new packets before other, lower priority handlers, can interrupt.

4 Integrating VDE switching into the F2F framework

This chapter introduces the lower level details of VDE switching, describes the implemented solution and the design decision made during the development of the F2F VDE plugin ("VDE plugin").

4.1 Design goals for the VDE integration

The main goal was to create a F2F framework plugin which would make it possible to integrate the F2F peer-to-peer network with existing virtualization technologies without making changes the their source code. The code running on the virtualization platform (F2F is currently mainly targeting the QEMU virtual machine) should be able to communicate with other peers in the cloud using standard network APIs, optionally without even having knowledge of the F2F framework.

In addition to the main goal, several other requirements were identified during the design phase of the VDE plugin:

- There must be clean separation between the F2F framework and the new VDE plugin. The framework code should not contain any VDE related code that would fit better into the VDE plugin. Furthermore, the implementation should not modify the F2F framework to contain any code that is only useful for the VDE plugin (but not useful for general use).
- Integrating VDE into the framework should not add any new platform specific code to the framework code. The VDE plugin may include platform specific code but that must not add new compile time dependencies to the F2F core code.
- The VDE plugin should use plugin architecture - using the VDE plugin must remain optional. Attaching the plugin to the F2F core should happen at runtime when the client code requests it. Finally, it should be possible to compile the F2F framework code separately from the VDE plugin (compilation of the VDE plugin should be optional). This implies that the F2F framework core should have no compile time dependencies on the new VDE plugin.
- To reduce security risks, the VDE plugin must not require root permissions to run or to set up any of the network connections.
- The VDE plugin should require minimal configuration to be useful. The API of the VDE plugin must be user friendly and easy to use. Unnecessary verbosity must be avoided.

4.2 Using the F2F VDE plugin

The VDE plugin is compiled as a separate library called *libf2fvde*. It depends on the F2F framework library and the Virtualsquare VDE library. Using it in a F2F client application is as simple as linking against the F2F *libf2fvde* and including the vde-api.h header. Only two functions are required (in fact, only two public functions are currently exported in the vde-api.h header as seen in Figure 13) to use all of the functionality provided by the VDE plugin.

```
struct vde_plugin;

int vde_plugin_detach(struct vde_plugin *vde);

int vde_plugin_attach(struct F2FCore *core,
                      char *switch_path,
                      struct vde_plugin **vde_out);
```

Figure 13: vde-api.h

The vde_plugin_attach function initializes all the data structures required by the VDE plugin and registers the plugin to receive the F2F framework core events. The vde_plugin_detach is the counterpart to the attach function - it removes the VDE plugin from the core event listeners and cleans up all the resources. It is recommended to set up a signal handler[9] in the client program so that the detach function would be called even if the application is aborted with SIGTERM or SIGINT (signals that are emitted by the operating system when the machine is shutting down or ctrl+c was pressed in the console).

After attaching the VDE plugin to the F2F framework, a VDE switch is started on the file system path that was passed to the attach function. The path can then be used to configure a virtual machine's network interface. A simple code example of using the VDE plugin in a F2F client application can be seen in Appendix 1.

4.3 Data flow between F2F and the VDE switch

4.3.1 Starting the switch

Virtualsquare VDE consists of several components: the **vde_switch** program, several helper programs, a library for transferring data to the switch (*vdeplug*) and a library for managing a running switch (*vdemgmt*).

It would have been optimal to run the vde_switch in the same process as the F2F framework, but unfortunately this is not supported. Instead, the VDE plugin must run the vde_switch as a separate process. The VDE plugin automatically creates the new child process for the vde_switch. Most of the configuration is passed to the switch process using the program's command line arguments.

Once the child process is started the VDE plugin must wait for it to initialize. It takes a few hundred milliseconds for the switch to be ready. During this time it creates the necessary sockets for its management console and a control socket which can be used to send data to the switch. The VDE plugin sleeps until the control socket is opened by the switch and then proceeds to the next step.

