IP over P2P (IPOP)

White Paper

07/18/2014

Renato Figueiredo, Youna Jung, Pierre St. Juste, Kyuho Jeong
ACIS Lab., University of Florida
http://ipop-project.org

History

DATE	AUTHOR	DESCRIPTION
OCT/24/2013	Youna	Structuring contents of white paper
NOV/12/2013	Renato, Youna	Update on contents
DEC/15/2013	Pierre	Revision of current content
DEC/17/2013	Pierre	Adding technical details
JAN/12/2014	Pierre	Updating section 5
JUN/11/2014	Renato, Pierre, Youna, Kyuho, Ken	Adding new features and bug fixes
JUL/18/2014	Renato, Pierre, Youna, Kyuho, Ken	Update content with new features and bug fixes for IPOP version 14.07.0

Table of Contents

1. Goals

1.1 Motivation

In the early days of the Internet, devices were able to directly communicate end-to-end with each other in an environment where cyber-security threats were uncommon. Much has since changed, and today the Internet presents an environment where security and privacy concerns are at the forefront, and where end-to-end connectivity among end users has been hindered by user mobility, IPv4 address space shortage and the widespread use of Network Address Translators (NAT) and firewalls.

Nonetheless, with the advent of cloud computing and online social networking (OSN), Internet users and applications increasingly require the ability to communicate end-to-end among peers with privacy and integrity (e.g. among personal mobile devices, or among cloud virtual machine instances deployed over multiple providers). Centralized services that mediate end-to-end user communication raise major concerns in privacy, fault-tolerance, and performance and are thus inadequate for many envisioned usage scenarios, for reasons including:

1) *Potential Privacy Loss/Leakage*- Centralized services such as online social networks (OSNs) allow users to communicate with their peers; because user-to-user communication goes through a centralized backend (e.g. Facebook), the service provider can view and store all interactions (e.g. wall posts, messages, photos, etc) conducted among social peers. A provider's privacy policy may change over time; thus many users worry about data that they would like to be restricted to a group of people (e.g. their friends) potentially being disclosed to others. In addition to OSN providers, centralized virtual private network (VPN) providers can also intercept and observe IP-level communications between peers. In contrast, IPOP follows an approach where peers communicate privately, end-to-end, effectively thwarting monitoring attempts by a centralized entity.
2) *Performance Limited by Service Providers*- In addition to privacy concerns, centralized services often constrain interactions between peers in order to scale to large numbers of users through the interfaces (APIs) exposed - for instance, by imposing limits in bandwidth, file sizes, and the types of interactions (e.g. only certain file types/sizes may be allowed). In contrast, IPOP follows an approach where peers are not constrained by any provider APIs or limits in their communication - the API that is exposed by IPOP is the standard IP network protocol, hence supporting applications that work over IP networks.
3) *Fault tolerance* - Centralized services can become unavailable to users due to outages, cyber-attacks, and government censorship, preventing user-to-user communication - even if there may be an Internet path between them. In contrast, IPOP supports direct peer-to-peer communication when there is an Internet path between the users.

In this context, IPOP has been designed to address the following requirements:

1) *User-defined and User-friendly Virtual Network Links* - Users must be able to define which devices to link to, using simple-to-use interfaces and leveraging well-adopted standards. IPOP allows users to establish relationships between devices they wish to connect using user-friendly social networking interfaces, through an OSN provider of their choice, or their own private service.
2) *Self-configuring Virtual Network Links* - Private links must be configured without exposing users/administrators to the challenges of configuring/exchanging security credentials, and managing endpoint IP addresses that may be private and change dynamically due to mobility and NATs.
3) *No Dependence on External Virtual Routing Infrastructure* - The virtual network must be able to encapsulate, tunnel, encrypt, and route packets at the endpoints themselves, without relying on the deployment of managed virtual routing infrastructure on the Internet.

1.2 Objectives

The vision for the IPOP project is to provide an open-source platform for user-centric Software-Defined Network (SDN), allowing end users to define and create their own VPNs connecting their own resources over the Internet.

The IPOP (IP-over-P2P) system creates an overlay virtual network supporting the vision that future Internet applications will increasingly demand direct, secure end-to-end P2P communication among devices. IPOP creates P2P links among endpoints, which can be mobile and/or NATed; provides virtual private IP messaging tunneled through P2P links as a core service; and enables the creation of peer-to-peer virtual private networks (P2P VPNs) of various topologies.

A key objective is to deliver the benefits of privacy, authentication and integrity in end-to-end communications to a wide range of users - including individuals, users/administrators aggregating resources across cloud infrastructures, and researchers studying next-generation cloud middleware. To accomplish this objective, IPOP requires no networking infrastructure deployment beyond endpoint resources, leverages user-friendly interfaces for configuration, and uses virtualization, supporting standard Internet protocols and existing applications.

Another objective of the project is to be community-driven and encourage contributions from developers. We use the liberal MIT license; the software development process follows open-source best practices, and the system has been designed in a modular fashion and incorporates standards and third-party OSS implementations to the extent possible, and is designed to run on a variety of platforms, including embedded systems, mobile devices,

personal computers, and cloud servers.

The resulting system is an overlay where every participating device has links to “social peers” (e.g. devices owned by friends in an online social network, or cloud-hosted computers that join a virtual cluster) and virtualization enables peers to communicate using IPv4/IPv6 standards, thus supporting existing, unmodified applications and allowing new applications to be developed using well-understood, widely-adopted Berkeley sockets interface. IPOP accomplishes this by implementing the core functionality of tunneling IP over peer-to-peer “TinCan” links and exposing a flexible API to control the setup and management of TinCan links to create various software-defined VPN overlays.

To address the requirements listed above, IPOP provides the following functionalities:

1) *IPOP allows users to define relationships through easy-to-use OSN interfaces*. These relationships can be among individual users, as well as group-oriented, supporting use cases including user-facing devices and friend-to-friend communication, as well as cloud servers connected as logical groups to form virtual clusters. They can be inherited from existing OSNs, or established in private/custom OSNs, and managed dynamically.
2) *IPOP automatically maps OSN relationships to configure and manage virtual network links*. This includes the generation and exchange of security credentials, setup of peer-to-peer links, virtual IP addresses, and overlay routing across multiple links.
3) *IPOP allows each endpoint device to not only pick and inject packets, but also provides a framework for overlay routing across multiple links*. To bootstrap itself, IPOP leverages external infrastructure/services to set up links - using standard protocols for messaging/notification (XMPP) and assistance in NAT traversal (STUN, TURN). Once links are established, IP-over-P2P messaging does not depend on the external infrastructure - it is done in a peer-to-peer fashion.

1.3 Use cases

There are several use cases for which IPOP provides useful features, as illustrated in the following examples. These examples highlight scenarios that use two of the IPOP “controllers” that have been developed thus far - ***SocialVPN*** and ***GroupVPN***. The functionality of these IPOP controllers is described in detail later in the white paper, but it is important to introduce their main characteristics at this point: in *SocialVPN*, each individual user, on his or her own, is able to determine which other individual users they wish to connect to by a VPN. In *GroupVPN*, instead of individual user-to-user relationships, a group relationship is established: each user in the group is able to communicate to all other users in the group, without having created a friendship link to other users. For instance, let us consider Facebook as a representative OSN to highlight differences between these two use cases. In SocialVPN, an individual user would only communicate with the Facebook friends they have explicitly accepted to join their social network. In GroupVPN, instead, a user with the role of group leader would create a Facebook group and invite users to join; any user who joins the group would automatically be able to communicate with all other users who also joined the group - even though they may not be Facebook friends to

each other. In essence, in GroupVPN, users delegate the establishment of trust to the group leader, whereas in SocialVPN, each user is in control of who they trust.

1) Distributed virtual clusters

Cloud users are becoming increasingly wary of vendor lock-in and expect the ability to painlessly move their workloads across cloud providers. Several projects are designed with multi-cloud deployment as a fundamental tenet. A motivation for IPOP is to facilitate such cross-cloud mobility by providing a virtual networking technology that requires little configuration and infrastructure.

For example, Alice runs a multi-tier web service consisting of a front-end server, application servers, and a database on a single cloud provider. For enhanced network security, Alice only assigns a public IP address to the front-facing web server and runs the application servers and database in a NAT-ted virtual network in the cloud. Alice then decides to move some of the application servers to a different cloud provider and restricts their Internet exposure with a similar NAT-ted network environment. Without IPOP, Alice needs to update all of the configuration files pointing to the application servers and is required to create port forwarding rules on the NATs at each cloud provider. She also has to ensure that all traffic between the front-end server and the application servers is encrypted because the VMs would be communicating over the open Internet.

Alice could potentially also use emerging SDN or overlay networking techniques to enable this cross-cloud migration; however, such solutions require networking expertise along with additional resources such as virtual switches/routers.

Instead, IPOP's GroupVPN can be used as the networking fabric to create virtual machine clusters deployed across multiple cloud providers (private and commercial) - without requiring any special support from the cloud providers, only the ability to run VMs. This allows software that runs on clusters (e.g. job schedulers, multi-tier Web frameworks) to seamlessly run across cloud providers, enabling greater flexibility in the management of workloads and costs by reducing concerns about vendor lock-in.

By running IPOP's GroupVPN in her virtual machines, Alice is able to maintain their private IP addresses because TinCan links are automatically re-created upon migration, even if the VM is behind a different NAT. IPOP node identifiers are preserved, and virtual IP addresses are decoupled from the physical infrastructure IP addresses.

2) Mobile computing and social network overlays

In the case of mobile computing, it is typically the case that a mobile user needs to communicate with trusted nodes to share either personal information (e.g. check-in at a restaurant), media (e.g. photos or videos), or computation (e.g. volunteer computing for mobile). For instance, many smartphones run a media server which facilitates media sharing with other devices in the same

LAN.

IPOP's SocialVPN creates a virtual LAN which makes it possible to extend this media sharing capability with social peers regardless of their location. As another example, through the VPN, users can make direct mobile SIP-based calls over the Internet using mobile softphone apps such as CSipSimple, thus enabling encrypted calls that are not logged by a centralized server.

With IPOP's SocialVPN, trusted peers are mapped to a virtual private IP address which is preserved as the mobile devices move across different networks (e.g from/to WiFi to 3G/4G). IPOP also supports IP multicast on within the virtual network; therefore LAN network discovery protocols such as UPnP and MDNS work out of the box.

IPOP's SocialVPN can be used as the communications layer to enable users to collaborate, share directly with friends over private end-to-end network links. This allows social peers to bypass the need to communicate through an online social network provider for privacy-sensitive or low-latency/high-bandwidth applications, while still benefitting from the ability to discover and establish friendships through an OSN provider.

2. Background

2.1 Software-Defined Network (SDN)

Recent developments in software-defined networking (SDN) have enabled unprecedented flexibility in the provisioning of elastic cloud services by data center providers. In SDN-enabled technologies, a user (typically a network administrator) is given the ability to program the behavior of network fabric (typically switches and routers) through a standard, programmatic interface (e.g. OpenFlow).

Yet, several use case scenarios require users to deploy virtual networks that span across multiple cloud providers, mobile device endpoints, and are subject to device mobility and VM migration. In these scenarios, prevailing SDN techniques are challenging to deploy, because no single entity has the capability to program SDN devices end-to-end - for instance, mobile Internet users Alice, Bob and Carol may wish to create a software-defined network connecting their devices together but do not have the authority to configure any networking equipment other than their own devices.

