EAI Endorsed Transactions
on Collaborative Computing

Research Article ICST.ORG

TinCan: User-Defined P2P Virtual Network Overlays for Ad-hoc Collaboration

Pierre St Juste*[1], Kyuho Jeong[1], Heungsik Eom[1], Corey Baker[2], Renato Figueiredo[1]

[1]Advanced Computing and Information Systems Lab, [2]Wireless and Mobile Systems Lab
Electrical and Computer Engineering, University of Florida, Gainesville, FL, 32611, USA

Abstract

Virtual private networking (VPN) has become an increasingly important component of a collaboration environment because it ensures private, authenticated communication among participants, using existing collaboration tools, where users are distributed across multiple institutions and can be mobile. The majority of current VPN solutions are based on a centralized VPN model, where all IP traffic is tunneled through a VPN gateway. Nonetheless, there are several use case scenarios that require a model where end-to-end VPN links are tunneled upon existing Internet infrastructure in a peer-to-peer (P2P) fashion, removing the bottleneck of a centralized VPN gateway. We propose a novel virtual network — TinCan — based on peer-to-peer private network tunnels. It reuses existing standards and implementations of services for discovery notification (XMPP), reflection (STUN) and relaying (TURN), facilitating configuration. In this approach, trust relationships maintained by centralized (or federated) services are automatically mapped to TinCan links. In one use scenario, TinCan allows unstructured P2P overlays connecting trusted end-user devices — while only requiring VPN software on user devices and leveraging online social network (OSN) infrastructure already widely deployed. This paper describes the architecture and design of TinCan and presents an experimental evaluation of a prototype supporting Windows, Linux, and Android mobile devices. Results quantify the overhead introduced by the network virtualization layer, and the resource requirements imposed on services needed to bootstrap TinCan links.

Received on 11 July 2014, accepted on 23 July 2014, published on 20 October 2014

Keywords: vpn, peer-to-peer, networking, privacy, virtual organization

doi: 10.4108/ cc.1.2.e4

1. Introduction

Virtual private networking (VPN) has become an increasingly important component of a collaboration environment because it ensures private, authenticated communication among participants, using existing collaboration tools, where users are distributed across multiple institutions and can be mobile. VPNs also allow groups from different organizations to create transient virtual networks thereby facilitating trusted resource sharing across the public Internet. The majority of VPN solutions are based on a centralized VPN model, where all IP traffic is tunneled through a VPN gateway. This model focuses primarily on one of these three goals: 1) secure network access to a private corporate network, 2) circumvention of a firewall restricting complete access to the global Internet, or 3) one-hop anonymity on the Internet. Nevertheless, there are several use case scenarios that require a model where end-to-end VPN links are tunneled upon existing Internet infrastructure in a peer-to-peer (P2P) fashion removing the bottleneck of a centralized VPN gateway, so called peer-to-peer virtual private networks (P2PVPNs).

The availability of P2PVPNs also help address the growing privacy concerns caused by recent NSA revelations through programs such as PRISM [1] because P2PVPNs create an environment where computing devices have end-to-end encrypted P2P tunnels which are used to route IP packets without the involvement of a middleman. This end-to-end encryption makes digital monitoring more challenging because there is no overseer that has direct access to all IP traffic flowing through the VPN, as is the case in most centralized VPN implementations. The main challenge to address is architecting an open VPN technology that is efficient, robust, easy to deploy and manage in a P2P fashion without compromising trust and while making it practical for common use in today's Internet.

This paper presents TinCan, a P2PVPN that allows flexible V PN o verlays o f d ifferent t opologies t hat can be instantiated atop Internet infrastructure with low

*Corresponding author. Email: pstjuste@acis.ufl.edu

configuration and management overhead [1]. TinCan integrates with existing online social networking (OSN) services for peer discovery and notification to allow deployments that bootstrap private peer-to-peer tunnels using relationships established through intuitive OSN interfaces. The overlay implied by private end-to-end TinCan links exposes IP endpoints, allowing existing applications to work unmodified, and providing a basis for overlay peer-to-peer routing in the virtual network. The TinCan design also supports tunneling of both IPv4 and IPv6 packets within the VPN, which is implemented as IPv6 packets encapsulated within UDP packets and sent over IPv4 P2P tunnels, as well as IPv4 packets within IPv6 UDP packets.

A key goal in the design is to minimize the amount of configuration and infrastructure necessary to sustain these virtual private networks. Because TinCan runs on endpoints (e.g. VMs or personal devices), it requires little additional infrastructure for maintaining the network. TinCan links make it possible for VMs and mobile devices to tunnel IP traffic directly to each other — even when constrained by NATs — while simultaneously giving end users the flexibility to define the IP address ranges, subnets, and access control policies for their private network. TinCan integrates with ubiquitous messaging overlays that use the XMPP protocol for signaling, along with well-adopted technologies for NAT traversal (STUN, TURN, and ICE [2–4]) to bootstrap encrypted TinCan links. In one use case, social peers can run TinCan to deploy VPNs comprised of their personal devices (including mobile) and their social peers' devices by leveraging existing Internet services for discovery and reflection (e.g. Google Hangouts, Jabber.org XMPP and STUN servers). The only requirement for deploying a TinCan VPN is an XMPP server; therefore, end users can use any freely available XMPP service on the Internet, or deploy their own private XMPP server such as *ejabberd* [5].