The only thing that cannot be configured at the switch startup using command line arguments is the spanning tree protocol (STP) support. By default vde_switch has STP disabled because it doesn't add any value in simple network topologies. To enable STP, the VDE plugin establishes connection to the switch management console using the vdemgmt library. Next, the VDE plugin uses the management console to send a command to enable STP for all ports.

Finally the process id of the switch is stored for later used and the switch is marked as initialized.

4.3.2 VDE virtual cables

The vde_switch is a virtual Ethernet switch. To send and receive data it uses "virtual cables". A virtual cable is something that connects to the control socket on the vde_switch and handles the actual data transfers. The switch itself only handles the switching, including running the spanning tree algorithm, storing the MAC addresses that are visible from each virtual cable and sending data to the virtual cables.

The VDE plugin works by starting a vde_switch and creating a virtual cable to the switch for each peer. As each peer running the VDE plugin starts a local vde_switch, the result is a mesh of vde switches connected over the F2F data connections. The switches can then use standard Ethernet networking algorithms to route data between the peers.

The VDE plugin uses the *vdeplug* library to create virtual cables for the F2F peers. The *vdeplug* library is quite simple - it only provides functions to read from the switch and to write to the switch (Figure 14).

```
ssize_t vde_recv(VDECONN *conn, void *buf, size_t len, int flags);
ssize_t vde_send(VDECONN *conn, const void *buf, size_t len, int flags);
int vde_datafd(VDECONN *conn);
```

Figure 14: libvdeplug.h

The last function, vde_datafd, can be used for non blocking input/output. It returns the connection to the switch control socket and can be used to check if the switch is ready to read/write the next piece of data.

4.3.3 Switch shutdown

The VDE plugin can be shut down by calling the vde_plugin_detach function. This starts several cleanup activities:

- Each virtual cable created by the F2F process is disconnected from the switch.
- The VDE plugin stops accepting new packets from the F2F framework network layer.
- A SIGTERM signal is sent to the vde_switch child process using the process id stored during the startup.
- Experiments show that the vde_switch process sometimes fails to clean up all the files it created during its lifetime. As the last stage of the shutdown, the VDE plugin scans the switch's working directory and removes any temporary files that the vde_switch failed to remove.

4.4 Using F2F core events in VDE integration

One of the goals of the VDE plugin was to require minimal configuration from the user. To achieve that, the VDE plugin takes full advantage of the F2F core events. At first, core events only supported notifications for new incoming packets and the readiness to write new packets out. That information wouldn't have been sufficient as the user would have needed to manually register new peers with the VDE plugin and manually remove any peers that had disconnected.

To automate the cloud membership changes, two new events were added to the core event list: PEER_IO_CONNECTING and PEER_IO_DISCONNECTED. Both of the events were already mentioned in Chapter 3.5.

The CONNECTING event is used by the VDE plugin to detect new peers and run the necessary initialization:

- a new VDE virtual cable is allocated and connected to the vde_switch process
- the internal data structures are allocated for holding buffers and synchronization primitives
- the non-blocking input/output loop is notified of the new cable

The DISCONNECTED event causes the plugin to clean up the resources allocated for that peer, essentially reversing any initialization done when the peer first connected.

The VDE plugin also makes heavy use of the RW event. As soon as the VDE plugin is notified of a input/output event in a peer, it looks up the virtual cable and buffers related to that peer. The VDE plugin maintains two buffers for each peer: one for data that must be sent to the switch and another for data that must be sent over the network to the peer.

If the packet assembly component has a packet ready to be claimed and the buffer for data going to the switch is empty, then the packet is claimed. If the buffer containing data that must be sent to the peer is not empty and the packet assembly component is ready to accept a new packet for sending, then the buffer is passed to the assembly component.

On the other side of the core events are the virtual cables connected to the vde_switch. The virtual cables work similarly to the I/O providers - the number of connections to the vde_switch increases as the number of connected peers increases. Having a dedicated thread to handle each virtual cable separately would not be practical, so the same non-blocking APIs are used for handling the virtual cables as for the I/O providers.