The techniques currently available for typical "data center SDN" technologies do not apply in a straightforward way when the endpoints are personal devices managed by multiple end users and connected by the public Internet. In contrast, IPOP enables setup and management of private end-to-end tunnel links allowing virtual networks over shared infrastructure. It reuses existing standards and implementations of services for discovery and notification (XMPP), reflection (STUN) and relaying (TURN), facilitating configuration with an approach where trust

relationships maintained by centralized (or federated) services are automatically mapped to end-to-end overlay links.

2.2 Overlay Virtual Networks in Cloud Computing

Over the past few years, major IaaS cloud providers have introduced network virtualization capabilities allowing users to create their own isolated virtual network and define IP address ranges and subnets on the cloud.

IaaS vendors, such as Amazon EC2,Windows Azure, and Google Cloud Engine, also enable additional features such as specifying DHCP and DNS setting for the private network. Moreover, users can define routing rules and network access control for the network and IPSec VPN gateways which make it possible to combine multiple different subnets from private or public clouds.

It is clear that the cloud computing industry understands that network virtualization is a crucial component for cloud provisioning; however, there is no open standard for interoperability, thus placing the entire burden on users desiring cross-cloud deployments.

To address challenges in network virtualization across different clouds, various third-party commercial solutions have emerged. VMware NSX is a network virtualization technology that runs at the hypervisor level, recreates the whole network in software at both layers 2 and 3, and also supports Xen and KVM. It uses a virtual switch in the hypervisor to connect to other virtual switches, virtual bridges or virtual routers, while only requiring an IP backplane for connectivity. It also supports virtual networking across different data centers since the virtual networking components connect over IP. However, this solution is difficult to support across multiple providers, as it requires privileged access to the hypervisor.

Both VNS3 and RightScale's Cloud Management products lets users provision virtual machines in the same virtual private network across different public cloud providers through a common interface. VNS3 runs a virtual appliance manager at each cloud provider and implements a virtual switch/router, and a VPN gateway in the appliance; hence, VNS3 is not dependent on the underlying cloud provider's virtual networking technology because it re-implements its own in the cloud on top of the IP backplane. RightScale provides a unified wrapper around the virtual networking API of various cloud providers and greatly simplifying the deployment of virtual networks spanning multiple public clouds. However, these third-party solutions require additional resources to configure and manage these networks, again placing a significant burden of configuration and management on end users. While this burden may be acceptable in environments where dedicated staff is employed to manage the virtual network components, it becomes a significant barrier for small/medium-scale deployments - a typical use case of clouds. IPOP targets the needs of users who are not willing to afford the configuration and management of additional virtual network infrastructure.

Academic and industry research have also explored applicable solutions for cross-cloud virtual networking. Researchers at IBM have developed VirtualWire, which implements a layer 2 virtual network tailored to deployment of legacy applications and VM migration across clouds. Virtualwire is a hypervisor-level virtual network integrated with the Xen-Blanket nested virtualization technology, enabling VM migration across public clouds. VIOLIN uses a very similar approach to Virtualwire providing layer 2 communication with networking components such as switches and routers implemented purely in software. A drawback with these approaches is that users are still required to configure virtual switches, routers, and deploy their own DHCP and DNS servers within the virtual network.

Another solution is CloudNet which advocates MPLS-based VPNs to bridge virtual networks and provide layer 2 connectivity across different cloud providers. However, this approach requires public cloud vendors to expose compatible MPLS-based VPN gateways and layer 2 access to their networking virtualization technologies. Major public cloud providers do not support layer 2 connectivity.

VNET also provides layer 2 connectivity across private clouds and it is implemented at the hypervisor level. This is accomplished through a layer 2 proxy that bridges the two networks but this approach would not work on public clouds since access to the hypervisor and layer-2 networking is unavailable to users.

All of these previous works do not explicitly deal with NATs and firewalls, and assume the availability of VPN gateways and virtual routers with public IP addresses. As cloud usage increases, the pool of IPv4 addresses become more scarce --- compounded by recursive virtualization and the use of containers --- establishing end-to-end virtual network links across NAT-constrained devices becomes increasingly important.

VINE is a layer 3 virtual networking alternative which supports NAT/firewall traversal through relaying. However, it requires users to manage and configure the virtual routers if an application server is migrated across clouds, and does not provide NAT-traversed end-to-end tunnels that bypass a relay/router node.

OpenVPN is a solution that is applicable in both cross-cloud VPN environments and mobile virtual networking. However, OpenVPN follows a client/server architecture where all IP traffic is routed through a central gateway. This incurs high latency and creates a resource bottleneck.

Many other solutions improve on the OpenVPN model; for instance, Hamachi uses a proprietary central server to setup P2P connections between hosts, even through NATs and firewalls. IP traffic is tunneled over these encrypted P2P connections.

Other approach such as Tinc, Vtun, and N2N all create mesh VPNs where nodes create direct connections to each other, but they require nodes to be openly accessible over the Internet. While these solutions can potentially be used to enable cross-cloud virtual networking, they are

not currently supported by mobile platforms, and do not provide a flexible overlay architecture that supports other VPN topologies, such as those implied by friend-to-friend social network graphs.

3. Architecture and Features

3.1 Overall Architecture

From a developer's perspective, IPOP consists of three major modules depicted in Figure 3.1:

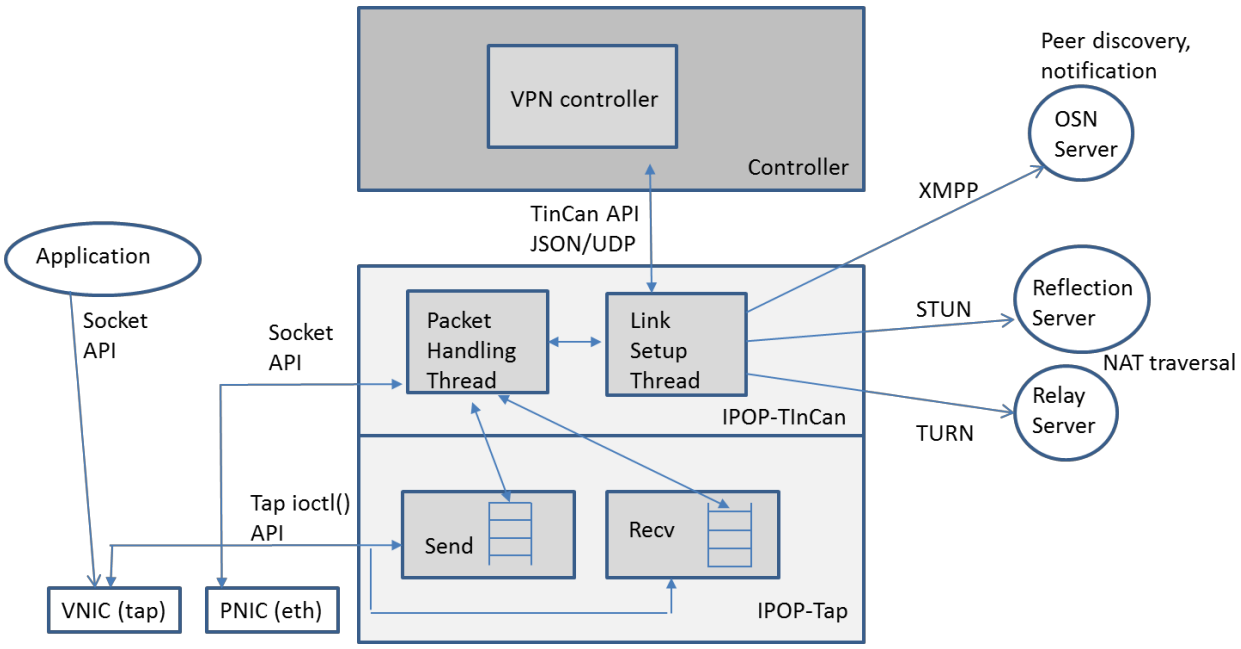


Figure 3.1: Major IPOP modules: IPOP-Tap, IPOP-TinCan, and Controller

IPOP-Tap: this is the module that interfaces with a virtual network interface (VNIC) to be able to pick/inject IP packets from/to the virtual network. It is responsible for maintaining send/receive queues, and using the system call interface of the O/S (e.g. Unix ioctl()) to configure and read/write from a virtual network interface (tap device). It also handles the encapsulation of an IP packet by adding the required IPOP headers for routing over the P2P links.

IPOP-TinCan: this is the module that handles links between pairs of IPOP nodes. Specifically, it manages each "TinCan" link that a node has. A "TinCan" link is a private end-to-end connection between two peers through which the virtual network's IP traffic is tunneled. IPOP-TInCan handles the setup/tear-down of each link (link setup thread), and the sending/receiving of tunneled IP packets over these links (packet handling thread). The link setup thread uses external services through the XMPP protocol to discover and notify peers for which TinCan links are to be formed, and STUN/TURN protocols to establish links between nodes that are

constrained by NATs (Network Address Translators). It also exposes a management API to the controller module. The packet handling thread uses the IPOP-tap module to interact with the virtual network, and the Berkeley sockets API to send/receive tunneled packets through the physical Internet.

Controller: this module is responsible for configuring and controlling the setup and management of a collection of IPOP-TinCan links to form overlays, using the API exposed by the IPOP-TinCan module. The controller is responsible for establishing the policies for topology creation of an IPOP overlay (e.g. on-demand topology in GroupVPN, or social graph topology and multi-hop routing in SocialVPN), and determining when links are created/destroyed (e.g. when a peer node's presence is detected, or on-demand triggered by IP traffic), using the IPOP-TinCan mechanisms exposed by its API to implement the policies.

3.2 Features

1) Supporting unmodified TCP/IP applications through P2P tunneling

It helps to understand the architecture and the functionality of the major modules of IPOP by looking at the system from different perspectives. Starting from the perspective of applications (Figure 3.2), the most important benefit of network virtualization in IPOP is that it supports existing, unmodified applications. From the application's perspective, IPOP creates a virtual private network supporting existing socket APIs exposed by the O/S through a virtual network interface (VNIC) and end-to-end tunneling, such that existing applications (IPv4 or IPv6 based) can execute without requiring modifications, and new applications can be deployed using well-known Berkeley socket APIs. For performance reasons, IP tunneling in IPOP typically takes place across a single TinCan link, but multi-hop routing over several TinCan links is also supported in the architecture

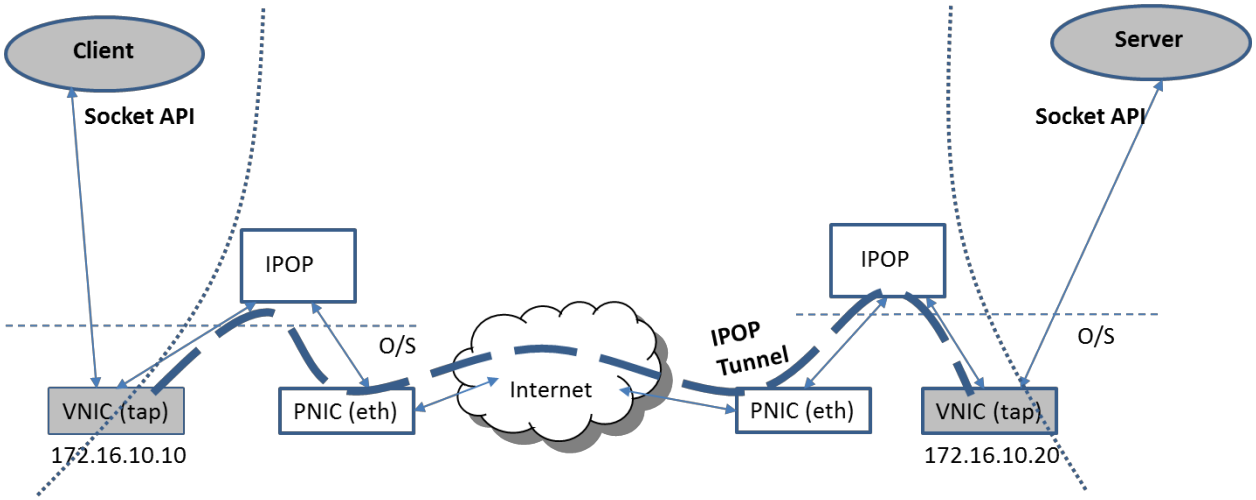


Figure 3.2: IPOP exposes a virtual network to applications, thereby supporting the standard Internet IP protocol and existing, unmodified applications

2) Leveraging Social Relationship in IPOP

In order for IPOP to provide the virtual network's perspective illustrated above, it is necessary for users to establish who they want to link with in their virtual network, and to configure and deploy IPOP software on their devices. Figure 3.3 illustrates the process for users to establish relationships, and how IPOP utilizes XMPP messages to establish TinCan links.