The novel design of TinCan is logically divided in two key layers — reminiscent of the OpenFlow model, but applied to tunnels over UDP/TCP links: 1) a datapath packet capture/forwarding layer, responsible for capturing/injecting packets from a virtual NIC, and maintaining TinCan links (over UDP or TCP) to neighboring peers, and 2) a control layer, responsible for implementing policies for the creation and tear-down of TinCan links. Each TinCan peer runs the two layers; communication across modules that implement each layer within a node is achieved through a JSON-UDP RPC interface. The available API allows for the control of TinCan link creation and deletion, mapping IP addresses to identities and TinCan links, and configuring virtual networking interface. Coordination among endpoints and overlay routing is possible through message forwarding along TinCan virtual IPv6 links, supporting user-defined overlay topologies and routing policies implemented as a separate module from the core datapath.

To demonstrate its applicability in different use cases, TinCan implements a common datapath based on Google's libjingle P2P library [6], and two different TinCan controllers: a "group" controller (which provides a private subnet with a flat address space for group collaboration), and a "social" controller (which automatically creates VPN links from social networking relationships established through an external OSN provider, for user-to-user collaboration). With the GroupVPN controller, nodes bind to the same subnet in the virtual network and can address each other using unique private IP addresses within the scope of the VPN. In SocialVPN mode, each user is able to define their own private IP range/subnet and locally map social peers to IP addresses within that subnet thus forming an unstructured social network graph overlay topology.

The analysis shows that the TinCan design is practical and scalable. In the experiments, a network of 300 nodes consumes 29 KB/s of bandwidth on the XMPP server. The management of these TinCan links uses about 1 KB/s of bandwidth per connection. The design incurs a 14% network per-packet encapsulation overhead. This overhead is due to our use of an MTU of 1280 bytes — selected to minimize packet fragmentation — rather than the traditional 1500 byte MTU along with the cost of an additional 40-byte header necessary to encapsulate the virtual IP packets. To measure system throughput, we conducted an experiment between two nodes in a 1 Gbps LAN and ran the iperf networking benchmark to obtain the bandwidth measurements. The results show a latency of less than 1 ms and a TCP bandwidth of 64 Mbps; since our target is to create virtual networks across the Internet, for most applications, the bottleneck will be the bandwidth limit imposed by their local ISPs.

The main contribution of this paper is a novel VPN design that leverages XMPP servers to bootstrap end-to-end VPN tunnels, supports decoupled controller/datapath model and P2P communication among controllers to implement different VPN membership, address mapping and overlay topology/routing policies, and leverages existing P2P technologies (STUN, TURN, and ICE) for establishing direct and secure P2P tunnels for IP connectivity. To the best of our knowledge, this is also the first P2PVPN design that allows computing devices to maintain their virtual IP address

[1] The name is inspired by tin can phones that provide private, ad-hoc communication links between friends

as they migrate across different networks while automatically re-establishing P2P connections with other nodes in the virtual network without the use of a relay.

The rest of the paper is organized as follows. A few motivating use cases are described in section 2. We summarize related works in section 3. We follow with an abstract view of our design choices in section 4 along with our policies in section 4.3. In section 5, we elaborate on the implementation details. The analysis is in section 6 and we conclude in section 7.

2. Usage Scenarios

VPNs have a history as a collaboration-enhancing tool. For instance, the Grid Applicance project [7] relies on a P2PVPN that allows researchers from different organizations to pool their virtual machines (VM) together in a virtual computing cluster. By using the HTCondor batch scheduling system [8], users are then able to securely submit computing workloads which are then dispatched to the geographically dispersed VMs that form this "virtual organization". For a more concrete example, the Archer collaborative environment [9] has demonstrated a system that employed a group-oriented VPN to connect cluster resources from various US universities, as well as student/researcher laptops and desktops. Collaborators may also decide to elastically augment their cluster by adding cloud compute nodes (e.g. Amazon EC2 VMs) to their computing cluster to enhance their capabilities. Our P2PVPN greatly facilitates this deployment because each VM instantiated in the cloud will seamlessly join the virtual cluster because they will have secure IP connectivity to each other in a scalable P2P fashion. The strength in our approach lies in the fact that these resources can be bridged seamlessly, without the intervention of university network administrators, because our P2P NAT traversal techniques allow nodes to connect from behind university firewalls. Moreover, since these virtual networking connections are peer-to-peer, the P2PVPN can be deployed with minimal infrastructure by using a public XMPP service such as Google Hangouts to bootstrap the VPN connections. Users can also run their own private XMPP service in the cloud in order to have complete control over their deployments. Through the XMPP service, user authentication and access control is administered.

The underlying advantage of a VPN is that it allows computing devices secure access to each other over the public Internet. A P2PVPN enhances this model because it ensures that only endpoints are able to encrypt/decrypt the IP traffic removing the reliance on a VPN gateway to handle that task. This can be a powerful tool in collaborative environments where a group of individuals need to quickly set up virtual organizations to achieve a common goal. Our P2PVPN approach makes it trivial for a group of collaborators to create a virtual network consisting only of trusted group members. Through this private network, collaborators are then free to share resources knowing that the user authentication and access control is already handled at the networking layer. Our tool therefore facilitates the creation of these virtual communities through private network access. The social aspects of our design also makes our approach novel because it uses well-known paradigms of social interactions to establish trust in the P2PVPN.

3. Related Works

Cloud Provider Virtual Networking. Over the past few years, major IaaS 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 a 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 or interoperability, thus placing the entire burden on users desiring cross-cloud deployments.

Third-Party Commercial Virtual Networking. To address challenges in network virtualization across different clouds, various third-party commercial solutions have emerged. VMware NSX [10] 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 [11] and RightScale's Cloud Management [12] products let 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 reimplements 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. TinCan targets the needs of users who are not willing to afford the configuration and management of additional virtual network infrastructure.

Overlay Virtual Networking Research. Academic and industry research have explored applicable solutions for virtual networking that allow geographically-dispersed nodes to create virtual private networks across the Internet. IBM researchers have developed VirtualWire [13] which implements a layer 2 virtual network tailored to the deployment of legacy applications and VM migration across different physical networks. Virtualwire is a hypervisor-level virtual network integrated with the Xen-Blanket [14] nested virtualization technology, enabling VM migration across public clouds. VIOLIN [15] uses a very similar approach to Virtualwire providing layer 2 networking with 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. The TinCan approach does not necessitate setting up additional DHCP and DNS servers.

