Improving the Performance of an IP-over-P2P Overlay for Nested Cloud Environments

Research Project 2

Dragos Barosan
dragos.barosan@os3.nl
Supervisors:
Ana Oprescu
Kaveh Razavi
Kyuho Jeong
Renato Figueiredo

August 19, 2015

UNIVERSITEIT VAN AMSTERDAM

Abstract

IPOP is a relatively new open source software that allows users to define their own private virtual networks, leveraging social network connections. Achieved throughput over IPOP links is bellow what is measured over physical links. Analysis shows that a main source of overhead is the encryption provided by IPOP. Traffic is encrypted by default over all IPOP connections, with no granularity in enabling or disabling it. This research proposes and implements the feature of selective security, in which only a few users will be affected by the encryption overhead. The investigation produced detailed measurements in order to profile the application and help on improving the performance. Code analysis shows that the current IPOP version suffers from implementation issues in the code handling packet forwarding. Solutions are proposed, implemented and tested, with discovered issues being discussed.

Contents

1 Introduction

Cloud computing has become a ubiquitous way of assuring an affordable infrastructure for today's new or developing companies. Even tough technology has matured in recent years, there are still requirements that cannot be fulfilled by providers. This comes either from economical or technological limitations. Academic research focused on these issues and, by leveraging open-source software, aims to create architectures that can interoperate with cloud providers and deliver more flexibility to the users.

One such initiative is Kangaroo, developed as a joint international project between VU University Amsterdam, University of Florida and Universite de Rennes [1]. It was developed as a solution to three issues currently present in the cloud providers services: No fine granularity in selection of virtual machine (VM) types, limited or no control over VM placement, lack of standardized API. The project proposes a tenant-centric architecture on top of resources of different clouds by employing nested virtualization. Kangaroo promotes vertical scaling: End-to-end network performance is much better between co-located nested VMs and larger VMs are often more cost-efficient. The main technologies used are Openstack [2] for managing the nested VMs, Skippy [1] for providing an abstract API for users regardless of the underlying cloud provider and IPOP for end-to-end connectivity. Once resource limits are reached within a VM, nested VMs have to communicate with their counterparts on different VMs. IPOP is used to manage this connections by creating an overlay over the network that connects the underlying VMs.

IPOP is an ongoing project that aims to provide a secure, cheap and simple to configure end-to-end connection between users [3]. IPOP's architecture, inspired by Software-Defined Networking(SDN), allows for flexibility and straightforward implementation of new extensions. Virtual links are created and managed by IPOP-tincan module. Links are created between users that are identified as peers, using XMPP [4] as a peer discovery protocol. In order to assure connections even when the users are behind a Network Address Translation(NAT) device, STUN [5] and TURN[6] protocols are used.

Preliminary tests have been done in order to verify the project setup, using a 1 Gbps link between two servers running IPOP. We noticed that there is a difference between network throughput when two users are communicating directly (i.e. 950 Mbps) and when they are using IPOP (i.e. 550 Mbps and 160 Mbps when security is enabled). While we expect a small performance overhead due to a level of indirection (i.e., up to 10%), this initial measurements show around 6x degradation in performance with security, and 42% degradation in performance without security. Once the causes of this performance penalty are identified, we can improve IPOP's performance substantially. If IPOP proves to be an architecture that can support a performance close to what is available on direct links, then a larger user adoption can follow and more use cases become feasible.

2 Research questions

The aim of this investigation is to analyze the IPOP architecture, identify potential performance bottlenecks and provide, if possible, solutions. The scope of this project can be formulated as the following research questions:

- What are the sources of the performance problems?
- What are possible solutions for fixing the problems?

3 IPOP Overview

IPOP is an open source software that enables users to define and create their own virtual private networks(VPN). In order to create VPN tunnels, an XMPP overlay is used. The overlay enables IPOP nodes to discover each other and exchange endpoint information. To do this, they connect to an online social server(OSN) that is XMPP compliant. This server has relationships defined between users that enables them to connect. An example of a social network with this support enabled is Google Talk[7].

In the IPOP network, nodes are identified by their UID (User Identifier). A 40 bytes IPOP header is added to outgoing packets, containing 20 bytes destination UID and 20 bytes source UID. Using this information and a table mapping the IP of the peer to the UID, packet forwarding is achieved. After the IPOP header is attached, all packets that are to be transmitted over IPOP tunnels, regardless if they are TCP or UDP datagrams, are encapsulated with a new UDP and IP header, as shown in Figure 1, and transported over physical links as UDP datagrams. This is done using libjingle[8], a Google implementation of Jingle[9], as a signaling and forwarding protocol.

