Virtio-Vsock - Configuration-Agnostic Guest/Host Communication

Johannes Wiesböck, Johannes Naab, Henning Stubbe
Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: johannes.wiesboeck@tum.de, {naab, stubbe}@net.in.tum.de

***Abstract*—Virtio-vsock provides zero-configuration communication channels to exchange data between a host and virtual machines running on the host. It builds upon the Socket API and the new addressing format AF_VSOCK, which allows easy porting of network applications to virtio-vsock. This paper explains the fundamentals of the new address format, and shows a flexible approach, which enables existing network applications to use virtio-vsock. This approach does not implement vsock support in every application but instead uses inetd-style socket activation to be applicable for many existing applications without modifying their source code. We focus on providing SSH connections to virtual machines over virtio-vsock, which will allow access to the virtual machines with almost no configuration. Additionally, we provide a generic solution for TCP-based applications.**

***Index Terms*—virtio, vsock, virtual machine socket, guest/host communication, ssh**

1. Introduction

Virtio-vsock is a zero-configuration communication interface, which enables data exchange between a host and virtual machines (VMs) running on it. It is designed to be available on a system by default without any configuration required. Further, it is based on the Socket API, which is also used for traditional network protocols.

Possible use-cases for this communication channel are guest agents, which run in a VM and interact with the host system, like the qemu-guest-agent [1]. Another use-case is to provide a host service to a VM like a remote file system. In the implementation part of this paper we will focus on running SSH connections over virtio-vsock. The goal of running SSH over a vsock connection is to provide an administration interface to VMs, which is independent of a network configuration. Thus, less configuration is needed and the interface can work more reliably.

This paper is structured as follows: First, we introduce the core concepts of virtio-vsock, which include both, high-level features, such as the address format, and implementation details like the underlying protocol. Next, we compare virtio-vsock with other alternatives for host/guest communication and we also show a present project that uses virtio-vsock. After that we describe an approach for using existing protocols such as SSH or HTTP over the vsock communication channel. Last, we evaluate the implementation and give a conclusion of our work with virtio-vsock.

2. Fundamentals of Virtio-vsock

This section introduces the basic concepts of virtio-vsock and implementation details.

2.1. Addressing Scheme

With VM sockets a new address format for the `socket()` system call named AF_VSOCK is added. An AF_VSOCK address is a 2-tuple consisting of a Context Identifier (CID) and a port. A unique CID is assigned to the host and to every VM in order to identify the individual machines. The CID is implemented as a 32-bit integer given in host byte order. Table 1 gives an overview of CIDs including CIDs which are reserved for special purposes [2].

The port of a vsock address is used to differentiate between multiple services running on one machine. Port numbers are implemented as 32-bit integers in host byte order [2], unlike TCP/UDP port numbers, which use 16-bit integers in network byte order. Port numbers below 1024 are called privileged. Only root can bind a socket to the privileged ports.

2.2. Socket Creation

Vsock connections can be managed using the Socket API. Thus, a VM socket can be created by a call to the `socket()` system call.

```
vsock = socket(AF_VSOCK, socket_type, 0);
```

According to version 5.2 of the Linux kernel source code [3], the only supported value for `socket_type` is SOCK_STREAM. This type provides reliable and stream-based communication with guaranteed and ordered delivery.

CID	Alias	Purpose
0	VMADDR_CID_HYPERVISOR	hypervisor
1	VMADDR_CID_RESERVED	reserved
2	VMADDR_CID_HOST	host
$[3; 2^{32} - 2]$	-	virtual machines
$2^{32} - 1$	VMADDR_CID_ANY	binding

TABLE 1: Overview of special CIDs

doi: 10.2313/NET-2019-10-1_14

Option	Description
REQUEST	initiate connection
RESPONSE	acknowledge connection initiation
RST	connection reset or address not bound
SHUTDOWN	request connection shutdown
RW	application data
CREDIT_UPDATE	updated credit data
CREDIT_REQUEST	explicitly request a credit update

TABLE 2: Overview of operations

2.3. Implementation Details

This section provides an overview of the protocol, which is used in vsock connections.