VNET [16] provides layer 2 connectivity across different physical networks and it is also implemented at the hypervisor level. This is accomplished through a layer 2 proxy that bridges two different networks across the Internet. 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 connectivity. As the pool of IPv4 addresses becomes 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 [17] is a layer 3 virtual networking alternative which supports NAT/firewall traversal through relaying. However, it requires users to configure the virtual routers and does not provide end-to-end tunnels that bypass a relay/router node. Our work enables direct end-to-end IP tunneling without the need for a router middleman because each node runs an IP router locally.

Host-based and Mobile Virtual Networking. OpenVPN [18] is a solution that is applicable in 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 [19] 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 approaches such as Tinc [20], Vtun [21], and N2N [22] 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 wide-area 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. UIA [23] is a closely-related design aimed at providing ad-hoc virtual networking for mobile devices. One key difference in TinCan is the use of existing infrastructure, including OSN providers, to mediate peer discovery and bootstrapping. Previous work, IPOP [24], is a peer-to-peer VPN based on a structured P2P overlay for bootstrapping direct connection between nodes. While sharing similar goals, TinCan addresses several limitations of the IPOP design: in IPOP, peer discovery, bootstrapping, reflection, and relaying are provided by an overlay where peer-to-peer communication is layered atop a common structured P2P library (Brunet). TinCan decouples discovery, reflection, relaying and bootstrapping, decouples datapath from control modules, and exposes P2P communication through virtual IP links, allowing multiple overlay topologies. Our TinCan design does not depend on a structured P2P overlay, it uses publicly available STUN servers and XMPP servers to bootstrap P2P connections.

4. Design

This section describes the core components of the TinCan design which include a packet capture/forwarding datapath module, a network controller module, a discovery/notification overlay, reflection and relay servers. TinCan primarily enables an extensible framework for building P2PVPNs for various types of deployments. While the TinCan design supports other implementations, currently TinCan uses XMPP for discovery/notification, STUN for reflection, and TURN for relaying, and leverages the libjingle P2P library (developed by Google) to establish and maintain P2P TinCan links using the aforementioned services. The Session Traversal Utilities for NAT (STUN) protocol specifies how nodes behind network address translators (NAT) can discover their public IP address and port. Such