The handling of the virtual cables works by creating a separate worker thread to monitor all the virtual cables at once. The worker starts by going through all the existing virtual cables and setting them to receive events when the switch is ready to read or write. Next it waits for at least one virtual cable to become ready. The virtual cables that are ready are processed and the worker starts waiting for the next cables to become ready. If a new peer has connected or disconnected at any time, the worker goes through the existing virtual cables again and refreshes the list of connections it is watching, so that the new peers will not be ignored.

Processing a virtual cable works very similarly to the processing a RW core event. First, the buffers are checked for data that must be sent to the switch. If there is data and the worker thread detected that new data can be written to the switch, then a write is attempted. Since it is a non-blocking connection, the code must be able to handle a situation where the entire packet isn't written at once and the write will be resumed later. If the worker thread detected that there is data available to be read from the switch, then it is read and stored into a buffer for later. The next RW core event will try to send it out.

Finally, after updating the buffers (after either a VDE worker event or a core event) the internal state of the VDE plugin is updated. If the buffer for data going into the switch is empty, then the worker thread is configured not to poll events for switch being ready to accept new data. Likewise if the packet assembly is not ready to write out data to the peer, then the worker thread is configured not to poll events for data being available for reading from the switch.

4.5 Performance and quality

The virtual network provided by the VDE plugin will work well with the "out of the box" configuration. This chapter section will cover some of the default settings that the VDE plugin uses for the network and also provides some insight into how the virtual network might be further optimized.

4.5.1 Buffering strategy

The VDE plugin generally follows the same buffering strategy as the F2F network layer does. Incoming and outgoing packets are not queued and the buffers are held to the minimum.

Not only would excessive buffering increase the complexity of the code, it would also interfere with the existing flow control algorithms built into network protocols such as TCP. This phenomenon is known as bufferbloat[5]. Bufferbloat happens when the buffers along the network path become so large that it starts to negatively affect the goodput (number of useful information bits delivered per unit of time) of the network. The large buffers cause high volatility in the latency, which causes high volume of unnecessary retransmissions.

Even so, the data passing through the virtual network passes through several layers of data buffers:

- the operating system socket read buffer
- the read buffer in the VDE plugin
- the outbound buffer in the packet assembly component
- the operating system socket write buffer
- the buffers of the virtual machine

4.5.2 Throughput and latency

This section describes the results of performance measurements done on the final version of the VDE plugin. The tests mainly measure the overhead generated by the F2F framework and the VDE plugin. As a result it is visible that the overhead stays within reasonable limits, the VDE plugin is working correctly and doesn't generate any major latency spikes.

The testing environment uses the TCP reference provider from Section 3.3.6. The network topology consists of two F2F nodes connected over a gigabit Ethernet connection. The tests were also repeated on a 802.11n WiFi network, but the results were very similar to the wired connection, so they are not included here.

The latency of the connection was using the standard ping tool. The test used 200 ping samples in adaptive ping mode. The payload size was set to 1450 bytes which is close to the maximum transmission unit of the network.

```
ping -A -s 1450 -c 200 peer-vde-ip
```