2.3.1. Flow Control. The stream-mode of virtio-vsock features a credit-based flow control mechanism, which prevents the sender from overloading the receiver [4]. The receiver informs the sender about its absolute amount of allocated receive buffer (buf_alloc) with every packet sent back or implicitly with a CREDIT_UPDATE packet, which is introduced in Table 2. The receiver also informs the sender about the amount of data, which was already forwarded to the application (fwd_cnt). Additionally, the sender keeps track of the absolute amount of data it has sent to the receiver (tx_cnt). Using this information, the sender can calculate its credit, which is the maximum amount of data it may send without overflowing the receivers buffer:

$$credit = buf\_alloc - (tx\_cnt - fwd\_cnt) \quad (1)$$

If the credit limit is reached, writing to the socket blocks until the receiver updates the fwd_cnt value.

2.3.2. Protocol. In this section we describe the lifetime of a stream-based virtio-vsock connection together with the operations involved in the connection. An overview of all possible operations is shown in Table 2. A connection consists of two endpoints, a server and a client, where the server runs on the VM and the client on the host or vice-versa.

We will now look into the steps involved in a possible vsock connection. Therefore, we will first look into the connection from an application point of view and later from the protocol point of view. First, the client application initiates the connection. Second, the client sends data to the server and third, closes the connection. These three phases are visualized as colored areas in Figure 1. Originally virtio-vsock used a different protocol than the one shown in this section. However, according to Hajnoczi [5] virtio-vsock protocol was partially reworked from the original protocol shown in [1]. The description of the protocol in this section was derived from observations made while examining connections using the packet analyzer Wireshark.

As shown in Figure 1, a connection is initiated with a two-way handshake. It begins with the client sending a packet of type REQUEST. If the server accepts the connection, it answers with an RESPONSE packet. The connection is now established. Application data is sent in a packet of type RW. Every time received data is forwarded to the application, for example when the server application reads data from the socket, the server sends a credit update to the client. The credit update informs the client about the updated *fwd_cnt* value. The connection is terminated with a two-way tear-down. The disconnecting side sends a packet of type SHUTDOWN, which is acknowledged with a RST packet terminating the connection.

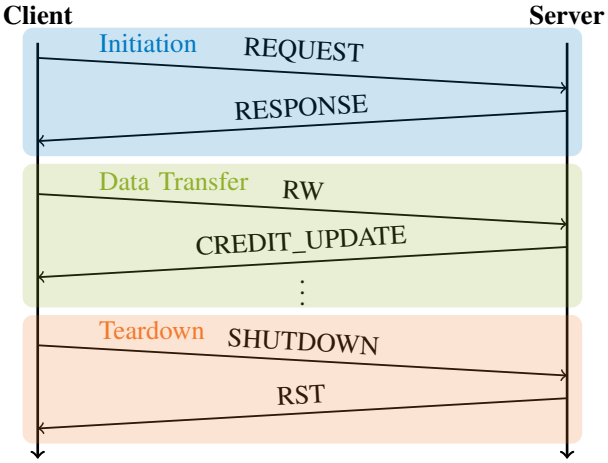

Figure 1: Overview of a sample stream-based vsock connection.

2.4. History

AF_VSOCK has originally been introduced to the Linux kernel in 2013 by VMware for VMware virtualization products [6]. AF_VSOCK was later implemented in virtio to be used with the kernel-based virtual machine (KVM) and QEMU. Virtio-vsock is part of the mainline Linux kernel since version 4.8 [7]. Support for virtio-vsock was added to libvirt in version 4.4.0 [8].

3. Related Work

This section will give a brief overview of alternatives to the virtio-vsock technology. Also an example where virtio-vsock is used in practice is covered in this section.

3.1. Alternatives to Virtio-vsock

Virtio-vsock can be compared to other technologies providing communication services between hosts and VMs. Two alternatives shown by Hajnoczi [1] are virtio-serial and virtual networking.

3.1.1. Virtio-serial. Virtio-serial is a virtual serial device, which is used to establish connections between hosts and guests [9]. A respective serial device is available on the guest and on the host-side. Applications can open the device and exchange data through the serial connection.

doi: 10.2313/NET-2019-10-1_14

Compared to virtio-vsock, virtio-serial has a few disadvantages [1]. The first downside is the limited number of channels, which equals the limited number of provided serial devices. To cope with this problem, data would have to be multiplexed on the application layer. Another disadvantage of virtio-serial is its implementation as a serial device. While this is not a problem per-se, it makes porting networking applications based on the Socket API more difficult than reusing the Socket API.