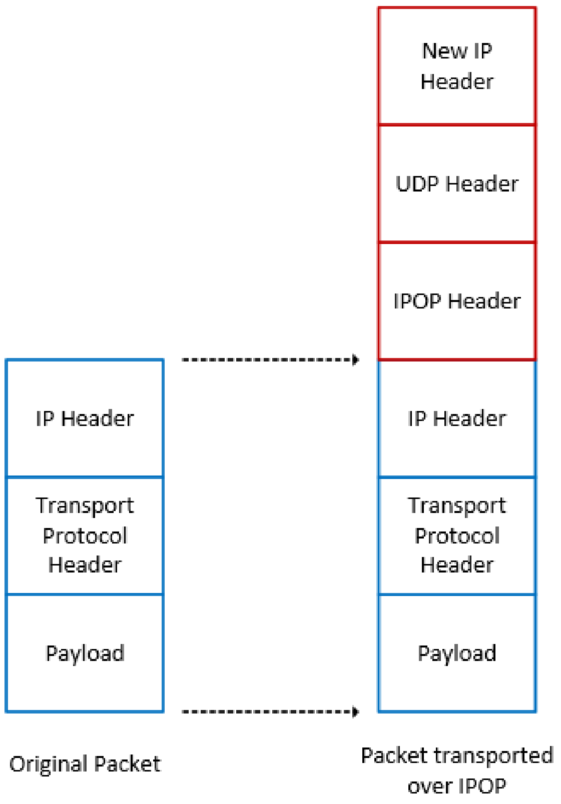

Figure 1: IPOP encapsulation process

Because of IPv4 address space exhaustion, it is important for users to be able to use IPOP when they are behind a NAT device. For this purpose NAT traversal functionality is implemented. Session Traversal Utilities for NAT(STUN) is used in order for peers to discover their public IP. In cases where a symmetric NAT is implemented, Traversal Using Relays around NAT(TURN) is used. The selection of the two tunneling protocols is done automatically by IPOP and it is transparent to applications running on the host.

Being based on SDN principles, the architecture separates the control plane and the data plane into multiple modules.

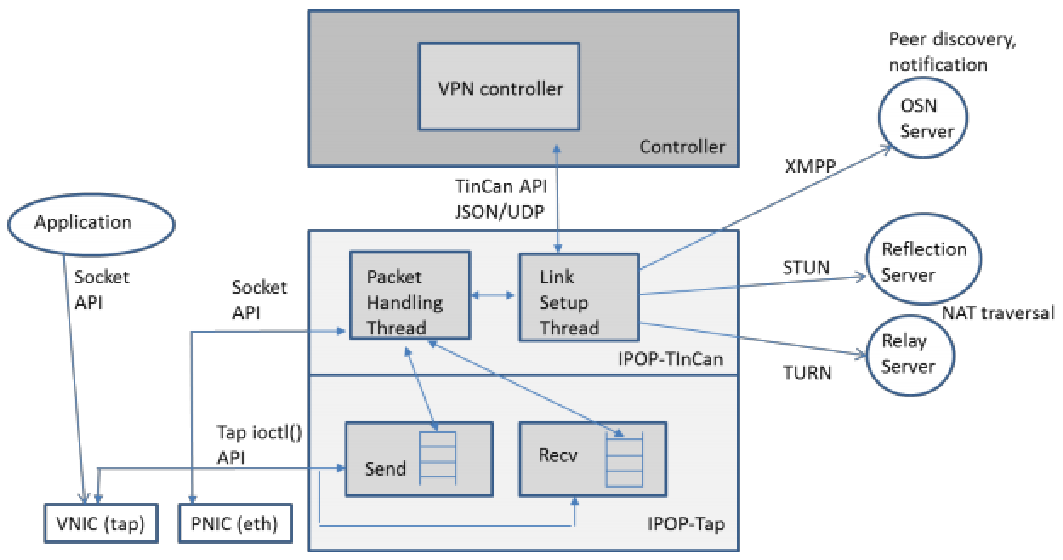

Figure 2: IPOP Architecture [3] [1]

As it can be seen in Figure 2, there are 3 main modules that provide the functionality of IPOP:

- **IPOP Controller** is responsible for setting up the policy of creating and destroying links between peers. It uses the Tincan API in order to control the behavior of IPOP-Tincan and implement different policies (e.g. link creation criteria, IP address allocation, network configuration scheme). If a UID is not mapped to any IP then IPOP-Tincan sends the packet to the controller who must handle it, usually by creating a new connection. At the time of this writing, there are two controllers available: SocialVPN and GroupVPN. The former creates connections to social peers such as each user has their own view of the network(each user has its own subnet and IP address translation takes place). In GroupVPN connections are created among everyone who is defined as being part of a group on the

[1]image 3.1 from IP over P2P (IPOP) White Paper 07/18/2014

OSN server. Every node is in the same virtual address space, so no IP translation mechanism is enabled. There are also two approaches on creating the links: On-Demand and Proactive. On-Demand mode creates a link only when a packet needs forwarding towards a certain peer and the link is trimmed after a period of inactivity. Proactive mode creates links immediately when a peer is detected as online.

- **IPOP-Tincan** is the core component of the system. It leverages libjingle to achieve three capabilities: XMPP support, encryption, connection establishment and management between peers. It receives the packets from IPOP-Tap and forwards them to the appropriate connection based on the UID. Here is where the table with UID-IP mappings is maintained. In a separate thread(i.e. Link Setup Thread), it offers an JSON API to the controller and listens for any requests in order to implement the IPOP policies. It also forwards to the controller any messages coming from the XMPP overlay.

- **IPOP-Tap** is the lowest level module and is an interface between IPOP-Tincan and the IPOP virtual network interface on the host. It has the role of interacting directly with the host operating system. At startup, it creates and configures a virtual network interface (VNIC) and assigns an IP address as specified by the controller. The VNIC is created using the kernel tap module. It reads outgoing packets from the VNIC and writes incoming packets to the VNIC. It is also responsible for adding and removing the IPOP header from packets.

4 Setup

In order to perform performance test on the IPOP architecture, a setup had to be build. It is represented in figure 3 and consists of the following hardware:

- An Ubuntu 14.04 x64 server with 8 GB of memory, two 1 Gbps Ethernet interfaces and an Intel processor with eight cores running on a frequency of 1.8 Ghz. It acts as an Online Social Network Server.
- Two Ubuntu 14.04 x64 servers with 16 GB of memory, 2 Gigabit Ethernet interfaces and an Intel processor with eight cores running on a frequency of 2.5 Ghz. Their role is that of IPOP nodes.