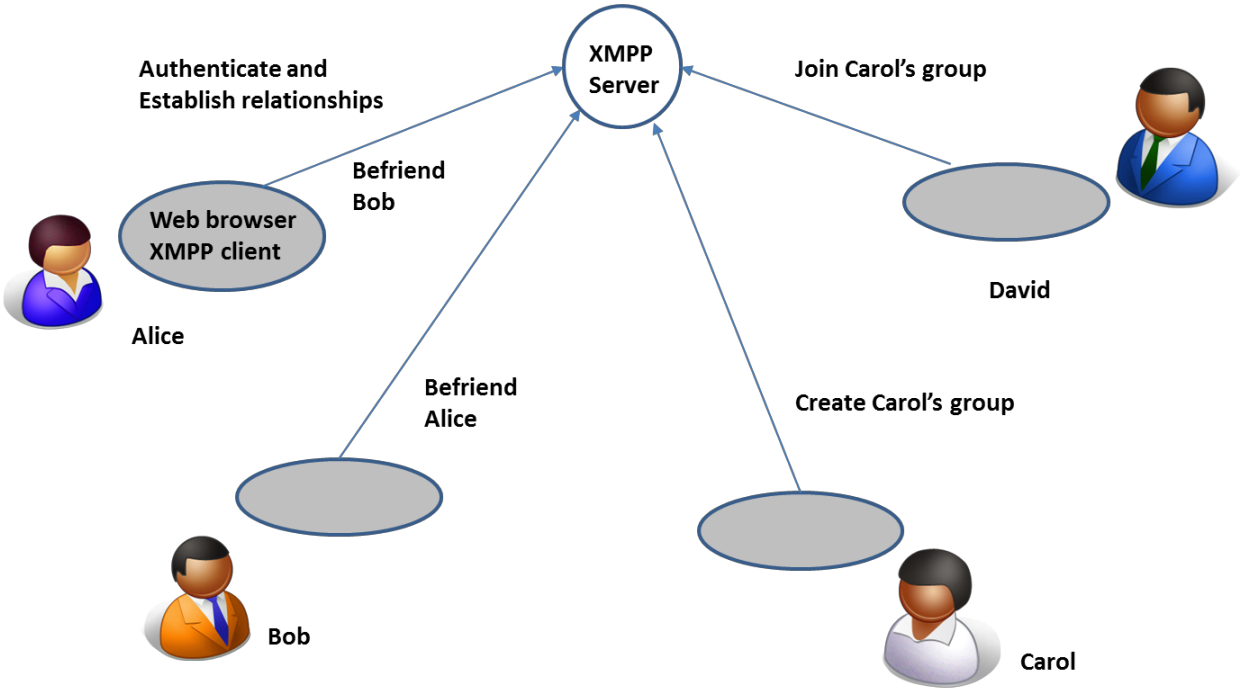


Figure 3.3: User establish relationships to create point-to-point links to other users, or establish groups, through an online social network (OSN) supporting the XMPP protocol.

In order to configure IPOP virtual networks, the user needs to determine which users they wish to connect to. This is done through an online social network interface - independently from using IPOP. In the typical case, a user creates/authenticates to an account in an online social network (OSN) server; this could be a public service (e.g. Google hangout), or a private service (e.g. a private ejabberd server), through a Web interface or XMPP client (e.g. Pidgin). The user establishes relationships with other users they wish to connect to through the OSN (e.g. with friend requests). These can be peer-to-peer (e.g. in IPOP's SocialVPN), or based on groups (e.g. in IPOP's GroupVPN). Currently, IPOP supports the XMPP protocol to query OSN relationships, and to send messages to online peers.

3) Easy Deployment with Minimum Configuration

The IPOP software typically runs on a user's personal computer, or on virtual machines deployed on cloud resources. Once peer relationships are established through an OSN server,

a local configuration file at each IPOP endpoint points to the OSN server, and the user (or system administrator) simply executes the IPOP software on the resources that are to be connected to the virtual network (Figure 3.4). IPOP then automatically installs and configures the local VNIC, and automatically creates end-to-end tunnels connecting to VNICs of peers determined by the OSN.

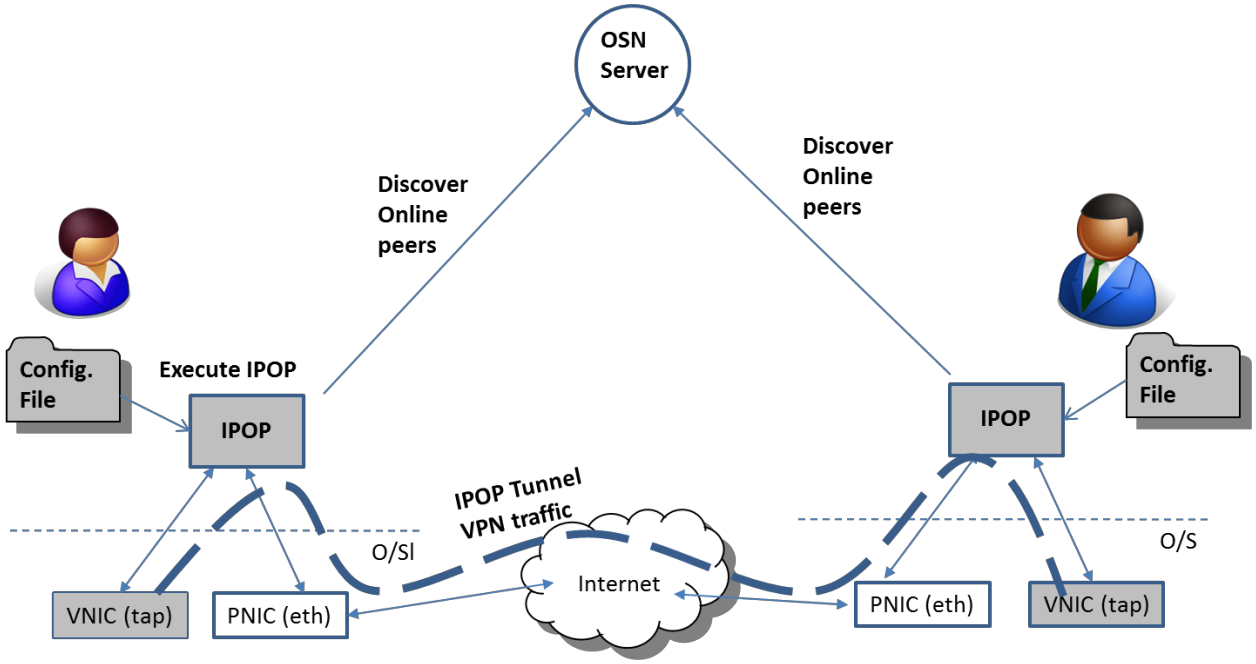


Figure 3.4: A user or system administrator provides a configuration file determining, among other parameters, which OSN server to connect to; upon running IPOP, TinCan links are autonomously created to tunnel IP traffic with end-to-end privacy and integrity.

4) Leveraging Existing STUN/TURN protocols

One of the key aspects of IPOP that enables it to transparently tunnel traffic between endpoint devices is its ability to traverse NATs. This is of key importance as IPv4 address space exhaustion, and desire for private address spaces as a line-of-defense against attacks have contributed to the proliferation of NAT devices in personal and enterprise networks. NATs complicate the process of creating end-to-end VPN tunnels, as resources behind distinct NATs are not directly addressable by each other.

IPOP leverages the libjingle library to perform NAT traversal in two major ways (Figure 3.5): IPOP leverages STUN/TURN protocols to discover their NAT endpoints and create tunnels directly with peers, if possible (“cone-type” NATs), or through an intermediary relay on the public Internet when more restrictive NATs prevent direct tunnels (“symmetric” NATs). The selection of a tunneling approach is managed dynamically by IPOP, and tunnels are completely transparent to applications.

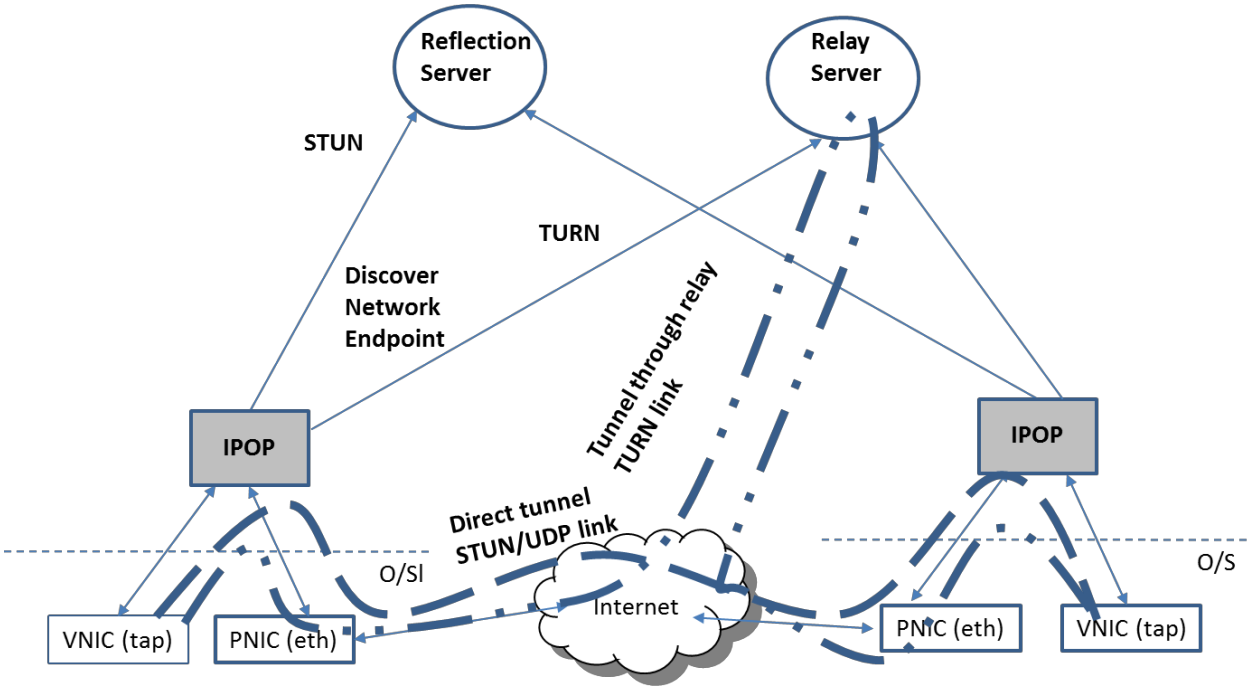
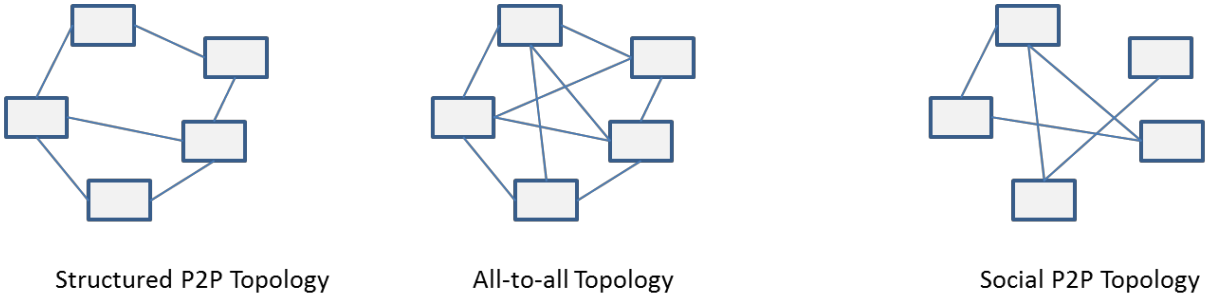