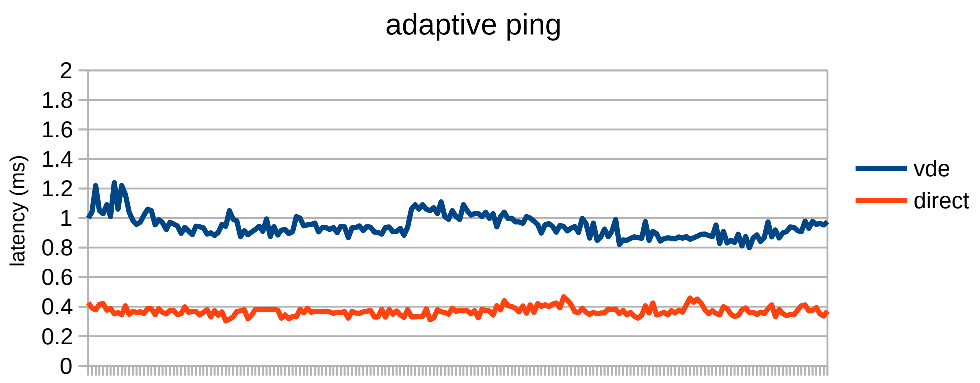

Figure 15: VDE latency

The results are visible in Figure 15. The round trip latency on the gigabit network is quite stable at 1 ± 0.2 milliseconds. The latency using a direct connection between the nodes was at 0.4 ± 0.1 milliseconds. The difference is mostly caused by wrapping the ping packets into the TCP I/O providers connection, which introduces additional round trips for delivering TCP acknowledgement packets. As a reference, the old F2F code had the average latency of 120ms (caused by the inefficient polling loop).

Another issue with the TCP connections is the delays when sending very small packets. Many TCP implementations use the Nagle[17] algorithm. The Nagle algorithm tries to group small TCP packets into a single large packet, which would reduce overhead from the TCP headers. When sending small packets, the Nagle algorithm causes a small extra delay when it is waiting for more data to pack into the outgoing packet. On a modern Linux machine the delay is around 40ms.

For the throughput test the iperf3 tool was used. Iperf is a popular open source tool for measuring TCP bandwidth. It works by starting iperf in server mode on one end and sending data to it from the other end. The server then reports the speed at which the data arrived.

```
iperf3 -i 0.2 -t 10 -f k -c peer-vde-ip
```

In the test, iperf was ran for ten seconds which was divided into 50 periods. The throughput was saved in each period and the results were plotted (see Figure 16) as a line diagram.

The results show that the VDE network manages almost the same throughput as the direct connection between the nodes. The average throughput for the direct connection was 934mbit/s while the VDE throughput was 877mbit/s.

The difference is caused by two factors. First, the TCP connection iperf is using for the benchmark is wrapped in the TCP connection of the F2F framework. This causes some header overhead (F2F TCP header + F2F protocol header + VDE Ethernet header) and interferes with the wrapped TCP connection's built-in congestion control.

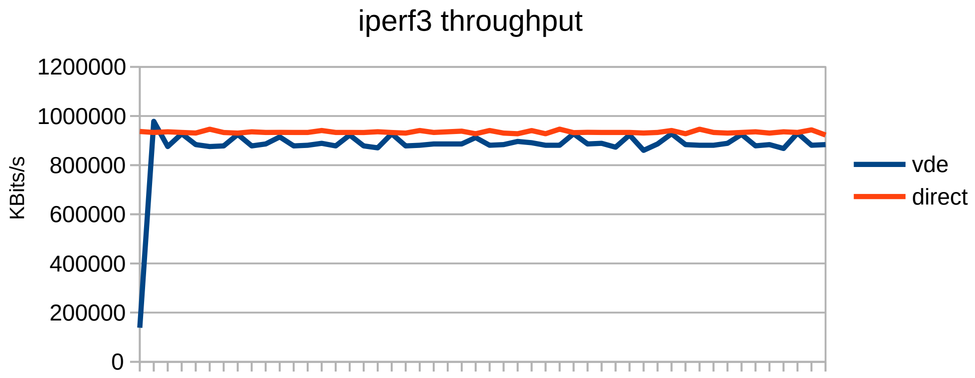

Figure 16: VDE throughput

The other reason is the F2F framework's buffering strategy combined with the TCP connection's expectations of network congestion. For most networks heavy buffering is used, which reduces packet loss and leads to higher throughput at a cost of lower latency. The F2F framework only has light buffering, so in the case of large file transfers the packet loss is the dominant flow control mechanism. The relatively higher packet loss causes TCP to reduce the transmission rate as a normal congestion control[28] response.