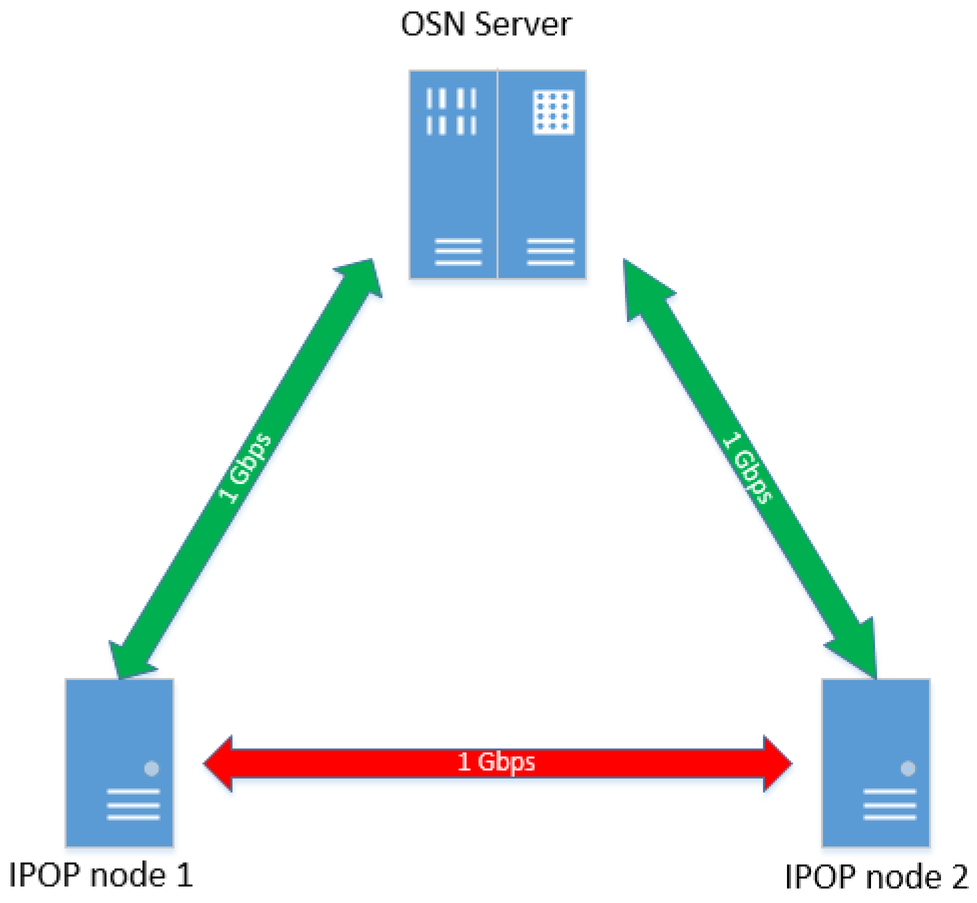

Figure 3: Test Setup

The OSN server is connected to both IPOP nodes and provides the services necessary for the functionality of IPOP. In order to fulfill the role of an OSN, ejabberd[10] was installed. This enables peers and group memberships to be defined and allows IPOP nodes to communicate over XMPP with the OSN server in order to query status information about their peers. For the experiments, multiple users were registered with ejabberd and configured to be part of the same group. Ejabberd also provides STUN services, but it was not enabled since NAT was not used.

The IPOP nodes, besides being connected to the OSN server, are also directly connected between them. This was done in order to accurately measure the network performance on the link without the complexity of adding extra variables (e.g. intermediary nodes, multiple routes). IPOP 15.01.01[11], the latest version at the start of the project, was compiled from source on this servers and the GroupVPN controller[12] was used. There is one configuration file, ***config.json***, used by both the controller and IPOP-Tincan. This must contain at least the IP address that will be used by IPOP tap interface in the virtual network, the IP address of the OSN server and the xmpp username and password of one of the users previously registered on the OSN server. A sample configuration file is attached in appendix A.

5 Security Functionality

IPOP was initially developed under the principle that tunnel connections between peers are build upon the Internet infrastructure, which is inherently insecure. In order to satisfy the requirement of security that is desired by some users and because UDP tunnels are used, IPOP makes use of the DTLS (Datagram Transport Layer Security) protocol[13]. DTLS is a derivation of TLS (Transport Layer Security). It is designed to accomplish the same objectives as TLS (i.e. integrity, authentication and confidentiality), but for packets that use UDP as a transport protocol. Security is turned on by default, but users have the possibility of turning it off by adding ***"sec": false*** as a parameter in the IPOP configuration file.

Bandwidth measurements between the two IPOP nodes has been performed using iperf. The performance was measured over the IPOP link in two cases: security disabled and security enabled. TCP was specified as transport protocol for the iperf tests, so in both cases we have the situation where TCP datagrams are transmitted over UDP tunnels. The results are shown in the table bellow.

Table 1: Security bandwidth test results

Security	*Average (Mbps)*	*Maximum (Mbps)*
Enabled	161	186
Disabled	550	668

As it can be seen from table 1, the overhead of using DTLS introduces a performance penalty of approximately 243%. At this moment, there is no possibility of selecting to which peers a connection should be secured and to which it should not.