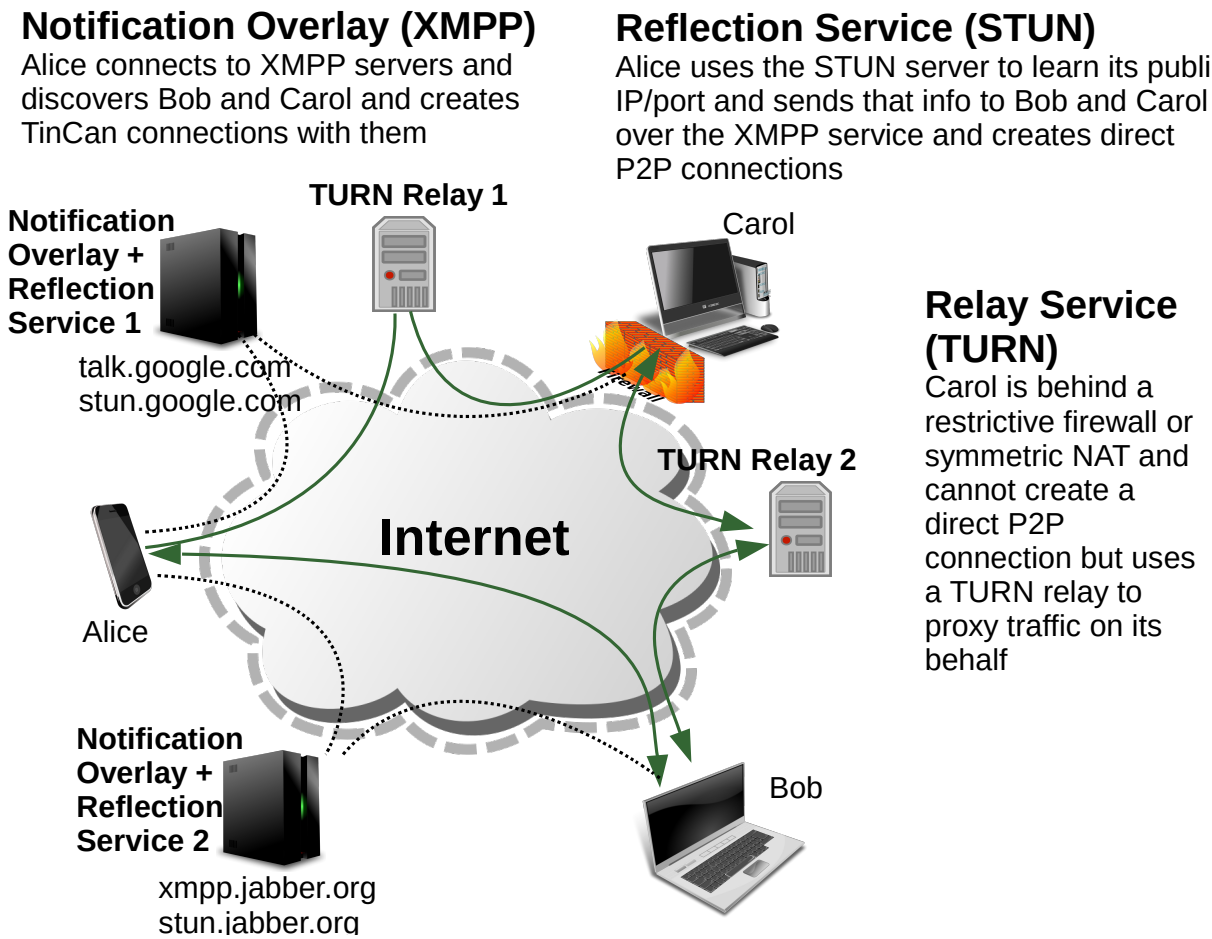

Figure 1. TinCan Components and Overview

information is crucial for establishing P2P connections with remote social peers over the Internet. The Traversal Using Relays around NAT (TURN) protocol describes how nodes behind restrictive (symmetric) NATs and firewalls can connect to each other through an intermediary relay node. Both of these protocols are widely used by SIP and WebRTC technologies (e.g. Google Hangouts). Figure 1 gives a general overview of the services involved in deploying the system. In figure 2, we demonstrate how unmodified applications, such as SSH, communicate through TinCan routers installed on each device. Through the TinCan framework, collaborators can reuse existing tools via their trusted P2PVPNs.

4.1. Endpoint-Hosted Components

Datapath packet capture/forwarding module. This component is a user-level module that runs on the end user device. It creates a virtual networking interface (vNIC) on the local operating system to capture and inject IP packets to/from local applications. It also possesses the mechanics of creating, maintaining, and tearing down encrypted TinCan links to peers, and manages a local routing table that maps a virtual IP address of an appropriate TinCan P2P link. In a typical packet flow scenario, this module reads an IP packet from the vNIC on the local OS, uses the destination IP address to lookup whether a mapping to a TinCan P2P link exists. If a TinCan link exists, the IP packet is encapsulated and sent directly over this link to the receiving data path module at the other endpoint node. Upon receiving the IP packet, the receiver decapsulates and injects it in the local vNIC (see figure 2).

The datapath module tracks the state of the local P2P links, and maintains a connection to one or more notification overlays (e.g. XMPP servers). TinCan links are typically tunneled over UDP — as it is most amenable to NAT traversal — and use DTLS for privacy, authentication, and integrity. DTLS stands for Datagram Transport Layer Security and it is the UDP version of the TLS protocol. The design also uses keep-alive messages to determine the state of P2P links, and uses the notification overlay to verify that online peers that are available to accept TinCan connections requests. This module is responsible for implementing

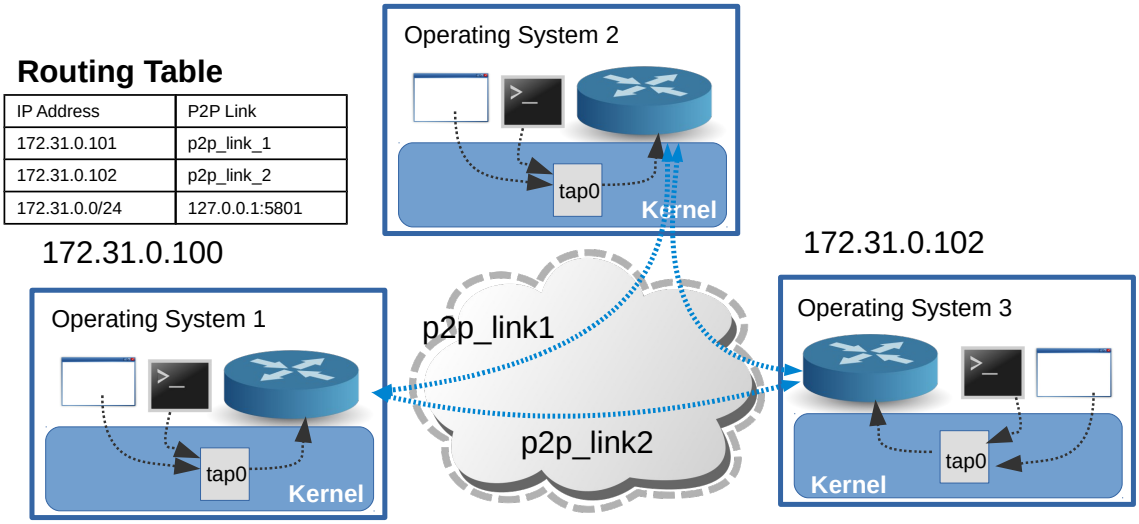

Figure 2. Applications Communication through TinCan Routers

the mechanisms to maintain TinCan links; however, it does not prescribe the policies associated with link creation and tear-down. To this end, it exposes an RPC interface to the controller module, decoupling mechanism from policy. The RPC API exposes the following functionality: 1) configuration of the virtual network interface, 2) creation and deletion of TinCan links, 3) registration into the notification overlay, and 4) adding a mapping for a destination virtual IP address (see figure 3).

Network Controller. The controller module implements different policies for managing TinCan links and the overlay topology. Through the API exposed by the datapath module, the controller determines the criteria for TinCan link creation, deletion, and the mapping of IP addresses. For example, a controller may implement a policy to create P2P connections when a node joins the network for a small-scale VPN with a proactive link creation policy, or only create connections on demand when virtual IP traffic is detected between endpoints. The controller also manages the configuration of the vNIC, including the IP address and network mask. Moreover, it maintains the necessary credentials to connect to the discovery overlay for certificate exchanges with peers in setting up private TinCan links. Figure 5 shows an example configuration for a controller.

In addition to programming local forwarding tables, the controller is also responsible for routing virtual IP packets not mapped to local TinCan links through one or more hops. This mechanism is used to route packets when a direct TinCan link is not available, for instance while a link is being initialized. Controllers bind to an IPv6 vNIC that allows it to communicate to neighboring controllers over TinCan links; this private IPv6 address is configured with a unique node ID which can be used for identifier-based routing. In doing so, the controllers can use this mechanism to implement different overlay topologies and routing algorithms without requiring changes to the core datapath (see figure 4). Finally, the controller also determines the policies for various network events such as node arrival and departures, TinCan connection requests, and link failures.