4.5.3 Handling topology changes and outages

One of the biggest advantages of using Virtualsquare VDE is that it supports STP (Spanning Tree Protocol) out of the box. STP automatically finds loops in the network topology, designates links as backup links as needed and automatically detects changes in the topology. As a result, the VDE plugin gains some very nice features for free:

- Failed connections are automatically detected and routed around. If a connection fails, the STP algorithm will get a timeout on that link and switch to a backup link. If the connection is later restored, STP may automatically switch back to it.
- Loops in the topology are removed by designating the duplicate routes as backups. This helps to avoid broadcast storms - a broadcast packet being sent around in loops infinitely, wasting bandwidth and delaying other packets.
- The peers will be reachable from other peers as long as there exist enough working connections to form a tree between all the peers. Since each peer tries to establish connections to each other peer, the high number of backup connections will be able to withstand many failed connections.

4.5.4 Quality assurance

The written plugin was tested and verified using different tools. After adding new functionality, several test scenarios were followed to ensure the correct functionality, including testing connection establishment, adding peers to the VDE network, disconnecting peers and running benchmarks on the VDE network.

The network usage was manually verified using the Wireshark network analysis tools. This enabled inspecting individual packets between the host machines and also inside the VDE network.

Finally, the entire software was checked for memory leaks. The Valgrind profiling tools were used for the leak analysis. The checks discovered several leaks inside the F2F framework's code as well as in Virtualsquare VDE code. The F2F framework's memory leaks were successfully fixed.

4.6 Connecting F2F client applications with F2F VDE

Once the vde_switch is running and the peers have been connected, the VDE network is ready for use. There are several options for doing that.

Using the virtual network in QEMU or VirtualBox virtual machine is the most common use case and it is very easy to configure. Newer versions of QEMU have native support for VDE and can be configured[4] by passing the VDE switch path to the virtual machine at startup:

```
qemu -net vde,sock=switch_path f2f-disk-image.img
```

VirtualBox also has built-in support for VDE. The connection cannot be configured using the graphical interface but the official tutorial[33] for VirtualBox configuration describes the commands that can be used to configure a VDE network interface.

For other programs that do not support VDE natively, tools from the Virtualsquare VDE package can be used. A program called vde_plug2tap can be used to create a standard TUN/TAP virtual network interface on the host machine (this requires root permissions). The created tap interface can be used by any program to send packets to the peers in the F2F cloud.

For more advanced use cases, a raw VDE virtual cable can be connected to the switch using a program called vde_plug. This creates a new port on the VDE switch, pipes all packets sent from the switch to the standard output and pipes standard input back into the VDE switch.

5 Conclusions

This thesis presented the results of rebuilding the F2F framework's network layer and the creation of the new VDE integration plugin for the F2F framework.

The process started with analyzing the requirements for the F2F network layer considering the future integration with the VDE technology. A new architecture for the F2F network layer was designed and implemented. The requirements for the VDE plugin were identified and the new VDE plugin was designed and implemented using the new network layer architecture.

The F2F framework's existing network layer was largely rebuilt to increase flexibility and performance. The old thread-per-network-stream model was replaced by using modern non-blocking input/output techniques. The polling based data flow was replaced by a more responsive callback based data flow. A clean application programming interface was added to the framework to simplify using and extending the framework.

A new VDE plugin was developed to enable the F2F framework's integration with different virtual machines. The plugin combines the Virtualsquare VDE with the connections provided by the F2F framework. The events from the F2F framework's new network layer are used to efficiently manage the connections in the Friend-to-Friend virtual network. The initial configuration of the virtual network is automated to require minimal effort from the user. Using VDE allows the framework to create the virtual network without requiring root access, which improves security and simplifies use in high security environments.

The functionality and performance of the developed code was verified using proven diagnostics tools and benchmark utilities. The throughput of the VDE network was measured using the iperf3 benchmark tool. Ping was used to measure the latency of the network. Wireshark was used to debug the code and analyze the performance issues. Finally, Valgrind memory profiling tools were used to detect and fix memory leaks.