To motivate why a more granular approach for enabling security would be better, an example use case is provided. One can think of a distributed virtual cluster where an organization may have its infrastructure virtualized over multiple cloud providers. In this case, virtual machines that belong to the same provider would communicate securely. If IPOP is used in this scenario, the requirement for IPOP secure links would be only between peers on different cloud provider networks because data would flow over the public network.

In order to understand how the feature of selective security can be enabled, the process of setting up the link between two peers must be explored.

The process of creating an IPOP connection is shown in figure 4 and described bellow in seven steps.

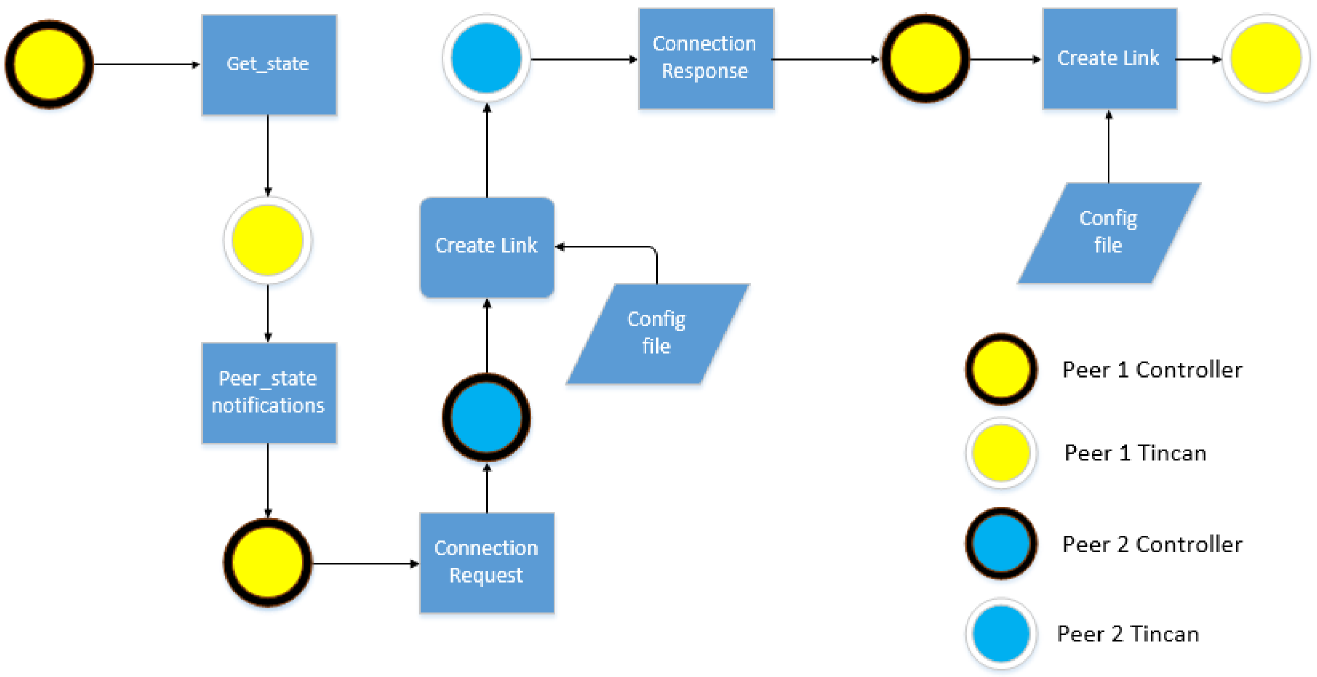

Figure 4: Connection creation in IPOP

1. The controller, using the Tincan API function *get_state* asks for status information about its peers.
2. Upon receiving the call, Tincan queries the OSN server about the configured peers and checks for which of them connections are created.
3. Tincan transmits the gathered information to the controller via notification messages, one for each peer.
4. The controller generates a connection request message and sends it to the other peer using the XMPP overlay.
5. The receiving peer of the connection request calls the Tincan API function *create_link* to indicate that it is ready to start a connection. Some of the function arguments are obtained from the configuration file, while others are results of local computations.
6. Tincan initiates the connection and also generates a connection response message to the first peer.
7. Upon the receival of the connexion response, the controller of the first peer also calls *create_link* and a connexion is established.

It is important to mention that the parameter that enables or disables security is obtained from the configuration file and used when ***create_link*** is called, as it will prove useful in the implementation of selective security.

As described in section 3, when IPOP starts, it creates a tap interface that is used to capture traffic from the host. This interface has an allocated IP which is transmitted to its peers and can be used to identify the IPOP node. Taking this into account, a first approach for enabling selective security would be to list in the configuration file the IPs of nodes that require secure communication. The controller is to be modified so in the process of setting up the link, before calling ***create_link***, it will search the IP of the current peer in the configuration file and enable security only if there is a match. This approach has issues regarding the scalability of the method, it becomes tedious to add peer IPs in the configuration file once their number grows significantly. Another issue is the fact that not all peer IPs may be known in advance by the users who have to add them in the configuration file. This requires extra communication between different users so they settle on a consistent value and every time there is an IP change this has to be communicated. These factors introduce unnecessary management overhead, so a better solution would be required.

To resolve this management overhead, we should not rely on the IP addresses for enforcing selective encryption. For this purpose, the notion of security group is defined and works as follows.