FIgure 3.5: IPOP supports the creation of direct TinCan links between endpoints for most deployed “cone” NATs. Peers use one (possibly out of many) STUN servers to discover their endpoints on the public network, and exchange endpoint information using XMPP. For certain restrictive “symmetric” NATs, IPOP uses TURN and relay servers to route through an intermediary node on the public Internet.

5) Supporting different overlay topologies

The discussion of the architecture thus far has focused on establishing single links between peers that wish to communicate. This is the common-case scenario in IPOP, as the goal is to connect peers that wish to communicate directly to each other, over a fast path. Thus, IPOP preferentially creates TinCan links that leverage the underlying Internet path between the two endpoints. However, it is possible that the direct Internet path between two endpoints cannot be used (e.g. because of restrictive NATs), or because of resource capacity or performance reasons (i.e. a node may be limited in terms of how many concurrent links it can maintain). Therefore, IPOP supports a framework upon which multi-hop overlay routing can be performed over TinCan links. This is illustrated in Figure 3.6.



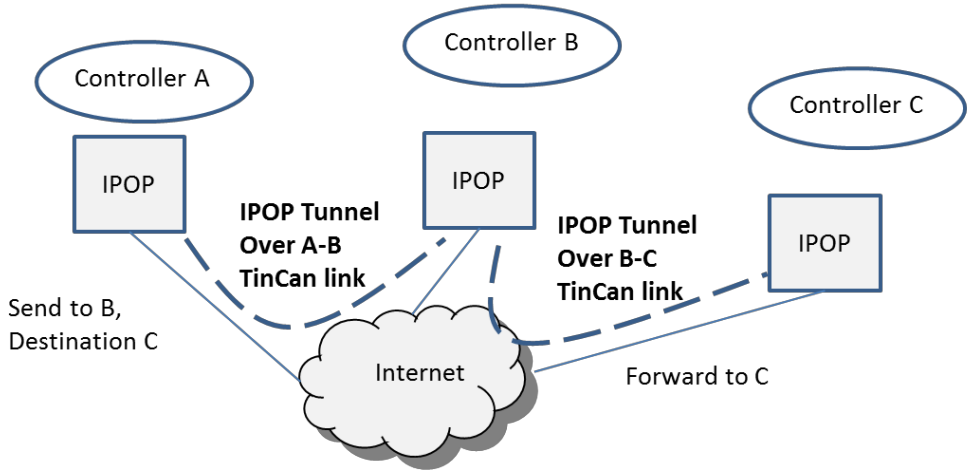


FIgure 3.6: IPOP enables different controllers to implement different overlay topology maintenance and routing for different usage scenarios. Each controller binds to an IPv6 VNIC with a unique address in the overlay, allowing identifier-based routing through multiple intermediaries. Neighboring controllers can communicate using transports layered upon IPv6, thereby facilitating the programming of controller-controller protocols.

The key insight is that IPOP does not prescribe a particular overlay topology (and associated routing); these are left to the implementation of controllers, reflecting the fact that different uses of IPOP may be best served by different topologies. For instance, for a small-sized virtual cluster (few to tens of nodes), an all-to-all topology with direct connections among each pair of peers offers good performance and fault tolerance with very simple topology maintenance and routing; however, it does not scale to large numbers of nodes. A structured P2P topology can scale to much larger numbers, but requires more complex maintenance and routing. An unstructured social network graph topology scales well, is resistant to Sybil attacks, and provides good connectivity among friends and friends-of-friends; however, routing across users who are distant in the graph is difficult because a path must be first discovered.

Because different use cases can be best served by different topologies and routing policies, IPOP provides core mechanisms and abstraction layers upon which controllers can be designed. The controller module in IPOP is thus responsible for implementing the policies that control topology creation and management. To support overlay routing, IPOP exposes the

abstraction of a virtual IPv6 private address space - where each node is assigned a unique IPv6 identifier - and core primitives for forwarding along TinCan links.

Specifically, the IPOP-TinCan module has a local forwarding table and is capable of resolving a virtual IP address to a unique node identifier (UID). If a packet is destined to a neighbor at the other side of a TinCan link, IPOP-TinCan simply forwards the packet along the link, without the involvement of the controller. However, if the destination is not reachable directly by a TinCan link, the IPOP-TinCan module forwards the packet to its local Controller module for overlay routing. Controllers use this primitive to initiate overlay routing; controllers can do so by using IPv6-based transport protocols to communicate with a neighboring controller (over a TinCan link) to forward a packet, and to uniquely identify the destination. It is also possible for applications to create their own application-layer overlay routing by using the basic mechanism of forwarding to neighbors over a TinCan IPv6 link.

3.3 Technical Details

1) IPOP-Tap

This component deals directly with the operating system by configuring the virtual network interface (VNIC) with an IP address and netmask specified by the controller. Currently, this is done using the ioctl() system call on Linux-based systems such as Ubuntu/Debian, Android, and OpenWRT. For Windows, this configuration is done using the netsh command. This module also maintains the file descriptor used to read and write Ethernet frames to/from the VNIC. The VNIC is created through the use of the kernel tap module available for both Linux and Windows.

IPOP-Tap operates with two separate threads which perform IP packet encapsulation on all outbound packets and optionally perform IP translation on inbound packets (e.g. for SocialVPN). In the sending thread, IPOP-Tap encapsulates every outbound IP packet read from the VNIC by prepending a 40-byte IPOP header. This header consists of a 20-byte source UID followed by another 20-byte destination UID. The source UID is the UID of the local node, and the destination UID is determined by using the destination address in the IP header to look up the corresponding UID from the peerlist table, which maps peer UIDs to IPv4 addresses. After performing the encapsulation, IPOP-Tap adds the packet to the outgoing queue for processing by the upper layer (i.e. IPOP-Tincan).

In the receiving thread, IPOP-Tap optionally performs packet translation upon reading an inbound IP packet from the incoming queue connected to the upper layer. The IPOP header of the received packet is examined and the source and destination UIDs are used to update the IP header of the packet - since each peer may maintain their own different UID-to-IP mappings (e.g. in SocialVPN), inbound packets need to have their IP header updated to ensure consistency with the local peerlist. Hence, for every inbound packet, the destination IP address is set to the local node’s IP address, and the source IP address is set by looking up the corresponding IP address mapped to the destination UID in the IPOP header. After the IP header has been translated, the

destination MAC address of the Ethernet frame is updated with the local VNIC's MAC address to guarantee that the operating system accepts the packet. The IPOP header is then removed and an Ethernet frame (including the IP packet as the payload) is sent to the VNIC via the tap kernel module's file descriptor.

IPOP-Tap exposes the following API allowing the IPOP-Tincan module to configure the VPN:

1. tap_set_ipv4_addr(const char* ip4_str, int ip4_mask): this function sets the IPv4 address and netmask of the VNIC in the local system (e.g. 16 means w.x.y.z/16)

2. tap_set_ipv6_addr(const char* ip6_str, int ip6_mask): this function sets the IPv6 address and netmask of the VNIC in the local system (e.g. 16 means w.x.y.z/16)

3. peerlist_set_local_p(const char* uid_str, const char* ip4_str, const char* ip6_str): this function sets the local IPv4/IPv6 addresses for the local user. This is needed for IP encapsulation and translation.

4. peerlist_add_p(const char* uid_str, const char* ip4_str, const char* ip6_str): this function sets the local IPv4/IPv6 addresses for a remote user. This information is stored in the peerlist table and is used for IP encapsulation and translation.

5. set_subnet_mask(int subnet_mask): this function sets the subnet mask for the router mode in GroupVPN (the router mode in GroupVPN allows a device (e.g. an OpenWRT wireless router) to run GroupVPN and route for all devices within a LAN). It is set to 32 when routing for one node, 31 for two nodes, 24 for 255 nodes, and so on.

2) IPOP-TinCan

This module does the heavy lifting necessary to enable direct, encrypted P2P connections among peers. It relies on the libjingle library for three main capabilities: XMPP support, P2P connection establishment, and OpenSSL-based socket encryption. The first crucial task performed by this module is the establishment of a TLS connection with an XMPP provider (e.g. Google Hangout, Jabber.org, or a private ejabberd server). IPOP-Tincan also creates an X.509 certificate every time the process is started. It then uses the XMPP service as a trusted, out-of-band overlay to share the local peer's X.509 fingerprint with other trusted peers along with Interactive Connection Establishment (ICE) protocol information necessary to bootstrap a secure P2P connection. Using the XMPP roster (or buddylist) feature, IPOP-Tincan is able to discover online friends and send connection requests/replies to each other. IPOP-Tincan also manages the connections to online friends.

Each friend possesses a unique identifier (UID, a 20-Byte long identifier) and each P2P connection is mapped to a peer's UID. IPOP-Tincan uses the UID of each peer for forwarding packets to the appropriate P2P connection. Our current design uses two blocking queues (an outgoing queue and an incoming queue) to move packets between the IPOP-Tap and

IPOP-Tincan modules. As described earlier, IPOP-Tap reads an Ethernet frame from the VNIC, encapsulates it with a 40-byte IPOP header, and puts it in the outgoing queue. IPOP-Tincan pulls the encapsulated packets from the outgoing queue and uses the destination UID in the IPOP header to look up the corresponding P2P connection for that UID. If a P2P connection for that UID exists, the encapsulated packet is sent over that connection; otherwise, the packet is sent "up" to the IPOP-Controller for processing.