3.1.2. Networking. Another approach for guest/host communication is the usage of a virtual network [10]. This solution provides full network functionality to VMs. Thus, it is not only usable for guest/host communication but it also provides inter-VM networking and internet-access. The virtual network enables network applications to run between VMs without modifying them.This is possible, as the virtual network uses the internet protocol (IP) and thus supports all IP based applications. The downside of the networking approach is that creating network interfaces on the host and on the guests can be very complex and may not be desired [1]. In our case, we explicitly want to avoid additional network interfaces on the guest side, because they might influence the results of network-related tests or benchmarks running on the VMs.

3.2. NFS-vsock - File System over AF_VSOCK

Stefan Hajnoczi proposed support for the network file system (NFS) in 2016 [11]. The goal is to support NFS over vsock connections natively in the NFS implementation of the Linux kernel. For example, NFS over vsock could be used for network attached storage (NAS) services in cloud environments or to provide files to VMs during installation. Unfortunately, patches for vsock support in NFS have not been applied to the mainline Linux kernel so far, so using it requires a patched kernel.

4. Implementation

In this section we present the motivation for our implementation and possible implementation approaches. We select one approach and implement it for use with SSH and other protocols, such as HTTP and SMB.

4.1. Motivation

The motivation for this implementation is to enable various applications to use virtio-vsock for transport between hosts and VMs. We specifically focus on running SSH connections over VM sockets to provide a zero-configuration interface for VMs that is independent of a network configuration. Besides the SSH solution, a generic solution for TCP based services is also provided.

4.2. Approaches

In the following we compare two possibilities to enable applications to use virtio-vsock, namely native support and inetd-style.

4.2.1. Native Support. As stated in Section 2.2, AF_VSOCK reuses the existing Socket API, which simplifies the porting of network applications, as it should only require minor changes to the source code. By changing the first parameter of the call to `socket()` to AF_VSOCK and by updating the addresses accordingly, a network application could be ported. In many cases, this might not be sufficient to port the entire application, as only the networking part of an application can be ported easily, which might not apply to the entire application. An application, which is tightly bound to the TCP/IP protocol stack, can use the network configuration and the address format internally. Therefore, changes to the application logic are required to enable the AF_VSOCK format. Also user interfaces may be influenced when an additional protocol should be implemented. In general, porting an application natively to AF_VSOCK is not a trivial task and must be done for every application separately. Changes have to be made to the server and the client software respectively. A native implementation of AF_VSOCK support can be complex and therefore requires a lot of application knowledge.

4.2.2. Inetd-Style. An approach that can be applied to many services without modifying the application code is known as inetd-style socket activation.

When using socket activation, a super-server is set up to listen to incoming connections on a configured port. When a client connects to this port, the super server will accept the connection, start the actual application server and pass the connected socket to the application server. In inetd-style socket activation the connected socket is passed to the application by setting the servers standard input and output to the connected socket. In this scenario, the application server is not involved in the connection establishment and can be provided with a connected VM socket to communicate over AF_VSOCK. A possible super server is `systemd` which supports VM sockets since version 233 [12].

A disadvantage of the inetd-style is that it is not optimized for a specific application and thus has restrictions. Most importantly, it is required that an application server supports inetd-style socket activation. Another restriction is that additional ports can not be opened on behalf of the application, since all relays have to be set up in advance. This would make it unusable for example for FTP, which opens additional ports while in operation. Advantages of inetd-style are a simple implementation and support for many different services.

4.3. SSH

As mentioned in Section 4.1, we have a large interest in running SSH connections over VM sockets. SSH offers many features which can be used to enable this ability without port-forwarding and without modifying the source code of the SSH components. We show these features and how they are used in the following sections. Figure 2 shows the connection establishment, when a SSH client on the host tries to connect to a server running on a VM.

doi: 10.2313/NET-2019-10-1_14

```
ssh -o ProxyCommand='socat - SOCKET-CONNECT:40:0:x0000x16000000x04000000x00000000' user@vm
```