1. Each IPOP node is part of one or more security groups, as specified in the configuration file.

2. The peers will communicate to each other the security groups they are member of.

3. Each peer will make an intersection of the set of locally defined groups with the ones received from the peer.

4. If the result is a non-empty set, then security is enabled, otherwise communication goes over an unencrypted link.

The challenge that presents itself with this solution is how to implement step two of the process described above. Both peers have to transmit to each other the groups they are part of. Looking at figure 4, it can be seen that there is an exchange of messages between the peers (i.e. connection request and connection response) that could be used to encode security group information. The connection request is created by the controller, while the connection response is created by Tincan, so both IPOP modules have to be modified.

Debugging the application and analyzing the code, the format of the two messages was determined and it is presented in figures 5 and 6:

TYPE: con_req	DATA: Fingerprint	UID

Figure 5: Connection request message

TYPE: con_resp	DATA: Fingerprint, Connection Candidates	UID

Figure 6: Connection response message

As it can be seen the format is similar. A good place to introduce the group information is in the **DATA** section. Because the fingerprint is of fixed size and the length of Connection Candidates can vary, in order for IPOP to parse correctly the messages, we preserve this order and introduce the group data at the beginning of the field encoded together with its length in the first position. The parser will read the group information with the specified length, it will then parse the remaining of the message as before.

Figures 7 and 8 show the message format after the modification.

TYPE: con_req	DATA: Group List Length, Group List, Fingerprint	UID

Figure 7: Connection request message with group information

TYPE: con_resp	DATA: Group List Length, Group List, Fingerprint, Connection Candidates	UID

Figure 8: Connection response message with group information

The implementation has been completed and tests show that it is functional. Security groups can now be defined and as a result selective security is enabled. Bandwidth improvement up to three and a half times can now be achieved using this feature.

Using this method for enabling security scales well because only a few security groups have to be defined in the IPOP configuration file. No individual information about IPOP peers is required and multiple IPOP nodes can be part of one or more security groups. The problem of management overhead is also solved. Now organizations can predefine criteria on which IPOP nodes are members of a specific security group and as a result the users will only have to consult this norms in order to choose the correct configuration. This benefits come at the expense of an increased cost in the implementation of the solution. The comparison of the two proposed solutions is presented in table 2.

Table 2: Comparison between IP based and Security Group based Selective Security

	IP address based	*Security Group based*
Implementation Complexity	✓	✗
Management Overhead	✗	✓
Scalability	✗	✓

6 Profiling IPOP's code

For this project and future work, it is important to determine the runtime behavior of IPOP. To accomplish this, multiple measurements have been performed. It is important to mention that all results presented in this section are from tests where TCP was used as transport protocol for outgoing packets.

Firstly it was measured at what point does a bottleneck occur. Bandwidth tests were done over the IPOP link with various packet lengths. We started with a small packet size (i.e. 50 bytes), measured the bandwidth achieved and moved on with an increased packet size. It is apparent that the maximum average bandwidth (i.e. 550 Mbps) is achieved when the packet is of size 1000 bytes. When performing the same test directly over the physical link, it was observed that for a packet size of 1000 bytes we get the same average bandwidth as when transmitting over IPOP links. The bandwidth results diverge once we increase the packet size over 1000 bytes. Increasing the packet size continues to increase the achieved throughput over the physical link, towards the channel limit of 1 Gbps. With IPOP links, for values higher than 1000 bytes, we get approximately the same average as for packet sizes equal to 1000 bytes, but with a bigger variation of bandwidth measured in the interval of a second. This shows that IPOP can handle traffic with packet sizes under 1000 bytes, but problems appear once that limit is reached.

In order to investigate if the application is CPU bound, a CPU load test was done. Because IPOP is a multi-threaded application and it is important to measure each core in part, *mpstat*[14] was used as measuring tool. During the execution of the program it was observed that, on the sending side, no core went near 100% utilization, while on the receiving side, one of the cores was most of the time at 100% or near that value. Further investigation showed that the receiving thread of IPOP-Tap was responsible for this intense activity, indicating that the receiver side may be a bottleneck.

To profile the code with more accuracy, specialized tools were used: OProfile[15] and Zoom[16]. Both are statistical profiling tools that determine the amount of time spent on certain functions by sampling the processor. In order to get meaningful results, symbol information is required by the tools. For this purpose kernel and libc debug symbols were installed and IPOP was compiled with debug information.

The tools were used in the following cases on both the receiver and the sender nodes:

- On iperf running over an IPOP connection
- On iperf running over a direct connection
- System wide profiling during an iperf test over an IPOP connection
- System wide profiling during an iperf test over a direct connection
- On IPOP-Tincan while iperf was running

In order to gather a significant number of samples, tests were run for the duration of ten minutes and were repeated multiple times in order to eliminate any possible error. The files containing the full results are available on Github in the link provided in appendix B for everyone desiring to further work on improving the project.

One of the results shows how many more samples are captured in a system wide profile test when running IPOP and when not. This is shown in table 3.

Table 3: System wide number of samples collected by OProfile

IPOP sender node		IPOP receiver node	
IPOP running	*IPOP not running*	*IPOP running*	*IPOP not running*
15087703	797849	28793400	833717

It is important to notice that when IPOP is not running, the number of collected samples is similar. When IPOP is running we see that the number of samples collected from the sender is half of that on the receiver node. This further suggests that the receiver is a possible bottleneck in the communication.