As a result of the work done here (available at [2]), the F2F framework has gained the capability for integrating with multiple popular virtual machine solutions. The new integration allows the F2F framework to safely execute distributed applications on the F2F cloud members' machines. Improvements to the F2F framework network layer's interfaces makes it easier to add support for new connection types between the F2F peers. The added virtual network support provides an intuitive way for the F2F distributed applications to communicate, while providing fault recovery, good latency and throughput characteristics.

The improvements made to the F2F framework open up new avenues to future F2F extensions. The IPOP project introduced in Chapter 2.4.3 incorporates many technologies that could be useful in the F2F framework - investigating potential integration with the IPOP code could yield some connectivity improvements for the F2F framework.

The spanning tree protocol used in the VDE network could be made more efficient by automatically assigning priorities to the data links - currently, the topology of the VDE network is somewhat unpredictable and this can lead to non-optimal network topologies in some cases.

The VDE connection performance could be investigated in situations of extreme network congestion and delays. The current solution relies on the retransmission and flow control mechanisms that are built into the standard network protocols, but improvements could be made to the F2F connections' own flow control.

References

[1] Akamai. IPv6 Adoption Trends by Country and Network. http://www.stateoftheinternet.com/trends.html. Accessed: 2015-05-16.

[2] Märt Bakhoff. Integrating VDE into the F2F framework. http://ds.cs.ut.ee/Members/mbakhoff/integrating-vde-into-the-f2f-framework, 2015. Accessed: 2015-05-21.

[3] D. J. Bernstein. The self-pipe trick. http://cr.yp.to/docs/selfpipe.html. Accessed: 2015-05-19.

[4] Debian. QEMU - Debian Wiki. https://wiki.debian.org/QEMU#QEMU_networking_with_VDE. Accessed: 2015-05-09.

[5] Jim Gettys et al. Bufferbloat: Dark Buffers in the Internet, 11 2011. https://queue.acm.org/detail.cfm?id=2071893.

[6] Apple Inc. Elevating Privileges Safely. https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/Articles/AccessControl.html. Accessed: 2015-05-20.

[7] Jason. Linux Thread Memory Usage. http://blog2.emptycrate.com/content/linux-thread-memory-usage, 2008. Accessed: 2015-05-06.

[8] Maxim Krasnyansky. Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt, 2000. Accessed: 2015-05-21.

[9] GNU libc project. The GNU C Library: Signal Handling. https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html. Accessed: 2015-05-20.

[10] LogMeIn. Hamachi VPN. https://secure.logmein.com/products/hamachi/. Accessed: 2015-05-20.

[11] Malyszkz. Network Topologies. https://commons.wikimedia.org/wiki/File:NetworkTopologies.svg, 2011. Accessed: 2015-05-14.

[12] Nick Mathewson. Working with events. http://www.wangafu.net/~nickm/libevent-book/Ref4_event.html, 2009. Accessed: 2015-05-06.

[13] Microsoft. select function. https://msdn.microsoft.com/en-us/library/ms740141%28v=vs.85%29.aspx. Accessed: 2015-05-06.

[14] Silver Moon. Code a simple tcp socket server in winsock. http://www.binarytides.com/code-tcp-socket-server-winsock/, 2013. Accessed: 2015-05-06.

[15] J. Moy. RFC 1583: Routing table lookup. http://www.freesoft.org/CIE/RFC/1583/55.htm, 1994. Accessed: 2015-05-20.

[16] Rashid Bin Muhammad. Threads: Why Threads? http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/threads.htm. Accessed: 2015-05-11.

[17] John Nagle. Congestion Control in IP/TCP Internetworks. https://www.rfc-editor.org/rfc/rfc896.txt, 1984. Accessed: 2015-05-21.

[18] Niels Provos Nick Mathewson. libevent. http://libevent.org/, 2000. Accessed: 2015-05-06.

[19] Inc. OpenVPN Technologies. OpenVPN - Open Source VPN. https://openvpn.net/. Accessed: 2015-05-20.