Listing 1: SSH command to connect to a server over a vsock connection

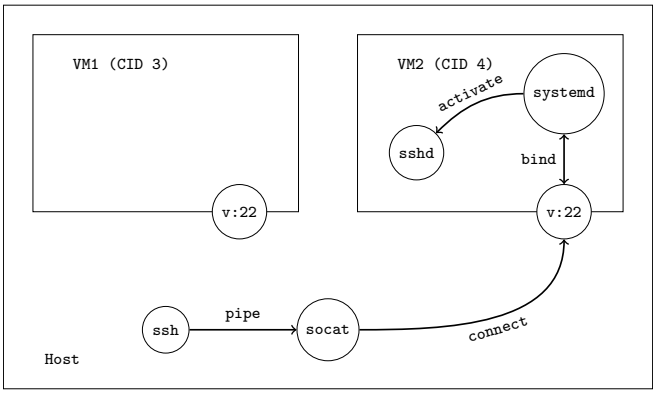

Figure 2: Establishment of a SSH connection from the host to a guest over AF_VSOCK

4.3.1. Client-Side. For the client, we use the widely installed OpenSSH client, which offers the `ProxyCommand` command line parameter. `ProxyCommand` can be any application that can connect to a remote SSH server and forward traffic coming from its standard input to the server and vice-versa. This proxy application will then connect to the destination server and forward all traffic from the client to the server and vice-versa. If this parameter is used, all SSH traffic is then passed through this proxy instead of the usual network connection. This creates the possibility to use `socat` [13] to forward the SSH connection to the destinations vsock. A possible command to connect to a SSH server using this method is shown in Listing 1. Here, `socat` connects to the SSH server running on port 22 of CID 4 and forwards this connection to the SSH client. Since `socat` does not offer special syntax for AF_VSOCK, the generic syntax has to be used. After `SOCKET-CONNECT`, `socat` is instructed to use protocol number 40 (AF_VSOCK) and type 0. After that, a hexadecimal representation of `struct sockaddr_vm` is given, which contains the port 22 (0x16) and the CID 4 (0x04) of the destination.

4.3.2. Server-Side. On the server-side, the SSH server `sshd` is started using inetd-style socket activation provided by `systemd` as explained in Section 4.2.2. This way, `systemd` is listening to incoming SSH connections on a local vsock port. Once a connection arrives on this port, systemd will accept the connection and start `sshd`. The connected file descriptor, which represents the accepted SSH connection, is passed to `sshd` as its standard input and standard output. `Sshd` is now able to use the SSH connection without being involved in the establishment of the connection.

4.4. Generic Solution using Port-Forwarding

Many existing network applications have no native support for the vsock protocol. Therefore, we implemented a generic solution that can be used by many applications using the TCP protocol but do not support vsock. It is not necessary for the application server to support inetd-style socket activation. This solution works by mapping vsock addresses to local IPv6 addresses. Thus, applications which are restricted to use TCP connections can access the vsock protocol over the interface introduced in Section 4.4.2.

4.4.1. Address Mapping. To make the vsock protocol available to applications, which support only TCP connections, CIDs of the vsock domain are mapped to local IPv6 addresses. For this mapping we use the IPv6 subnet fc00::/7, that is assigned for unique-local-unicast addresses, which are not routed on the internet. IPv6 addresses from this subnet can be chosen for local usage without colliding with globally unique addresses. Before CIDs can be mapped to IPv6 addresses, a random /64 prefix is chosen from the subnet fc00::/7. By definition, a locally assigned prefix from this subnet should have its eighth bit set to one [14]. Therefore, a possible valid prefix would be fd00:abcd:ef12:3456::/64. After the prefix is chosen, a CID is mapped to the IPv6 address space by adding the value of the CID to the prefix. In this example, this would result in CID 3 being mapped to the IPv6 address fd00:abcd:ef12:3456::3 and vice-versa.

Port numbers are mapped to TCP ports without changes if possible. Since AF_VSOCK offers 2^{32} different port numbers, all 2^{16} TCP ports can be directly mapped to vsock ports. Thus, if a service known from TCP is offered over vsock, its well-known port number can be reused. For example, a web server which usually listens to TCP connections on port 80 or 443, should also be available on the same vsock ports.

4.4.2. Forwarder Implementation. The implementation of the generic forwarder extends the concept in Section 4.2.2 by adding a relay to the setup that can forward traffic from TCP to vsock connections and vice-versa. Also in this implementation `socat` [13] is used as a relay software.