Another observation can be made about the difference in the number of samples collected when IPOP is running and when it is not. The increased number of samples comes from the underlying activity of IPOP, specifically IPOP-tincan module. Analyzing the profiling results on the receiver side during the iperf test, without IPOP enabled, we can see that the function *copy_user_generic_string* is present with the most number of samples. When the iperf bandwidth test is done over IPOP links, we can see that the number of samples for *copy_user_generic_string* in the context of iperf is approximately the same, but extra copying takes place with functions such as *copy_user_generic_string* and *__memcpy_sse2_unaligned* that are being used by IPOP-tincan. The two functions used by IPOP-tincan for copying introduce together in the results approximately 7.5 more samples than *copy_user_generic_string* used by iperf. The same observation can be done on the sender side where *__copy_user_nocache* is the dominant function in number of samples captured in the processor when iperf is running over physical links. Once communication takes place over IPOP links, the dominant function becomes *copy_user_generic_string*, used together with *__memcpy_sse2_unaligned* as copying routines by IPOP-tincan. This extra copying takes place as a result of IPOPs architecture inner-workings. When an user space application is ready to transmit data over the network it calls kernel space functions which build the packet. If the packet is towards an IPOP node, the IPOP tap interface captures the packet and it is brought back in user space where it is processed by IPOP. After that the message is passed to kernel space where the final packet is constructed and sent over the physical interface. The inverse operations takes place at the receiver: the packet is received on the physical interface in kernel space, passed to IPOP for processing in user space, sent in kernel space where it is routed to the IPOP tap interface and it finally reaches the application in user space. The process is exemplified in figure 9.

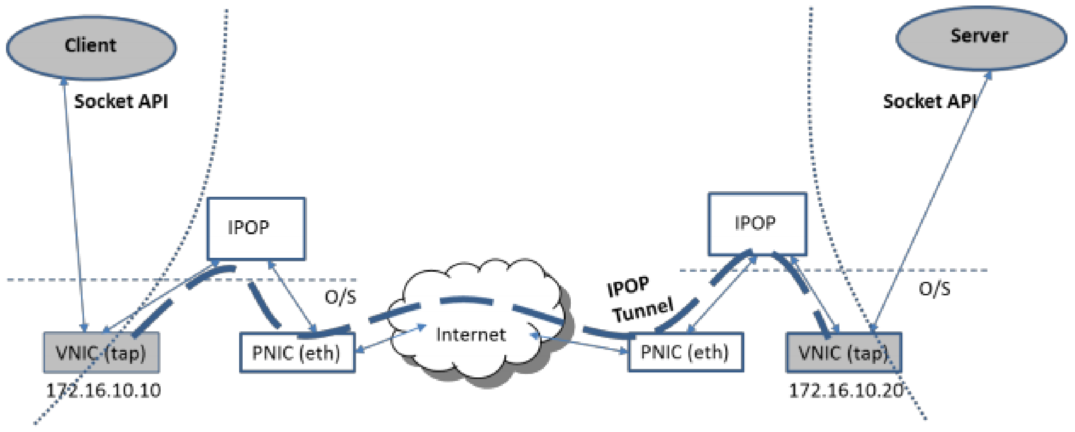

Figure 9: IPOP message transmission [3] [2]

The process described above does introduce extra overhead due to the extra copying and context switching and may affect the performance of IPOP, but not to the extent described in section 1. If this was truly the cause then on both sides processors with load of 100& should be observed since the overhead introduced by the extra copying on both the sender and receiver is at the same magnitude, judging by the number of captured samples.

In both nodes(i.e. receiver and sender) it is observed that the nf_conntrack kernel module[17] is one of the top symbols present in the results. This is used for connection tracking and if disabled it could reduce the stress on the CPU.

Another symbol present in the results is pthread_mutex_lock, used for synchronizing the receiving and sending queues of IPOP-Tincan and by libjingle. Investigating ways to reduce dependency on mutexes would eliminate part of the overhead and possibly improve performance.

[2]image 3.2 from IP over P2P (IPOP) White Paper 07/18/2014

7 Optimizing IPOP's code

IPOP is still an young project, development starting in 2013, and work is done to continuously improve the product. Since most of the time was spent on making the software work, rather then on developing the most efficient implementation, it is clear that it is room for improving the performance of the code.

Because we are interested in the bandwidth performance, it is important to explore the part of the code that handles packet forwarding. As described in section 3, there are two modules that handle packet forwarding: IPOP-Tincan and IPOP-Tap. The former is the one that interacts with the physical interface and the process is mostly under the responsibility of libjingle which, considering that was developed by Google, was assumed to be scalable and was not investigated. IPOP-Tap interacts with the VNIC and the libjingle receive and send queues. It was the primary investigation target.

As it can be seen from figure 2, IPOP-Tap is implemented with two threads. The implementation of the threads is found in the ***packetio.c*** source file. In order to get a better understanding of the packet forwarding workflow, the process of receiving and sending packets with IPOP is illustrated in figures 11 and 10. The sending and the receiving thread are components of IPOP-TAP.