When a UID is matched and a packet is sent over a TinCan P2P connection, the IPOP-Tincan module on the receiving end calls its incoming packet handling function, which reads the destination UID from the IPOP header to verify that it matches the local node's UID. If the UIDs match, then IPOP-Tincan puts the received packet in the inbound queue for processing by IPOP-Tap. IPOP-Tap will in turn get the incoming packet, perform IP translation on the IP header, update the MAC address and write the Ethernet frame to the VNIC. IPOP-Tincan performs all of this processing in its packet handling thread.

IPOP-Tincan performs two additional functions in a separate link setup thread. First, it handles all messages coming from the XMPP service and sends them to the IPOP-Controller for processing. Second, it listens on a UDP socket for incoming JSON RPC requests from the IPOP-Controller. For example, when a node connects to the XMPP service, it announces itself to every social peer in its XMPP roster (or buddylist). IPOP-Tincan receives these notifications and sends them to the IPOP-Controller. The controller receives these notifications and decides the appropriate action based on its policies (e.g. connection creation, or user notification). IPOP-TinCan exposes an API over the JSON RPC interface that allows a controller various capabilities including: creation and deletion a P2P connection, registration with an XMPP service, and messaging to XMPP. The IPOP-Tincan API also exposes the IPOP-Tap functions which allows for the configuration of the local network, and assignment of UID-to-IP mappings. IPOP-TinCan also sends various notifications to the controller as well including node joins, dead link detection, and XMPP login errors. Section 4 describes the TinCan API in more detail.

3) IPOP-Controller

The IPOP-Controller is the most extensible portion of the design. It uses the TinCan API (described in Section 4) to control various aspects of IPOP-TinCan's behavior. The controller implements various policies such as criteria for link creation, limit on the number of connections, network configuration settings, IP allocation scheme, and link deletion parameters. Also, the controller can be written in a scripting language - such as Python - which enables fast prototyping of different policies and functional/performance isolation from the IPOP routing core.

IPOP-TinCan also sends packets with destination UIDs that not mapped to a P2P connection "up" to the controller for processing. This is a key feature that enables overlay routing through the controller when a direct TinCan link to the destination UID is not available. Upon receiving the encapsulated IP packet, the controller may take various actions: typically, either trigger the creation of a new TinCan link to the destination UID, or forward the packet to another controller for overlay routing. Since IPOP-Controllers can also forward messages to each other over IPOP

connections, they can implement distributed policies by coordinating among themselves. For instance, it is possible to implement a policy that allows them to arrange themselves as a structured overlay network or a DHT. Since controllers can use the IPv6 virtual network primitive to communicate with “neighbor” controllers, as well as use XMPP messages to send notifications to other controllers, they have the flexibility to determine the most efficient organization for a given deployment scenario. A line-by-line breakdown of a very simple “template” controller is described in Section 5.

Currently, IPOP implements two different controllers: SocialVPN and GroupVPN.

A. SocialVPN controller

The SocialVPN controller creates TinCan links between social peers in a manner where each user has their own view of the network - i.e. with SocialVPN, a user Alice’s devices link to all of her devices and to all devices of friends she has explicitly added to her social network. Alice’s SocialVPN does not have TinCan links to any device that does not belong to herself or a friend. Because social networks have very large numbers of users, and IPv4 private address spaces are limited in size, SocialVPN implements an address translation mechanism whereby each user has a local virtual private subnet (e.g. 172.16.10.0/24) and maps all their friends onto this subnet. IPv4 addresses are automatically mapped (just addresses - ports are not mapped) by IPOP-TinCan. SocialVPN also supports private IPv6 addresses; because of the larger 128-bit address space, these can be considered unique with very high likelihood, and not translated.

B. GroupVPN controller

The GroupVPN controller creates TinCan links among nodes in a manner that all devices belonging to a GroupVPN can connect to each other. In GroupVPN, rather than having each user determine independently who to connect to (as in SocialVPN), a group is formed, and each user who belongs to the group is able to communicate with any other user. All nodes in the GroupVPN are bound to the same virtual private address space, and IPv4 addresses are not translated. This is useful in applications such as virtual clusters, LAN-style gaming, etc, where it is expected that all nodes joining the network are able to communicate with each other.

In GroupVPN, there are two approaches to the creation of TinCan links: proactive, and on-demand. In proactive mode, a GroupVPN node creates TinCan links to all online peers as soon as their presence is detected. This allows for fast linking among nodes, but does not scale to large networks (typically hundred or more nodes); thus, it is intended for small-scale groups. In on-demand mode, a GroupVPN node only creates TinCan links to online peers when IP packets are sent between nodes. Thus, on-demand mode only creates a link when there is demand for communication; it also trims links after a configurable period without communication. On-demand mode scales better, but incurs longer latencies (orders of seconds) to create links.

GroupVPN also allows operation in “Router mode”. In this mode, GroupVPN runs on a router device (e.g. an 802.11 wireless router patched with OpenWRT), and serves IPOP addresses to all devices connected to the router. This is useful when you want to have IPOP endpoints that do not run the IPOP software locally (e.g. iphones, TVs, etc). The GroupVPN router provides NAT

and DHCP functionality, allowing it to be in many cases a drop-in replacement for a residential wireless router that also provides connectivity to other GroupVPN routers.

3.4 Advanced Features

1) Switchmode IPOP

This mode only works in GroupVPN IPOP. Bridges are widely used as network solution for cloud computing - for instance, when creating multiple virtual machines or O/S containers on a single server using the OpenStack software, these virtual machines/containers are connected by a Linux bridge. Attaching IPOP's tap device to Linux bridge provides a capability that allows an IPOP GroupVPN where multiple virtual network interfaces share the same Linux bridge. This feature can be used in various cloud network environments and lead to more efficient deployments where the overhead of packet handling/forwarding in IPOP is avoided for communication that takes place among VMs/containers within a single server. For instance, IPOP can run on a physical cloud node providing IPOP network connectivity to multiple VM instances in cluster. Moreover, many virtual machine systems such as VMware or VirtualBox uses tap devices of which can be attached to Linux bridge alongside with IPOP. Multiple VM instances can access IPOP network without consuming any computing resources.

We named this mode of operation "switchmode", as IPOP runs as a virtual layer-2 switch for the ARP protocol, and provide attached virtual interfaces a gateway to access remote peers. Conventional IPOP tap take up partial address range in O/S. In switchmode, IPOP can share this address range with other virtual network interfaces or virtual bridges such as Linux bridge.

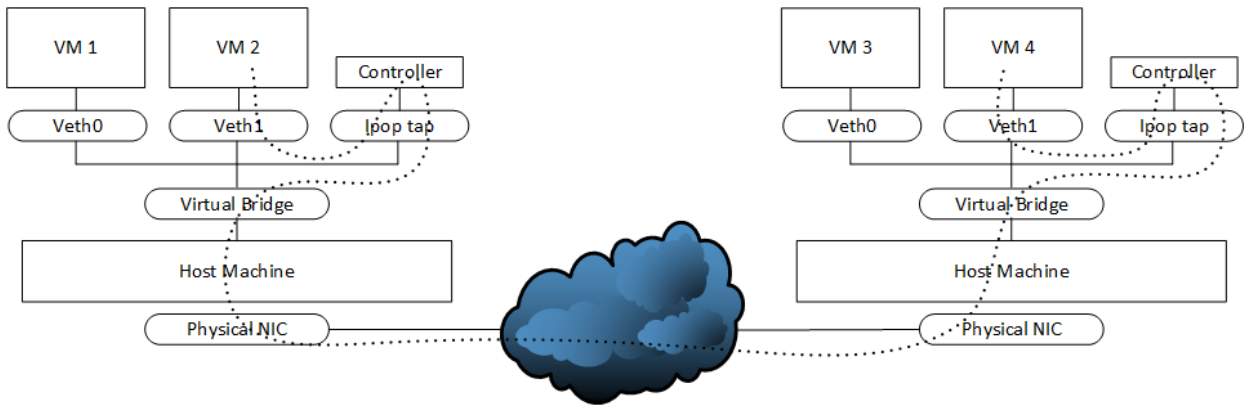


Figure 3.7. IPOP Switchmode

As depicted in Figure 3.7, the IPOP tap device can be attached to virtual bridges, such as Linux bridge. VM1 can send IP packets to VM2, and vice versa, without requiring IPOP to process these packets (hence, without network virtualization overhead) as these are connected through Virtual Bridges. IPOP also allows VM1 or VM2 to send IP packets to VM3 and VM4. These

packets are captured by “IPOP tap” and sent over IPOP overlay network and destined to VM3 or VM4. Thus, virtually, VM1, VM2, VM3 and VM4 are in the same “flat” virtual network subnet - without NAT or firewall - as if they were all connected through the same switch device as depicted in Figure 3.8.

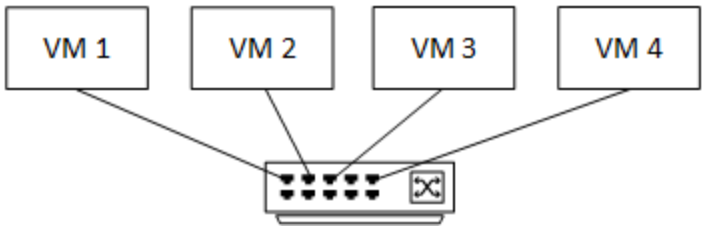


Figure 3.8. Switchmode from the perspective of virtual machines VM1-VM4

Switchmode is implemented by emulating local ARP request/reply message as a remote call. For example, a local ARP request/reply message is re-formatted in JSON, sent to an IPOP remote controller through a remote call; then it notifies remote peers with the information of which IP address belongs to which IPOP peers. These IPOP peers (and associated IP address mappings) are kept in every IPOP peer.

One benefit of IPOP switchmode over running IPOP in an setup such as multiple VMs or multiple containers in a host, is that the VM do not have overhead of running IPOP inside as its process. In the future, IPOP can be incorporated in openflow controllers, with the possibility of moving the overhead of processing packet translation to physical network devices from the host machine processes, which can further reduce the overhead of processing packets.

2) Multihop Routing

IPOP also has a feature of routing/hopping through multiple nodes in overlay network. This mode is currently only supported in the SocialVPN controller and with IPv6 virtual addresses.

One of the challenges of a social-network based overlay topology is that the overlay is unstructured, and the addressing of node cannot be utilized in a straightforward way to make routing decisions. For example, addressing in a tree-like topology allows the use of subnet information to make decisions in a hierarchical fashion based on IP address identifiers. In structured P2P topologies, such as DHT-based overlays, the address of a node can be used to make forwarding decisions by each peer based on a greedy approach - i.e. forwarding messages to the peer that is closest in the identifier space to the destination. To the best of our knowledge, there is no proper addressing scheme for social-network-like overlay topology which enables conjecture location of node by its address. To resolve this issue, we have implemented an approach inspired by schemes that are widely used in sensor/wireless networks. The

addressing itself is done in a random manner, simply assigning random IPv6 value to each node. However, for routing, we make use of limited-scope flooding to locate the destination node, and support source-routing and distributed routing. Flooding does not scale to very large networks; however, in SocialVPN, our goal is to support routing over a small number of “friendship hops” - e.g. friends and friends-of-friends.