To add a relay to a TCP based server software, a socket activated instance of `socat` is configured on the

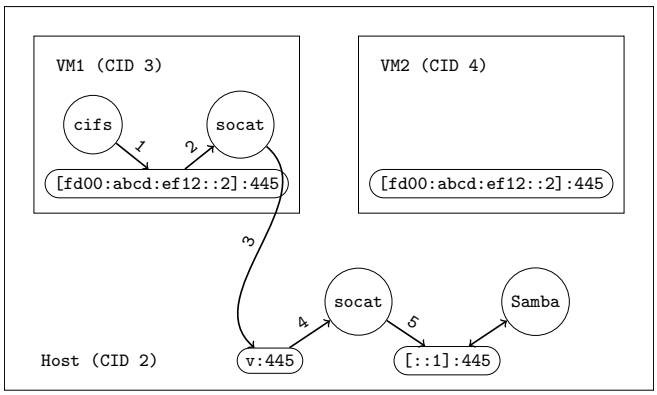

Figure 3: Establishment of a connection using two `socat` relays.

doi: 10.2313/NET-2019-10-1_14

servers machine. The configured VM socket is monitored by `systemd` for incoming connections. Once a client connects to the monitored port, `systemd` will start a `socat` instance and pass the connected vsock to it. The `socat` relay will then connect to the actual server listening on a local TCP port to establish the connection.

The client-side forwarder utilizes the address mapping introduced in Section 4.4.1. A client, which only supports TCP can connect to a `socat` relay via TCP. This relay can then forward the connection to the destination server via vsock. This socket activated relay is listening on a IPv6 address corresponding to the destination machines CID. If a client connects to a port on this special IPv6 address, the address will be translated into the corresponding CID. The `socat` relay will then connect to the given CID and to the port and forward traffic from the client to the server running on this address.

Figure 3 illustrates a possible scenario, where a Samba server is running on the host machine providing a file sharing service to VMs. If the SMB client software `cifs` tries to connect to the SMB server running on the host with CID 2, it actually connects to the IPv6 address fd00:abcd:ef12::2 representing this CID. Once a connection arrives on this socket, a `socat` instance is started via socket activation. This `socat` instance forwards the connection to the host over a vsock connection. The TCP port number is reused for VM sockets and is thus 445 for TCP as well as for vsock. When the host receives an incoming connection on vsock port 445, it will also start a `socat` relay to forward the traffic from this port to the SMB server listening on the local host on port 445. The connection is now established and data can be exchanged between client and server.

4.5. Evaluation

The proposed concept was successfully tested with SSH and worked reliably. The generic solution shown in Section 4.4 was tested with HTTP and SMB. HTTP was tested using a nginx web server running on a VM and a browser on the host. The SMB test setup included a Samba server running on the host system and a volume mounted in a VM over the vsock forwarder.

4.5.1. Performance. The achievable throughput of the forwarder was evaluated using the tool iperf3 with patched-in support for vsock connections [15]. For comparison, also the throughput of both, a virtual network interface and of a raw vsock connection was evaluated in addition to the forwarder. The base system for evaluation was a Lenovo ThinkPad T430 with a Intel Core i5-3320M CPU clocked at 2.60 GHz. The iperf3 server was running on a VM and the client was running on the host. In an iperf3 run with a duration of ten seconds, the virtual network connection achieved an average throughput of 14.2 Gbit/sec. The average throughput of a native vsock connection was 12.9 Gbit/sec on average and thus slightly slower than the network connection. In contrast to these comparably high values the forwarder setup using two `socat` relays only achieved an average throughput of 1.5 Gbit/sec. The considerably lower throughput may be caused by the multiple times that data has to be copied between buffers and the additional protocols involved.

4.5.2. Security Considerations. During development various connection scenarios were tested. As intended, we were not able to establish vsock connections between two VMs but only between the host and one VM. One exception is loopback connectivity. It is possible to connect from a VM to the same VM via AF_VSOCK. While this may be desired behaviour, it is to note that services, which are exposed over vsock, have to be secured properly if they should not be accessible from the VM itself. An example scenario would include a VM that should be configured via SSH over vsock. For configuration, a client must be able to connect as root over this SSH interface. Other than for configuration, the VM is operated by a untrusted user, who should not have root access to the machine. Because of the loopback connectivity, the user can connect to the local SSH server over vsock, which makes it necessary that the access is secured properly with a password or preferably with public keys. Without loopback connectivity, the SSH server would only be accessible from the host machine an thus could not be used by the user working on the VM. This might give the opportunity to omit authentication for SSH connection form the vsock interface, since it could only be access from the host machine. Loopback connectivity was explicitly removed by Google for ChromeOS [16] to prevent applications from connecting to other applications on the same machine. Loopback connectivity is present in the mainline Linux kernel and removing it would require a patched kernel.