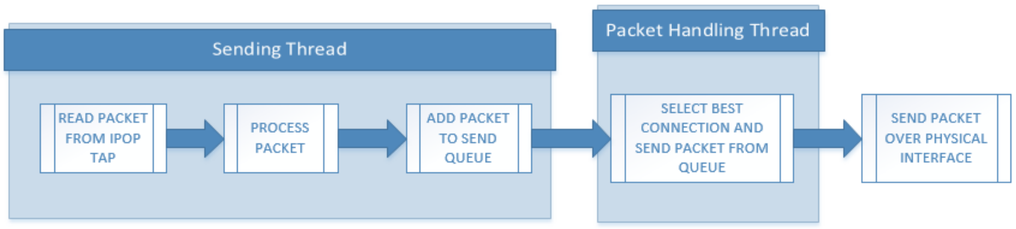
Sending Thread
Packet Handling Thread
READ PACKET FROM IPOP TAP
PROCESS PACKET
ADD PACKET TO SEND QUEUE
SELECT BEST CONNECTION AND SEND PACKET FROM QUEUE
SEND PACKET OVER PHYSICAL INTERFACE

Figure 10: Sending a packet in IPOP

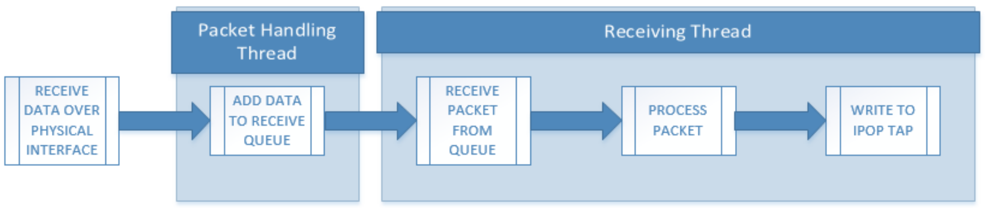
RECEIVE DATA OVER PHYSICAL INTERFACE
Packet Handling Thread
ADD DATA TO RECEIVE QUEUE
Receiving Thread
RECEIVE PACKET FROM QUEUE
PROCESS PACKET
WRITE TO IPOP TAP

Figure 11: Receiving a packet in IPOP

The sending thread reads packets that are generated locally from the tap interface and encapsulates them with the IPOP header and optionally performs IP translation. After encapsulation, it adds the packet to the sending queue for processing by the packet handling thread, part of IPOP-Tincan.

The receiving thread removes a packet from the incoming queue and optionally performs IP translation. The Ethernet frame destination address is updated with the local IPOP VNIC MAC address in order to be accepted by the operating system. The IPOP header is removed and the packet is sent to tap interface.

7.1 Serialization

It is apparent that both the receiver and sending threads introduce serialization in the packet forwarding process. There are three tasks that have to wait for each others completion before a new packet can be handled. In order to better perceive how time is distributed, code timing has been performed. This has been done using Linux function *clock_gettime()*, available in C in the *time.h* library [18]. It is the timing function with one of the best precisions available, up to nanoseconds. The results are illustrated in table 4.

Table 4: Code execution timing results

Sending Thread		Receiving Thread	
Read Packet from VNIC	*Process & Add Packet to send queue*	*Receive from Incoming Queue & Process*	*Write to VNIC*
2-4 μs	8-12 μs	2-4 μs	10-20 μs

As the table shows, there are sections in each thread that execute at a faster rate than others. More specifically, the writing operations occupy the largest amount of time. The times for reading and processing the packet have been merged because the measurements show that the processing time for the packet is negligible.

A proposed solution for solving the serialization problem is implementing the Producer-Consumer pattern in the discussed threads. Each thread is separated in two threads labeled as Producer and Consumer as follows:

- In the receiver thread:
  - The code responsible for receiving and processing packets from the incoming queue will act as Producer.
  - The code section that writes packets to the VNIC will act as Consumer.
- In the sending thread:
  - The code responsible for reading packets from the VNIC will act as Producer.
  - The code section that processes and adds packets to the send queue will act as Consumer.

The pattern works because, as we saw in table 4, the Producer code executes at a faster rate than the Consumer code. This observation assures us that the Consumer will never have to wait for packets from the Producer and will continuously execute its code routine.

The implementation has been done without mutexes, which is possible since there is only one Consumer and only one Producer. The downside is that two cores will always be used at 100% percent since the threads will continuously check if the buffer between the Producer and Consumer is empty or full. A possible refinement is the use conditional signals to announce the state of the buffer.

The results of the implementation have been measured and have shown to average at approximately 500 Mbps. Although improvements were expected, results show a slight decrease in average throughput. Timing measurements for the two threads were done and in order to identify the problem. It is apparent that the execution time of the Producer has increased on average. As a result there are multiple instances where the Consumer has to wait for packets to process. This is the probable cause of no gain in performance. It is not clear why the Consumer execution time is so high and the causes are to be investigated. Possible explanations are the use of extra structures and indirection in accessing memory and also the use of volatile variables, which are not saved in caches.

7.2 Asynchronous Writes

Another issue with the process shown in figure 11 is that of synchronous writes. Writing to the tap interface is the biggest bottleneck and it is important to improve on this issue. A proposed solution for increasing efficiency when writing on the VNIC is to use multiple asynchronous writes. This way the writes will be queued one after another and will immediately start after one finishes, without waiting for the Producer in the receiving thread to issue a new write.

In Linux there are two possibilities for executing asynchronous writes: *libaio*[19] and *POSIX AIO*[20]. The former is not available on any other platform other than Linux and it enqueues asynchronous reads and writes directly in the kernel, which should lead to good performance. The main drawback is that it does not works on all file descriptors. As discovered during testing, if used on socket file descriptors, libaio's I/O functions will fall back on synchronous behavior. For this reason, POSIX AIO was used. The implementation is described:

- The Consumer thread launches a number of asynchronous write requests.
- When one of them completes it will callback a write completion handler.
- The handler has the role of synchronizing multiple write requests with the Producer thread and also to create new write requests.