A. Lookup request/reply

When an IPv6 address that is not mapped to any TinCan UID is captured in IPOP tap device, the packet is sent “up” to the controller, as described earlier in this white paper. Then, if the controller does not known the packet’s target address, the source (say “S”) uses limited multicast flooding to send a lookup request messages to a range of neighbors within a “hop distance” - this range is configurable, with a default value of 3. All the nodes within this hop ranges receive the lookup request messages; if a node (say “T”) matches the target address among their direct peers, it sends back a lookup reply message which contains the history of nodes visited (so it can be routed back to the source).

Figure 3.9 depicts this process. In Figure 3.9 (a), source node “S” sends lookup request message to all its direct peers 1, 2, 3 and 4. The message is similarly forwarded to each peer by these four nodes, until the hop limit is reached or the target node is in the list of peers as in Figure 3.9 (b, c). Thus, the node which received the lookup request message either sends back look up reply message (when it has target in its direct peers) or it floods lookup request message with 1 less hopping count to its direct peers. In figure 3.9 (d), lookup request message reaches the node 5, since node 5 has D as its direct peers it sends lookup reply message all the way to the source. Note that, node 7 also has D as its direct peers and it sends lookup reply message all the way back to S, but this lookup reply message is ignored by S sinces its routing has a larger hopping count.

If there is no lookup reply message within certain time range, the source node floods lookup request message again with increased hop limit, until it reaches the maximum allowed hop limit.

All the lookup request/replies used in the multi-hop routing protocol are sent from a controller to a neighboring controller over an IPv6 virtual link tunneled through TinCan.

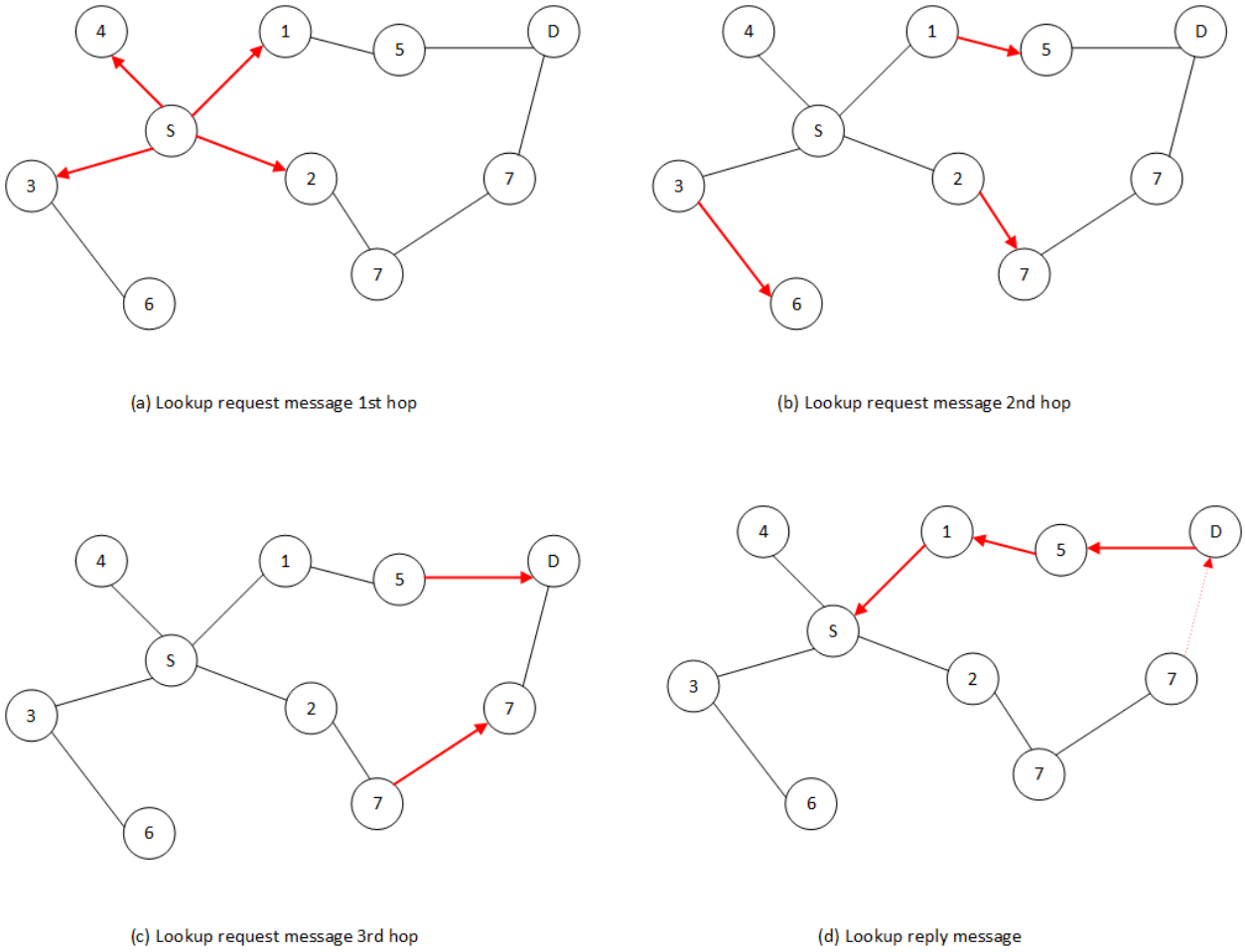


Figure 3.9. Lookup request/reply message

B. Source routing

In multihop mode, there are two alternative ways to manage routing information. One is *source-based routing* and the other is *distributed routing*. In source-based routing, every hopping address is listed in the header. Figure 3.10 shows the structure of source-based routing packet. The packet architecture is conventional TVL style as in common in data communication. “Count” is the total count of hopping node and “Index” is the location of current hopping node. Index increases by one every hopping and when it reaches the “Count”, the packet is arrived at destination. Thereafter, count number of IPv6 address attached and IP packet itself follows.

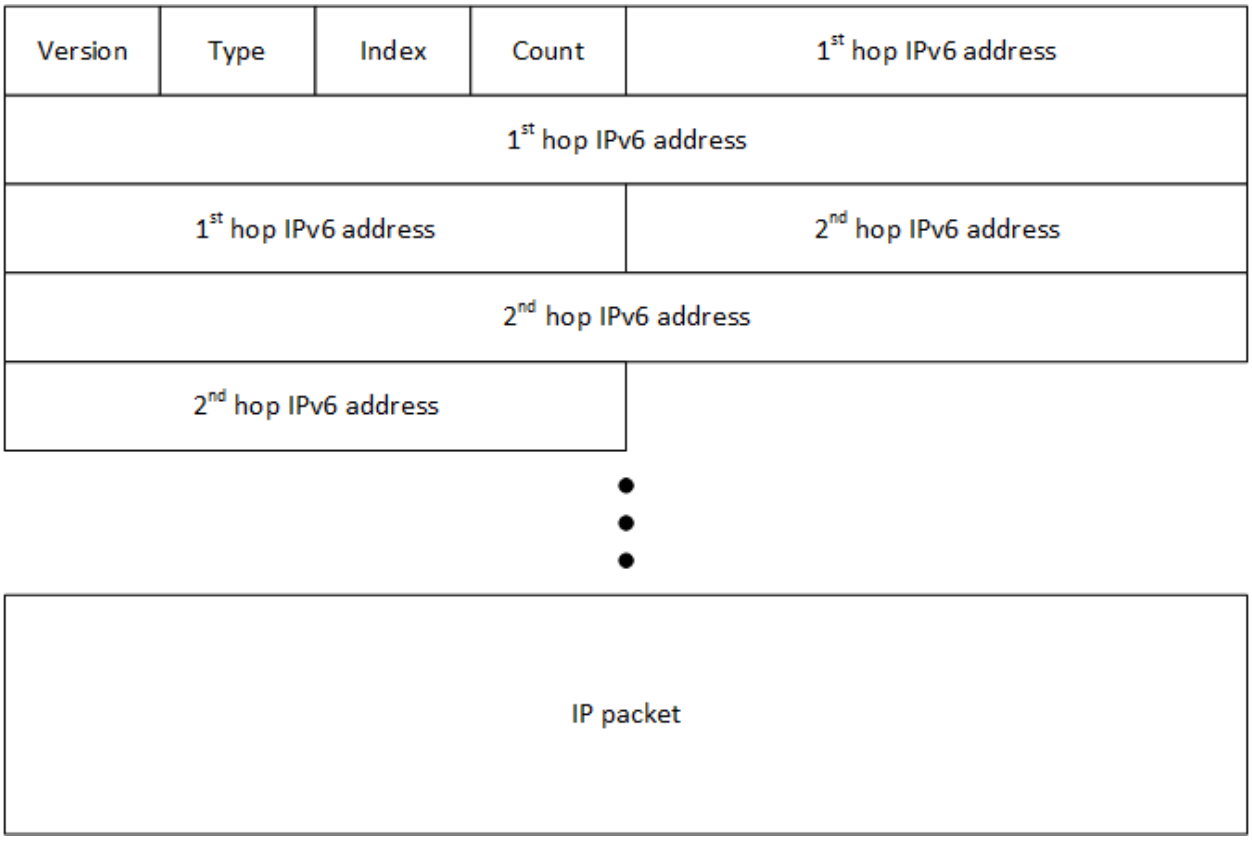


Figure 3.10 Source routing header

Routing information such as destination address with associated hopping node addresses are stored in source node. Thus intermediary nodes does not store routing information. In case of link failures, the intermediary nodes notify back to the source node that the link is broken. Then, the source node start lookup request/reply phase to find alternate multihop route.

C. Distributed Routing

In distributed routing, all the routing information distributed across the IPOP overlay network, so that the header does not have to contain all the routing information. But, it lacks a mechanism to notify back to the source when there is a link failure.

3) External services and protocols

IPOP relies on a series of open standards and technologies for various aspects of its functionality. First, it uses the Extensible Message and Presence Protocol (XMPP) as the trusted, out-of-band medium for bootstrapping secure connections. XMPP also enables peer discovery and notification in the network. To ensure P2P communication between nodes behind NATs, the libjingle library uses the Simple Traversal Utilities for NATs (STUN) protocol. This service allows peers to discover their public IP and port from behind NATs. We call this a reflection service. Finally, for nodes behind symmetric NATs with routers that cannot be

traversed with STUN, the libjingle library leverages the Traversal Using Relays around NATs (TURN) protocol. A TURN relay with a public IP address serves as a middleman routing IP packets between two nodes behind symmetric, untraversable NATs. Our use of open standards ensure a modular design where we can reuse existing mature tools available on the Internet.

A. XMPP (Peer discovery/notification)

Before a P2P connection can be established between two nodes, the nodes have to discover each other and exchange information such as their current public IP addresses and ports and X.509 fingerprints. This exchange has to occur through a trusted medium that allows for generic messaging. XMPP serves exactly that purpose and it is widely available as an open standard with many open-source implementations. By using XMPP's presence and query stanzas, IPOP-Tincan is able to send messages through XMPP services without any additional features or requirements. For our deployments, we tested our design on Google XMPP servers, Jabber.org servers, and even our own XMPP server through the use of ejabberd deployments on Amazon EC2. We have provided instructions on how to independently deploy an ejabberd XMPP service on our wiki.

B. STUN (Reflection/NAT traversal)

Most users connect to the Internet from behind a NAT device, which enables multiple machines to share a single IP address. As a result, most applications are not aware of their public IP address and port. Since a P2P connection is a direct IP connection between two devices, peers need to be aware of each others public IP addresses and ports in order to successfully create IPOP-Tincan connections. A STUN server enables such a capability. The ejabberd XMPP server has a built-in STUN server that allows peers to discover their public IP and port. Therefore, the same instructions about deploying a ejabberd XMPP service will also provide a STUN service by default. Google also runs a series of STUN servers for their XMPP and WebRTC protocol. A list can be found here.