4.2. Internet-Hosted Components

Notification/Discovery Overlay. As stated above, the datapath module maintains a communication link with a notification overlay (e.g. XMPP server) that allows for the advertisement of network-wide events such as node arrivals and departures. The notification overlay plays the role of the trusted out-of-band channel for bootstrapping encrypted TinCan connections (see figure 5). When two nodes decide to create a TinCan connection, they exchange a list of candidate endpoints (i.e. public and private IP addresses and ports) and security credentials (i.e. X.509 certificate fingerprints). The notification overlay provides the following primitives: 1) multicast notification to peers

connected to the overlay (e.g. XMPP buddies), 2) unicast message delivery to a specific node, and 3) node authentication and message integrity guaranteeing trusted node identity and message delivery.

Reflection and Relay Servers. There are two services needed in the public network to enable the bootstrapping of TinCan links through NATs and firewalls. First, reflection servers are used to inform nodes of their public-facing IP addresses and ports. Nodes are then able to exchange their public IP information with other nodes through the notification overlay to bootstrap TinCan connections. While most NATs are amenable to UDP NAT traversal, around 8% of the time [6], nodes behind symmetric NATs or some restrictive firewalls cannot create direct TinCan connections (see figure 1). In those cases, they require the assistance of a relay server with a public IP address. A relay service serves as an indirect communication path when a direct TinCan link cannot be established.

4.3. Controller Policies

A key aspect of the TinCan design is extensibility, accomplished through decoupling of the controller and data path. This approach is inspired by OpenFlow [25], but applies at the IP layer over tunneled links, rather than at layer 2 flows over physical links. To illustrate the extensibility of the design, this section describes two different controller models: a "group" VPN for virtual private clusters, and a "social" VPN connecting personal (and mobile) devices of social peers (see figure 4). For the former use case, the controller creates a VPN where nodes join the same virtual subnet (e.g. 10.10.0.0/16) and IP addresses are assigned by the VPN network creator. Virtual IP addresses are bound to node identifiers within the scope of this VPN by configuring the node ID to be a cryptographic hash function of the virtual IP address. TinCan links are created on-demand in response to IP packets being captured by the datapath module; while links are setup, packets may be dropped by a controller, or routed through overlay hops. In the "social" VPN model, the controller creates VPNs where per-endpoint virtual network address spaces are created at each node, peers are mapped dynamically to IP addresses within this namespace, and address translation is handled transparently. For instance, Alice has friends Bob and Carol; her VPN binds virtual IP addresses of Bob and Carol to a local private subnet (e.g 172.31.x.y). Bob and Carol have their own mappings of friends to virtual IP addresses within the local IP address space (e.g. Bob uses 10.15.x.y, Carol uses 192.168.5.y). Alice may link to Bob and Carol, while Bob and Carol may not have a direct link to each other if they are not friends. So far, we have only implemented two different types of controllers but we envision other controllers with various IP allocation and management policies.

Network Admission. Nodes joining the network advertise themselves and exchange connection information for bootstrapping P2P links through a trusted notification overlay. Hence, admission to the network is controlled by establishing identities and membership (e.g. friend-to-friend, or groups) in the notification overlay. In the group VPN scenario, each node in the network is given the following network settings: a private IP address and netmask, the network ID, the address of the notification overlay service, and a username/password for accessing the group through the overlay (see figure 5). For example, suppose Trent is a trusted user responsible for creating a VPN. Trent creates a personal VPN and distributes credentials for authentication and network access for each endpoint to join the notification overlay. Alternatively, Trent may establish relationships (e.g. XMPP buddies) with other users who are authorized to join the VPN. Trent would then determine the IP address range and netmask for the network and distribute these settings to each VM that joins the virtual network. In the social VPN case, users have their own customized view of the network and thus define their own peer-to-peer trust relationships in the notification overlay, and select their own local private IP address and netmask. In social mode, users are not required to share XMPP credentials to other members of the VPN because TinCan leverages existing social relationships to determine admission into the P2PVPN; therefore, only a user's XMPP buddies will be part of a user's social VPN.

Proactive Link establishment. If the controller implements a "proactive" link policy, it triggers a connection request as soon as a node joins the notification overlay and proactively creates TinCan P2P links to peers even if no IP packets are flowing. This policy has the benefit of reducing latency for packets, but comes at the cost of increased resource utilization (ports and bandwidth). The proactive link policy implied by this approach may be applicable for small overlays [26], but is not scalable (see figure 4).

On-Demand Links. An alternative controller policy is "on-demand connections" where TinCan links are formed when the controller receives a packet with a destination IP address that is not currently mapped to a TinCan link. This event triggers a connection request through the notification overlay, which results in a new TinCan link being mapped to the destination IP address. Such a policy causes a delay when connecting to new IP addresses. The controller with this policy also limits the number of connections, and can expire links with inactive IP flows.