5. Conclusion

This paper gave a short introduction to the virtio-vsock technology. We showed, that virtio-vsock provides a reliable and user-friendly communication mechanism for VM setups. We were able to enable vsock support for different services using inetd-style socket activation.

Even though socket activation worked for all tested services, we would like to see native support for AF_VSOCK connections in the future, as it might outperform the shown implementation using two `socat` relays. Native support would also make it easier to use network applications between hosts and VMs, with zero-configuration. So far, desirable features such as support for NFS are not part of the mainline Linux kernel, which would require building a custom kernel. Together with the security considerations in Section 4.5.2, it has to be considered if this effort is worth the benefits gained in features and security. Future research should investigate, if it is possible to increase the throughput of the forwarder-setup, possibly by investigating if a specifically developed and tuned forwarder would perform better than `socat`.

References

[1] S. Hajnoczi, “virtio-vsock Zero-configuration host/guest communication,” Accessed on: 2019-06-05. [Online]. Available: https://vmsplice.net/~stefan/stefanha-kvm-forum-2015.pdf

doi: 10.2313/NET-2019-10-1_14

[2] *man 7 vsock*, Accessed on: 2019-08-26. [Online]. Available: http://man7.org/linux/man-pages/man7/vsock.7.html

[3] "virtio_vsock.h (kernel version 5.2)," Accessed on: 2019-06-01. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/virtio_vsock.h?h=v5.2

[4] A. He, "Introduce VM Sockets virtio transport," *LWN.net*, 2013, Accessed on: 2019-06-01. [Online]. Available: https://lwn.net/Articles/556550/

[5] S. Hajnoczi, "Add virtio transport for AF_VSOCK," *LWN.net*, 2016, Accessed on: 2019-06-01. [Online]. Available: https://lwn.net/Articles/695981/

[6] A. King, "VSOCK: Introduce VM Sockets," Accessed on: 2019-09-01. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d021c344051af91f42c5ba9fdedc176740cbd238

[7] S. Hajnoczi, *Features/VirtioVsock*, Accessed on: 2019-06-09. [Online]. Available: https://wiki.qemu.org/Features/VirtioVsock

[8] "libvirt: Releases," Accessed on: 2019-06-09. [Online]. Available: https://www.libvirt.org/news.html

[9] A. Shah, *Features/VirtioSerial*, Accessed on: 2019-06-12. [Online]. Available: https://fedoraproject.org/wiki/Features/VirtioSerial

[10] *Documentation/Networking*, Accessed on: 2019-06-12. [Online]. Available: https://wiki.qemu.org/Documentation/Networking

[11] S. Hajnoczi, "NFS over virtio-vsock Host/guest file sharing for virtual machines," Accessed on: 2019-06-22. [Online]. Available: https://vmsplice.net/~stefan/stefanha-connectathon-2016.pdf

[12] "Systemd NEWS," Accessed on: 2019-06-13. [Online]. Available: https://github.com/systemd/systemd/blob/v233/NEWS#L303

[13] *man 1 socat*, Accessed on: 2019-08-26. [Online]. Available: https://linux.die.net/man/1/socat

[14] R. Hinden and B. Haberman, "Unique Local IPv6 Unicast Addresses," Internet Requests for Comments, RFC Editor, RFC 4193, October 2005.

[15] S. Garzarella, "iperf," Accessed on: 2019-08-27. [Online]. Available: https://github.com/stefano-garzarella/iperf-vsock

[16] "Chrome OS source: virtio_transport.c," Accessed on: 2019-06-22. [Online]. Available: https://chromium.googlesource.com/chromiumos/third_party/kernel/+/refs/heads/chromeos-4.4/net/vmw_vsock/virtio_transport.c#188

doi: 10.2313/NET-2019-10-1_14