C. TURN (Relay/NAT traversal)

According to Google statistics, about 8% of nodes cannot create direct P2P connections to each other due to symmetric NATs; in such extreme conditions, traffic relaying is the only option. This is accomplished through the TURN service. The job of a TURN server basically involves taking data from one peer and forwarding it to another. Since the TURN server has a publicly reachable IP address, nodes behind symmetric NATs can naturally connect to it. Using this middleman, two nodes behind symmetric NATs can communicate. Deploying a TURN server must be done with care because that machine will essentially be routing IP packets between two nodes therefore proper limits and restrictions have to be set in order to ensure the nodes does not become a network bottleneck. We provide the details of a TURN deployment on our wiki.

4. Controller/TinCan API

IPOP-Tincan provides a JSON-RPC API over UDP which enables developers to extend IPOP's functionality without making changes to the link management/forwarding core. These API calls

do not return results; however, some of them trigger a notification to get sent to the controller (e.g. “get_state” causes IPOP-Tincan to send update notifications to the controller). These function calls and notifications are described below.

[Function Calls]

4.1 register_svc

This function is designed to allow IPOP-Tincan to register to a backend service which will be used to discover social peers and bootstrap encrypted P2P connections. We currently only support XMPP service, hence this is basically the XMPP username and password. Future versions may extend this API to support services other than XMPP, and authentication mechanisms to XMPP other than username/password.

Parameter	Type	Description
username	string	the username used to login to the XMPP service (e.g. username@gmail.com)
password	string	the password used to login to the XMPP service
host	string	the host name or IP address of the XMPP server (e.g. talk.google.com)

4.2 set_local_ip

This call configures the VNIC and set the UID of the local peer. This call is important and has to be done first because it also triggers the creation of a X.509 certificate and configures the virtual NIC and operating system for packet flow.

Parameter	Type	Description
uid	string	the unique identifier for the local peer (e.g. hexdigest of the sha1 hash). The uid, encoded as a hexadecimal string, is 40 Bytes long.
ip4	string	the IPv4 address for the ipop tap virtual NIC
ip6	string	the IPv6 address for the ipop tap virtual NIC
ip4_mask	integer	the prefix length, in bits, for the network mask in the IPv4 address (e.g. 16 means a.b.c.d/16 allowing for a network of size 2^16)
ip6_mask	integer	the prefix length, in bits, for the network mask in the IPv6 address. This is usually set to 64
subnet_mask	integer	the prefix length for the network mask in the IPv4 address of the local router. This parameter is relevant for router

		mode - in router mode, an IPOP-Tincan node routes packets for more than one IP address; this parameter specifies the subnet it should route for. If not in router mode, you must set this parameter to 32 (a netmask of 32 gives a network size of 1, i.e. IPOP-TinCan only routes for its local address).

4.3 get_state

This is a very important call because it allows developers to query the state of IPOP-TinCan. In return, IPOP-TinCan replies (through the use of notifications) with the state of the local peer and each remote peer. The first notification sent to the controller is the “local_state” (see section 4.10) which tells the controller state information about the local user - such as UID and IPv4 address. Afterwards, a separate “peer_state” notification (see section 4.11) is sent to the controller for each remote peer containing information such as IPv4/IPv6 address and online status. The controller can provide the UID of a particular peer; in that case, only the “peer_state” of the specified user is returned to the controller. If no uid is provided, then the “peer_state” of every remote peer is returned to the controller.

Parameter	Type	Description
uid	string	the unique identifier of the remote peer that we want state information about. If this is an empty string, then all state is returned.

4.4 create_link

This call creates P2P TinCan link with a peer, with encryption if so specified. It requires that the controller provides the 40-byte hexadecimal encoded UID of the remote peer, and the peer’s X.509 fingerprint, along with optional STUN/TURN credentials. The X.509 fingerprint is generated by IPOP-TinCan once the `set_local_ip` call is made. The `get_state` call returns the X509 fingerprint as part of the local_state notification.

Parameter	Type	Description
overlay_id	integer	IPOP-TinCan will support multiple overlays in the future (e.g. XMPP, Gnunet) so we need an ID for each overlay. Currently, we only support two hardcoded IDs (0 for controller, 1 for XMPP service). For now, always set this to 1.
uid	string	every user has a unique identifier that is determined by the controller. This identifier has to be a 40-byte long hexadecimal string. In the GroupVPN controller we use the sha1 hash of the IPv4 address; in the SocialVPN controller, we use a random identifier.

fpr	string	this is the X.509 fingerprint (hash of X.509 certificate) of the user. This is obtained from the `get_state` api call described below. IPOP-TinCan generates an X.509 certificate once the `set_local_ip` call is made.
stun	string	this parameter specifies the STUN server that will be used for this connection (e.g. stun.google.com:19302). You can provide an empty string for this parameter, but the connection will only succeed if one of the two peers are not behind a NAT.
turn	string	this parameter specifies the TURN server that will be used for this connection (e.g. ip-of-turn-server:port). This can be an empty string and is only required if both peers are behind symmetric NATs.
turn_user	string	the username for accessing the TURN server, can be empty string.
turn_pass	string	the password for accessing the TURN server, can be empty string.
cas	string	a specifically formatted string that contain a list of IP addresses and connection credentials necessary for bootstrapping an ICE connection. This is generated by IPOP-TinCan.
sec	boolean	this parameter specifies whether or not the connection should be encrypted.

4.5 set_remote_ip

This call builds the forwarding table for IPOP-TinCan. It maps IP addresses to remote peer UIDs. The aforementioned “create_link” function causes IPOP-TinCan to create a P2P connection associated with the remote peer’s UID. This function thus allows IPOP-TinCan to know which P2P connection to forward the IP packets.

Parameter	Type	Description
uid	string	the unique identifier of the remote peer
ip4	string	the IPv4 address that the remote peer is mapped to. The address needs to fall within the subnet of the VNIC defined in the create_link call.
ip6	string	the IPv6 address that the remote peer is mapped to. It should fall within the subnet of the ipop virtual NIC defined in the create_link call.

4.6 trim_link

This trim call provides a mechanism to force the removal of TinCan P2P links from a node. This occurs regardless of whether peers are online or offline. Link trimming is important in resource-constrained devices, and to create scalable routing overlays.

Parameter	Type	Description
uid	string	the unique identifier of the remote peer specifying which P2P connection to trim.

4.7 set_cb_endpoint

IPOP-Tincan currently notifies the controller of two main events: 1) a connection request/reply and 2) a link state change (i.e from online to offline). This call registers the endpoint at the controller that IPOP-Tincan sends the notifications to.

Parameter	Type	Description
ip	string	the IP address of the controller
port	integer	the port number of the controller

4.8 send_msg

This call allows the controller to send an arbitrary message to another peer via the overlay (i.e. XMPP service) specified by the overlay_id. This serves as a secondary out-of-band channel to bootstrap P2P connections. Note: this call is not intended to be used for the datapath flow of IP packets between nodes - it is a control path call.

Parameter	Type	Description
overlay_id	integer	this parameter specifies which overlay should be used to send the message. This should be set to 1 because we only support send_msg over the XMPP overlay.
uid	string	the unique identifier of the remote peer that will receive the message

4.9 set_logging

By default, IPOP-Tincan prints various debugging messages to stdout. This call allows developers to select the level of logging.

Parameter	Type	Description
logging	integer	sets the logging severity, 0 = no logging, 1 = error logging, 2 = info logging

4.10 set_switchmode

This mode configures whether IPOP should handle ARP messages picked from the tap interface to the controller or not. If set to enabled (1), ARP request/response messages are forwarded to controllers, and the controller broadcasts this message as JSON formatted RPC to all the other IPOP peers looking for given IP address. If found, remote ARP reply message is received and local ARP reply message is generated to solicit this IP address. In the operating system's perspective, this IP address is associated with IPOP tap device. Then, IP packets can be transferred through IPOP network as if they were in the same network layer. As a result, when IPOP tap device is attached to a linux bridge, it works as a gateway toward IPOP network for the all the network interfaces that attached to the linux bridge. If this mode is disabled, IPOP discards ARP messages. It is disabled by default.

Parameter	Type	Description
switchmode	bool	Switchmode is enabled by setting this parameter as 1.

4.11 set_translation

This function allows the controller to enable/disable TinCan's IPv4 address translation feature, which is necessary for SocialVPN. The SocialVPN controller sets this value to 1, while the GroupVPN controller set this value to 0. In SocialVPN, IPv4 addresses are not globally unique and translation is necessary to map all friends' IPv4 addresses to a local address.

Parameter	Type	Description
translation	int	If set to 0, then you are running on GroupVPN mode, if set to 1 then you are running in SocialVPN mode

4.12 set_trimpolicy

If it is enabled, relay connections are removed among connection candidates, if there are STUN connections available.

Parameter	Type	Description
trim_enabled	bool	If set to true, relay connections are trimmed once a STUN connection candidate is available

4.13 echo_request

This function call works as a ping message allowing an external process to monitor whether TinCan is responsive or not. Once TinCan receives this message, it sends back "echo_reply" message to the controller.

Parameter	Type	Description

msg	string	The message contents send back to controller in “echo_reply” message.

4.14 echo_reply

This message is complementary to “echo_request”. Upon the receipt of “echo_request” message, the expected behavior for both the TinCan and controller is to reply back with “echo_reply” message echoing the same “msg”.

Parameter	Type	Description
msg	string	The “msg” contents has the exactly same payload with the “echo_request” msg.

4.15 set_network_ignore_list

As a default, IPOP tries to create STUN connection by using all available network interfaces. This can be inefficient and also produce much verbose logging messages by trying to create connection through network interfaces without public internet access. This call allows the controller to identify network interfaces that should be ignored.

Parameter	Type	Description
network_ignore_list	string	List up network interfaces to be ignored in json format. i.e., “network_ignore_list” : [eth1, wlan0]

[Notifications]

Notifications are sent by IPOP-TinCan and received by the controller in several events of interest.

4.16 local_state

This notification is sent to the controller as a result of the get_state function call. It contains information about the local node.

Parameter	Type	Description
type	string	set to **local_state** to indicate that it contains information about the local node
_uid	string	the UID of the local node
_ip4	string	the IPv4 address of the local node
_ip6	string	the IPv6 address of the local node
_fpr	string	the X.509 certificate fingerprint of the local node

4.17 peer_state

This notification is sent to the controller as a result of the get_state function call. It contains information about a single peer node.