[20] SDL project. Simple DirectMedia Layer. https://www.libsdl.org/. Accessed: 2015-05-21.

[21] The IPOP project. Architecture overview. http://ipop-project.org/learn/architecture. Accessed: 2015-05-11.

[22] The IPOP project. IPOP. http://ipop-project.org/. Accessed: 2015-05-11.

[23] The Linux Information Project. Packet Switching Definition. http://www.linfo.org/packet_switching.html, 2005. Accessed: 2015-05-20.

[24] Virtualsquare project. Virtual Distributed Ethernet. http://wiki.v2.cs.unibo.it/wiki/index.php/VDE. Accessed: 2015-05-16.

[25] Virtualsquare project. Virtual Square Networking. http://wiki.v2.cs.unibo.it/wiki/index.php/Introduction#Virtual_Distributed_Ethernet. Accessed: 2015-05-16.

[26] Ilkka Seppälä. Design pattern samples in Java: Observer. https://github.com/iluwatar/java-design-patterns#observer, 2014. Accessed: 2015-05-07.

[27] Benoit Sigoure. How long does it take to make a context switch? http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html, 2014. Accessed: 2015-05-06.

[28] Shweta Sinha. A TCP Tutorial. http://www.ssfnet.org/Exchange/tcp/tcpTutorialNotes.html#CC, 1998. Accessed: 2015-05-20.

[29] Gabriel Southern. how to set pthread max stacksize. https://stackoverflow.com/a/15442272, 2013. Accessed: 2015-05-20.

[30] Single UNIX Specification. select(2) - Linux man page. http://linux.die.net/man/2/select, 1997. Accessed: 2015-05-06.

[31] Firewall CX team. The IEEE 802.3 Frame Format. http://www.firewall.cx/networking-topics/ethernet/ethernet-frame-formats/200-ieee-8023-frame.html. Accessed: 2015-05-21.

[32] George Thomas. Hubs versus Switches: Understand the Tradeoffs. http://www.automation.com/library/articles-white-papers/industrial-ethernet/hubs-versus-switches-understand-the-tradeoffs. Accessed: 2015-05-20.

[33] VirtualBox. 6.9. VDE networking. https://www.virtualbox.org/manual/ch06.html#network_vde. Accessed: 2015-05-10.

[34] Ian Wienand. Computer Science from the Bottom Up: Implementing abstraction. `http://bottomupcs.com/abstration.html`, 2004. Accessed: 2015-05-06.

[35] Wikipedia. OSI model. `https://en.wikipedia.org/wiki/OSI_model`. Accessed: 2015-05-21.

1. vde-runner.c example code

A minimalistic code example of using the VDE component:

```
int main(int argc, char *argv[])
{
  if (argc < 2 || !strcmp(argv[1], "--help")) {
    printf("usage: vde-runner <server port> [peer-ip peer-port [..]]\n");
    exit(1);
  }
  char *server_port = argv[1];
  struct F2FCore *core;
  struct tcp_server *server;
  struct tcp_group *group;
  struct vde_plugin *vde;

  if (f2f_init(&core, "my-display-name"))
    exit(1);
  if (tcp_server_start(&server, core, atol(server_port)))
    exit(1);

  char path[255];
  snprintf(path, sizeof(path), "/tmp/vde-%s", server_port);
  if (vde_plugin_attach(core, path, &vde))
    exit(1);

  tcp_group_create(&group);
  for (int i = 2; i < argc; i += 2) {
    struct F2FPeer *remote;
    if (f2f_peer_create(core, &remote))
      exit(1);
    if (tcp_comm_connect(group, &remote->comm, remote, argv[i], argv[i+1]))
      exit(1);
  }

  run_the_vm("f2f-vm-image.img", path);

  vde_plugin_detach(vde);
  tcp_server_stop(server);
  tcp_group_destroy(group);
  f2f_destroy(core);
  exit(0);
}
```