Social Profile Links. Another connection policy deals is one where nodes are only interested in connecting with social peers rather than every node in a

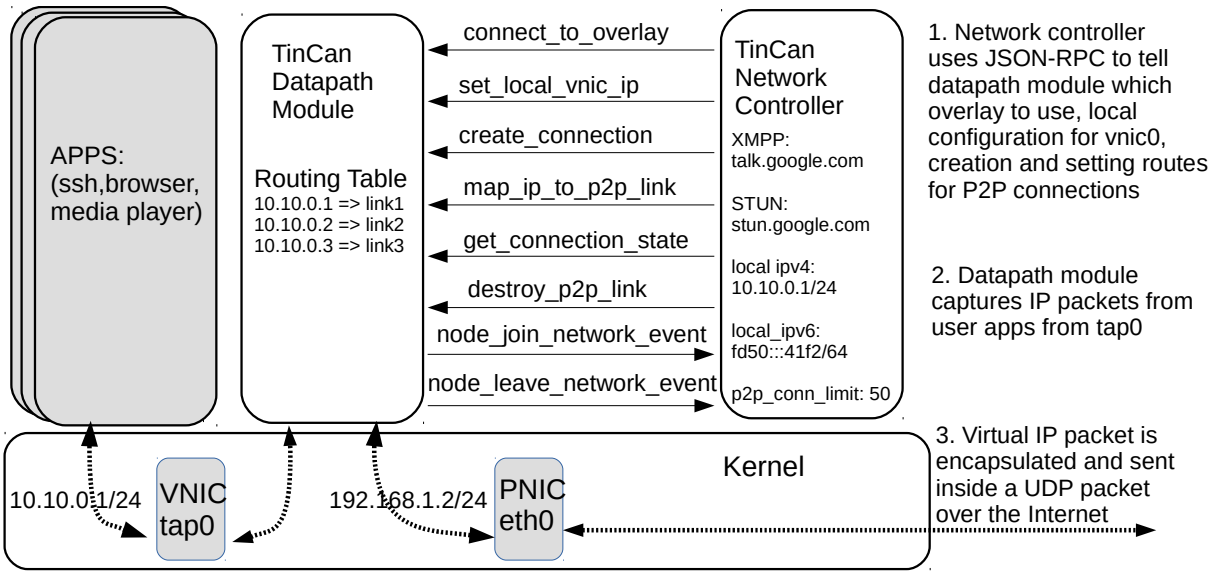

Figure 3. Interaction between TinCan Modules

particular group. In this model, peers create proactive links with friends that they have frequently communicated with in past sessions, and on-demand or multi-hop routing through common friends for nodes for which communication is infrequent. Other policies may include a combination of on-demand and proactive connections, and create overlay topologies that attempt to match communication patterns expected (or observed) by applications (see figure 4).

IP Addressing and Translation The controller has the flexibility to assign an IP address to the device and map friends to IP addresses in a subnet range that does not conflict with the local network. Each controller is able to select its own subnet range without coordinating with a centralized entity or other controllers. Therefore, in "social vpn" mode, each controller can select a different subnet for their network; meaning that IP addresses are only valid locally. This is of crucial importance for IPv4 addresses where the virtual address space is limited and can lead to IP conflicts. Since each user defines their own network, they can freely select IP addresses without fear of network subnet collisions. The datapath module performs IP packet translation on incoming packets which ensures no IP conflict following the approach described in previous work [27]. For example, Alice maps her mobile phone to 172.31.0.1 and maps Bob to 172.31.0.2. On his mobile device, Bob's controller maps his mobile device to 192.168.0.1 and maps Alice to 192.168.0.2. Hence, Alice is able to reach Bob's mobile phone using the 172.31.0.2 IP address and the IP translation performed by the datapath module on Bob's phone would make it seem as if the request came from 192.168.0.2. The proposed design also readily creates a private IPv6 address space and pseudo-randomly assigns IPv6 addresses to nodes in the network. Since the IPv6 address space is so vast, no IP translation is necessary due to much lower probabilities of collisions in the virtual IP space.

5. Implementation

The current TinCan implementation reuses existing technologies and infrastructures that enable P2P connections for both SIP and WebRTC standards by leveraging Google's libjingle [6] P2P library to create private TinCan links. XMPP servers (possibly federated) serve as the discovery/notification overlay. By using STUN and TURN servers for reflection and relaying, which are Internet services already freely accessible, users can deploy their own VPNs without any additional infrastructure.

5.1. Endpoint-Hosted Components

Packet capture/forwarding. The datapath packet capture/forwarding module is written in C/C++ and currently runs on Linux, Android, Windows and OpenWRT. Through the TUN/TAP kernel driver, TinCan is able to receive and send Ethernet frames to the vNIC. TinCan uses libjingle [6], leveraging its adoption in existing software (e.g. the Chrome browser). While the typical use of libjingle is for audio/video streaming in WebRTC, TinCan uses it to tunnel virtual IP packets.

Controllers. The controllers are written in Python and run as a separate process on the local machine. Controllers access the datapath module's API through a JSON-RPC interface over a UDP socket. The controller uses the API exposed by the datapath module to:

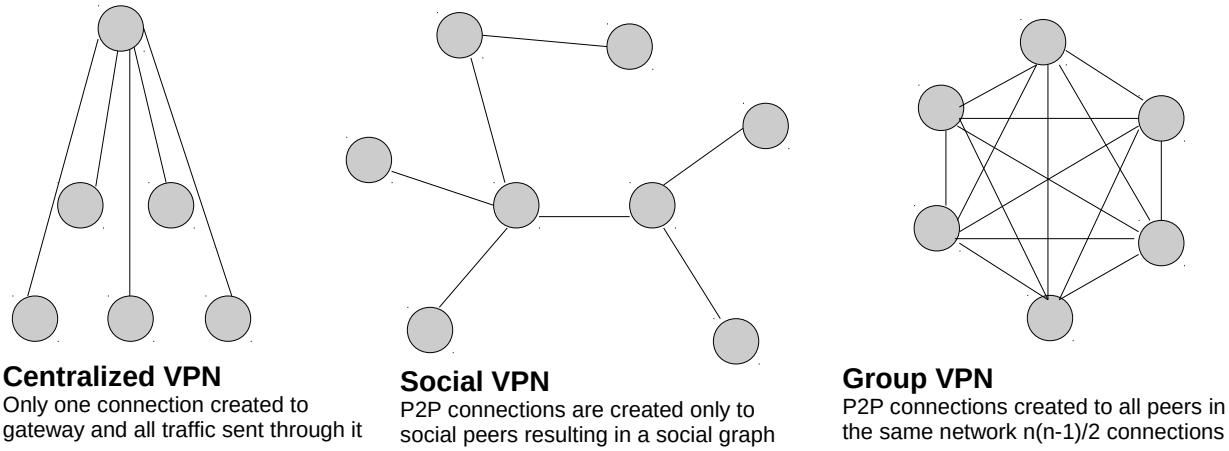

Figure 4. VPN Topologies

- Register with credentials to the XMPP overlay
- Setup the local vNIC with IPv4/IPv6 addresses, and netmask
- Create/Delete a TinCan link over jingle
- Map an IP address to a TinCan link
- Query the state of a TinCan link
- Handle notifications received through the XMPP overlay (e.g. new node presence, request to connect)
- Handle notifications received from the data path module (e.g. to forward virtual IP packets to other controllers when the destination is not mapped to a local TinCan link)

Through the API, one can extend TinCan to support various combinations of policies and deployments based on anticipated use cases.

5.2. Internet-Hosted Components

XMPP Notification Overlays. The messaging overlays play a crucial role in providing access to the network and as well as serving as a trust anchor for signaling and bootstrapping private TinCan links. The XMPP protocol accomplishes this role by securely routing XML messages through user authenticated TLS connections (see figure 5). Hence, TinCan-based VPNs are able to utilize public XMPP providers (such as Google Hangouts or Jabber.org), as well as use their own XMPP service (e.g. an ejabberd server) if they desire that level of control.

STUN and TURN Servers. For the reflection and relay servers, the STUN and TURN protocols are used, respectively. These technologies are used in the SIP/WebRTC communities to enable P2P connections for audio and video conferencing; as a result, there are many publicly available STUN servers that TinCan can utilize when creating P2P connections. For some nodes behind symmetric NATs or restrictive firewalls, an XMPP server and STUN server may not be enough to bootstrap a TinCan link; therefore, less than 10% of the time [28], these nodes require the assistance of a relay server to help proxy their TinCan connections. The TURN standards [3] provide such a relaying capability. Google Hangouts is an example of an existing service that already provides such a capability for its users; therefore TinCan links can leverage that for connection relaying through libjingle. There are also many open-source implementations of TURN relays. This work uses one of those implementations [29] for experimentation.

5.3. Bootstrapping Private TinCan Links

TinCan assumes that the XMPP server is a third party trusted for peer discover, notification, and exchange of X509 certificate fingerprints. Users can connect to servers they trust, or deploy their own private XMPP server. All communication from TinCan modules to the XMPP server is encrypted at the socket layer using transport layer security (TLS). A user authenticates herself with the XMPP server and broadcasts a presence probe to all peers (or buddies in XMPP terminology) that are part of their group. Therefore, all of the nodes within the group that are connected to the XMPP server receive the presence probe. Each node in the network periodically broadcasts a ping message to all other nodes in the network every two minutes. The datapath module maintains a list of online peers along with the timestamp of their last XMPP broadcast message. Once peers are able to discover and notify each other through the XMPP server, they can proceed to create trusted TinCan links.

To this end, a connection request is created containing the requester's X.509 fingerprint, a list of

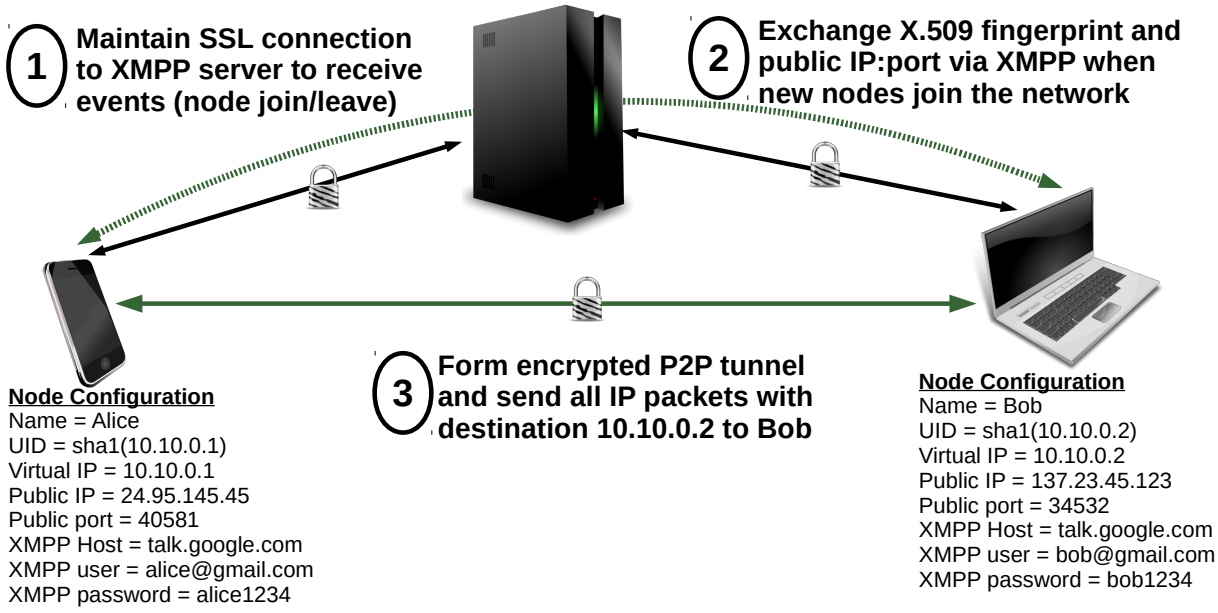

Figure 5. Bootstrapping TinCan Connections

endpoints containing private/public IP addresses with port numbers, and security credentials to ensure access control for the connection. The request is then sent to the peer over the XMPP overlay. The recipient replies to the request with a query response mirroring the contents of the request: X.509 fingerprint, list of endpoints, and security credentials. Once both sides have the necessary information, they initiate a TinCan link with each other by sending packets directly to these public IP addresses until a response is received (see figure 5). This process follows the Interactive Connectivity Establishment (ICE) RFC [4].