Parameter	Type	Description
type	string	set to peer_state to indicate that it contains information about the peer node
uid	string	the UID of the peer node
ip4	string	the IPv4 address of the peer node
ip6	string	the IPv6 address of the peer node
fpr	string	the X.509 certificate fingerprint of the peer node
status	string	set to either **online** or **offline**. Online is defined as the TinCan P2P link is active - i.e. keep-alive messages sent over the TinCan link have been acknowledged by the destination. If no link has been created, or if keep-alive messages have not been received within a period of time managed by libjingle (15 seconds by default), the status is offline.
security	string	set to either **none** or **dtls**. It indicates whether the P2P connection is encrypted or not
stats	string	This attribute only available in GroupVPN. It lists connection candidates and traffic records. Candidates are in array form and attributes of candidates are listed below. Attributes are associative array form in JSON format. ● best_conn: True, if this candidate is best connection. ● local_addr: reflective or relay transport address of candidates. In most cases, it is the public address of NAT with ports that the IPOP is binding to. ● local_type: Local IPOP node NAT traversal type ● new_conn: True, if it is new connection ● readable: True, if this connection is readable ● recv_bytes_second: Byte received for last one seconds ● recv_total_bytes: Total byte received since the connection is created ● rem_addr: reflective or relay transport address of candidates remote peer. It is usually the public transport address of NAT that the remote peer is binding to ● rem_type: remote peer NAT traversal type ● rtt: Round trip time ● sent_bytes_second: Byte sent for last one seconds

		• sent_total_bytes: Total byte sent since the connection is created • timeout: True, if this binding timed out. • writable: True if writable
xmpp_time	int	the number of seconds since the local node receives an XMPP presence message from the remote peer. Presence messages serve as an indicator that the node is ready to accept TinCan connections. The controller can use the XMPP time to determine when to trigger a connection request to a node

4.18 con_stat

This notification is sent to the controller when a TinCan P2P link changes state either from unknown to online or offline.

Parameter	Type	Description
type	string	set to con_stat to indicate that it contains connection status information.
uid	string	the UID of the peer node
data	string	set to the current status: online, offline, or unknown. Unknown means the connection has just been initialized

4.19 con_req

This notification is sent by a controller through the XMPP overlay when a node wants to initiate a TinCan connection with a remote peer. The controller determines the policy for connection creation. This connection request should trigger a connection respond on the receiving controller if it agrees to create a connection with the requestor.

Parameter	Type	Description
type	string	set to “con_req” to indicate that it is a connection request
uid	string	the UID of the peer node
data	string	contains the X.509 fingerprint of the peer node and optionally followed by ICE information containing public IP address, credentials, and connection type

4.20 con_resp

This notification is sent back through the XMPP server as a reply to the connection request event described above. The response message is almost identical to the connection request except for the message type. Once a connection response is received, then nodes can start creating their TinCan connections.

Parameter	Type	Description
type	string	set to con_resp to indicate that it is a connection response
uid	string	the UID of the peer node
data	string	contains the X.509 fingerprint of the peer node and followed by ICE information containing public IP address, credentials, and connection type

5. Inter-Controller Message Types

Each controller runs UDP server at its IPv6 address with default port number 30,000. Inter-controller messages communicate through this transport address using conventional UDP. The packet structure simply have two-byte header prepending the Payload. TinCan control message, IP packet or source route IP packet are demultiplexed simply by “type” field in the header.

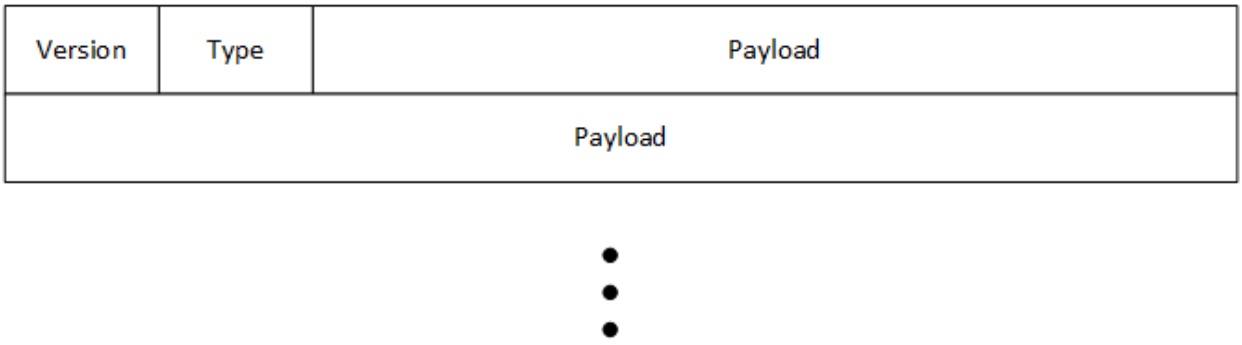


Figure 5.1 Inter-Controller Message Type

5.1 TinCan control message

The type field of TinCaan control message is 0x01. Payload of TinCan control messages uses JSON format. These messages are used to share information among controllers or send control messages.

1) arp_request

This message is flooded through the IPOP overlay network for the purpose of finding “target_ip” address.

Parameter	Type	Description
msg_type	str	set to “arp_request”
target_ip	str	Target IPv4 address

2) arp_reply

This message is sent back to the sender of arp_request message notifying that the target_ip address is associated with its own alongside with its UID and IPv6 address.

Parameter	Type	Description
msg_type	str	set to “arp_reply”
target_ip	str	Target IPv4 address
uid	str	UId of IPOP node that associate with target
ip6	str	IPv6 address of IPOP node that associate with target

3) lookup_request

This message is flooded from source node to the range of ttl nodes notifying that source node is looking for the target.

Parameter	Type	Description
msg_type	str	set to “lookup_request”
target_ip6	str	IPv6 address
via	str[]	Array of IPv6 address of hopping nodes. IPv6 address every hopping node is attached to this field one by one at every hopping.

ttl	integer	Time to live. This field decreases by one at every hopping. When it treaches 0, the lookup_request message stop flooding.

4) lookup_reply

Lookup reply message is sent back to all the way to the source node.

Parameter	Type	Description
msg_type	str	set to “lookup_reply”
target_ip6	str	IPv6 address
via	str[]	This field contains all the history of hopping nodes.
via_idx	integer	The initial value is -2 and decremented by one from target to source denoting the current location from via list. This field is used for indexing the current hopping node among via list.

5) route error

This message notifies to the source node that there is a link failure in route to target.

Parameter	Type	Description
msg_type	str	set to “route_error”
via	str[]	This field contains all the history of hopping nodes.
index	integer	This field is used to denote the current hopping nodes. This field decreases one at every hopping. When it reaches 0, this message reaches the source and source node evicts the routing information and sends lookup_request message again.

5.2 TinCan data packet

The type field of TinCan data packet is 0x02.The payload is the whole IP packet.

5.3 TinCan source route packet

The type field of TinCan source route packet is 0x03. The packet structure is shown in FIgure 3.10 in section 3.3.5.2.

6. Controller Workflow

In this section we explain with more detail the organization of an IPOP-Controller to give a better

understanding on the setup required to get IPOP-Tincan to work. This example uses a Python-based controller; other languages can also be used to write IPOP controllers.

6.1 Create UDP socket for JSON-RPC

The IPOP-Controller communicates with IPOP-Tincan over a UDP socket and the messages are serialized using the JSON format. Therefore, the first step is to create a UDP socket and register that UDP socket as the callback endpoint for notifications. This is accomplished with the following code snippets from the Python controller:

```
self.sock = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
self.sock.bind(("", 0))
do_set_cb_endpoint(self.sock, self.sock.getsockname())
```

6.2 Set local node UID and IP addresses

The controller has to determine a policy for creating globally unique 160-bit (20-byte) identifiers. In our code snippet below, we just use a random number generator from the operating system. We also convert the 20-byte identifier into a hexadecimal string. The controller also determines the appropriate IP address to set for the VNIC. In SocialVPN mode, the controller can choose any IP address and subnet that does not conflict with the local machine's network settings. In GroupVPN mode, the controller has to ensure that the IP allocation scheme does not cause any IP conflicts with other nodes in the VPN network.

```
ip4 = "172.31.0.100"
uid = binascii.b2a_hex(os.urandom(CONFIG["uid_size"]/2))
do_set_local_ip(self.sock, uid, ip4, gen_ip6(uid))
```

6.3 Login to the XMPP service

As described earlier, the XMPP service is a crucial component for discovering other nodes in the network and exchanging necessary information for creating secure P2P connections. Therefore, the controller has to provide the XMPP server and login credentials to the service before it can connect to other peers in the network.

```
do_register_service(self.sock, user, password, host)
```

6.4 Periodic update notifications and maintenance

The IPOP-Tincan API does not return results. It only sends back callback notifications when certain events occur. For example, if the "register_service" call fails due to an improper password, this will trigger an XMPP error notification sent to the controller. Therefore, in order to receive update notifications (see sections 4.10 and 4.11) from IPOP-Tincan, the controller has to explicitly request them using the "get_state" function call. Our controllers periodically calls "get_state" in order to keep track of node and connection status in the network. If the updated state information tells us that a connection is offline, our code deletes the connection with the "trim_link" function which basically closes down the connection and releases its resources.

```
while True:
        server.serve()
        time_diff = time.time() - last_time
        if time_diff > CONFIG["wait_time"]:
            count += 1
            server.trim_connections()
            do_get_state(server.sock)
            last_time = time.time()


def trim_connections(self):
        for k, v in self.peers.iteritems():
            if "fpr" in v and v["status"] == "offline":
                if v["last_time"] > CONFIG["wait_time"] * 2:
                    do_trim_link(self.sock, k)
```

6.5 Notification Processing

Since the controller registers a UDP endpoint with IPOP-Tincan, it has to listen on that UDP socket for incoming notifications from IPOP-Tincan. As documented above, there are five types of notifications issued by IPOP-Tincan. The “local_state” and “peer_state” notifications contain information about peer connections, IP addresses, and packets sent. The “con_stat” notification occurs when a connection goes from “unknown” to “online” or “offline”. The “con_req” and “con_resp” notifications indicate a node’s desire to create a P2P connection. Each of these notifications are processed accordingly.

```
def serve(self):
        socks = select.select([self.sock], [], [], CONFIG["wait_time"])
        for sock in socks[0]:
            data, addr = sock.recvfrom(CONFIG["buf_size"])
            msg = json.loads(data)
            logging.debug("recv %s %s" % (addr, data))
            msg_type = msg.get("type", None)

            if msg_type == "local_state": self.state = msg
            elif msg_type == "peer_state": self.peers[msg["uid"]] = msg
            elif msg_type == "con_stat": pass
```

6.6 Connection Creation

When the controller receives a “con_req” or “con_resp” message, it can decide to create a P2P connection. The data field of the object is a space-delimited string consisting of the remote peer’s X.509 fingerprint, and a list of candidate addresses (i.e. IP and port information with ICE credentials). In the code snippet below, when a new request arrives, the controller always calls

the create_link function which will attempt to create a new P2P connection. IPOP-Tincan will only create a new connection if one does not currently exist. The controller also determines an IP allocation scheme for each new incoming connection. In this example, each new connection is assigned by simply incrementing the last allocated IP address by one.

```
elif msg_type == "con_req" or msg_type == "con_resp":
                fpr_len = len(self.state["_fpr"])
                fpr = msg["data"][:fpr_len]
                cas = msg["data"][fpr_len + 1:]
                ip4 = gen_ip4(msg["uid"], self.peerlist, self.state["_ip4"])
                self.create_connection(msg["uid"], fpr, 1, CONFIG["sec"],cas, ip4)


def create_connection(self, uid, data, overlay_id, sec, cas, ip4):
    do_create_link(self.sock, uid, data, overlay_id, sec, cas)
    do_set_remote_ip(self.sock, uid, ip4, gen_ip6(uid))


def gen_ip4(uid, peers, ip4=None):
    if ip4 is None:
        ip4 = CONFIG["ip4"]
    return ip4[:-3] + str( 101 + len(peers))
```