The results of this implementation were also tested and the throughput result was approximately 250 Mbps. Debugging the code and searching how POSIX AIO is implemented, the loss of throughput can be explained. The illusion of asynchronous AIO is achieved by having multiple parallel threads that execute in parallel blocking AIO. When a write is finished, a new thread is launched where the write completion handler will run. The overhead of creating a new thread every time a new write request is generated(i.e. whenever a packet is received) affects the performance of the code. A better solution that is to be explored is implementing the principle of POSIX AIO, with multiple threads in parallel, but without using a handler. This would reduce overhead but could introduce new complexity in synchronizing the threads.

8 Conclusions and Future Work

The investigation determined that encryption, used by default in IPOP, is a big source of performance overhead that decreases significantly the achievable throughput. As a solution, we implemented the feature of selective security, which effectively triples the communication bandwidth for connections that do not require encryption, while maintaining the security over links that do require it. This was done by leveraging the existing protocol messages and the changes are backward compatible with previous versions of IPOP.

Complete measurements have been done in order to determine the behavior of IPOP. Using multiple tools and methods, a profile of the code was achieved. This results are shared and can be used to further research the software and improve upon it.

Based on profiling results and code analysis, it was determined that there are issues with code that handles packet forwarding in IPOP. The identified problems are serialization and blocking synchronous writes, most affected being the receiver.As solutions, the Consumer-Producer pattern has been proposed for the former, while for the latter asynchronous writes is suggested. We implemented this solutions and discovered issues that reduce performance and offered possible resolutions.

This research focused mostly on IPOP code, without investigating in depth libjingle code, which is part of the forwarding process. It is interesting in future work to search for possible bottlenecks in this area and propose solutions. Future work also includes bug checking and solving the issues of the implementations described in this paper.

References

[1] Genc Tato Kyuho Jeong Renato Figueiredo Guillaume Pierre Kaveh Razavi Ana Ion and Thilo Kielmann. *Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure*. URL: `http://www.cs.vu.nl/~kaveh/pubs/pdf/ic2e15.pdf`.

[2] Openstack.org. *Home OpenStack Open Source Cloud Computing Software*. 2015. URL: `https://www.openstack.org/` (visited on 08/18/2015).

[3] Pierre St. Juste Kyuho Jeong Renato Figueiredo Youna Jung. *IP over P2P (IPOP)*. URL: `http://ipop-project.org/wp-content/uploads/2014/07/IPOP-WhitePaper-1407.pdf`.

[4] Xmpp.org. *The XMPP Standards Foundation*. 2015. URL: `http://xmpp.org/` (visited on 08/18/2015).

[5] Tools.ietf.org. *RFC 5389 - Session Traversal Utilities for NAT (STUN)*. 2015. URL: `https://tools.ietf.org/html/rfc5389` (visited on 08/18/2015).

[6] Tools.ietf.org. *RFC 5766 - Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. 2015. URL: `https://tools.ietf.org/html/rfc5766` (visited on 08/18/2015).

[7] Google Developers. *FAQ: Open Communications*. 2015. URL: `https://developers.google.com/talk/open_communications?csw=1` (visited on 08/18/2015).

[8] Google Developers. *Introduction to libjingle*. 2015. URL: `https://developers.google.com/talk/libjingle/developer_guide`.

[9] Xmpp.org. *XEP-0166: Jingle*. 2015. URL: `http://xmpp.org/extensions/xep-0166.html`.

[10] Ejabberd.im. *ejabberd — robust, massively scalable and extensible XMPP server*. 2015. URL: `https://www.ejabberd.im/`.

[11] GitHub. *ipop-project/ipop-tincan*. 2015. URL: `https://github.com/ipop-project/ipop-tincan`.

[12] GitHub. *ipop-project/documentation*. 2015. URL: `https://github.com/ipop-project/documentation/wiki/Running-GroupVPN-on-Linux`.

[13] Tools.ietf.org. *RFC 6347 - Datagram Transport Layer Security Version 1.2*. 2015. URL: `https://tools.ietf.org/html/rfc6347`.

[14] Linux.die.net. *mpstat(1): Report processors related statistics - Linux man page*. 2015. URL: `http://linux.die.net/man/1/mpstat`.

[15] Wikipedia. *OProfile*. 2015. URL: `https://en.wikipedia.org/wiki/OProfile`.

[16] Rotateright.com. *Zoom*. 2015. URL: `http://www.rotateright.com/zoom/`.

[17] Kernel.org. 2015. URL: `https://www.kernel.org/doc/Documentation/networking/nf_conntrack-sysctl.txt`.

[18] Linux.die.net. $clock_gettime(3) : clock/timefunctions{-}Linuxmanpage$. 2015. URL: `http://linux.die.net/man/3/clock_gettime`.

[19] Fsl.cs.sunysb.edu. 2015. URL: http://www.fsl.cs.sunysb.edu/~vass/linux-aio.txt.

[20] Ibm.com. *Boost application performance using asynchronous I/O*. 2015. URL: http://www.ibm.com/developerworks/library/l-async/.

9 Appendices

9.1 Appendix A

An example of the IPOP configuration file used for testing:

```
config.json
{
"ip4_mask": 24,
"xmpp_password": "09qqik46hpm0qhf3ivakmvfdvwd4k9",
"xmpp_host": "145.100.105.193",
"xmpp_username": "RP2_1@ejabberd",
"sec": false,
"tincan_logging": 0,
"ip4": "172.31.0.1",
"on-demand_connection": false,
"controller_logging": "DEBUG",
"stat_report": false,
"switchmode": 1,
"group": "test"
}
```

9.2 Appendix B

The profiling results and source code of the implementations described in the paper can be found at the following link:

https://github.com/dragosb91/Research-Project-2