As mentioned earlier, nodes exchange their X.509 certificate fingerprint as part of the connection request/reply messages. To encrypt the link, the libjingle library uses the OpenSSL Datagram TLS (DTLS) protocol with peer certificate verification. Once the certificates have been successfully verified and a symmetric key is derived from the Diffie-Hellman exchange, the DTLS protocol can proceed to encrypt data flowing through the P2P channel between the peers. IP packets picked by the vNIC interface are encapsulated into data packets sent over the link, and thus protected by DTLS. It is possible to apply IPsec-layer end-to-end security atop of the virtual network overlay as well.

6. Analysis

Various experiments were conducted to understand the resource requirements of the TinCan design. This analysis also focuses on measuring the overhead of maintaining the VPN, packet processing, and the power consumption on mobile devices. In order to make these experiments reproducible, all of the source code is open on Github at http://github.com/ipop-project. To test scalability, we setup a 300-node deployment on FutureGrid [30] using a mix of virtual machines and Linux containers (LXC). FutureGrid is an experimental Infrastructure-as-a-Service (IaaS) cloud environment that is available for academic research.

By running a 300-node experiment, we are able to analyze the bandwidth usage on the XMPP server, as well as the maintenance cost of managing TinCan P2P links. Rather than reuse existing infrastructure such as Google XMPP and STUN servers, for these experiments, independent *ejabberd* XMPP and STUN servers were deployed in order to have greater control over the testing environment. This experiment consisted of 8 virtual machines (VMs) running Ubuntu 13.10. One VM ran the notification overlay service, we used the *ejabberd* [5] open-source XMPP server implementation. Another VM hosted the reflection and relay servers, we also used another open source implementation of the TURN protocol to enable these services [29]. For these two deployments, the bandwidth load on the XMPP, STUN and TURN servers is summarized in Table 1.

Each of the remaining 6 VMs ran 50 instances of our TinCan implementation through the use of Linux containers (LXC [31]) which is a lightweight virtualization technology. Using LXC allows for more efficient utilization of resources because it makes it possible to simulate a 300-node network without needing to use 300 VMs or personal devices. The LXC environment is configured to create an isolated virtual network for the containers residing in the

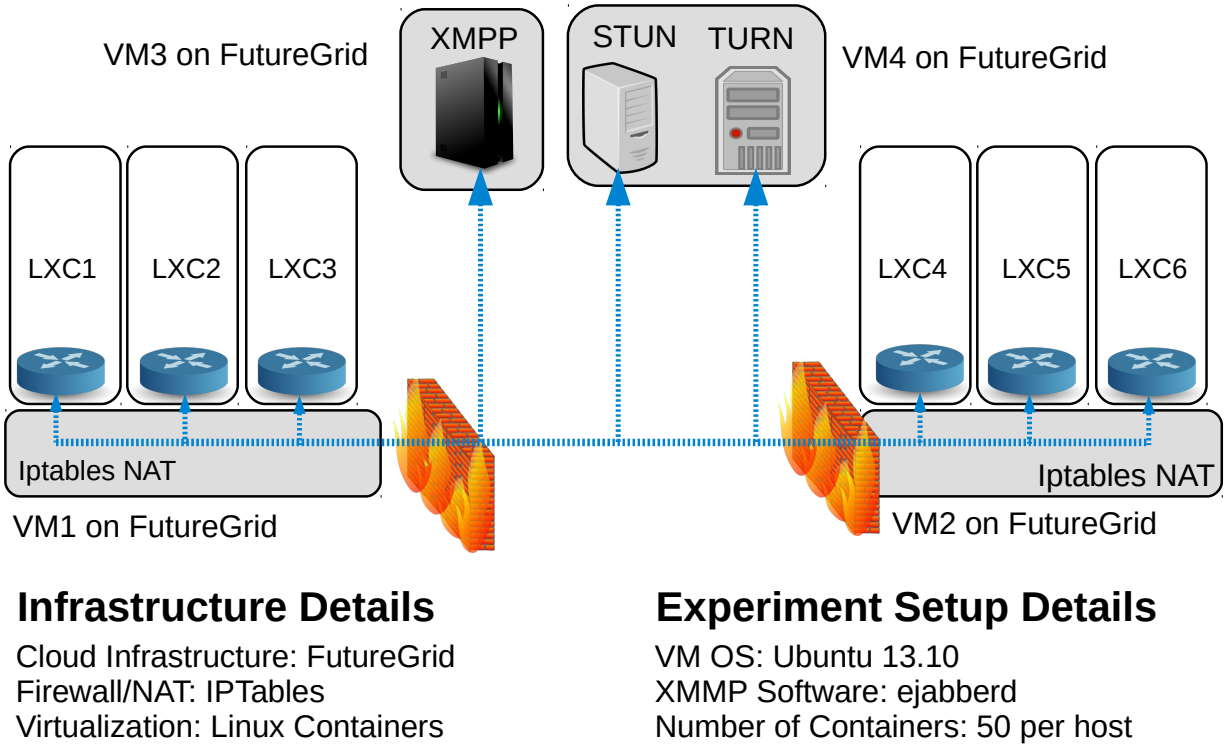

Figure 6. Experimental Setup Details

same VM; these containers are then able to connect to the outside world through the IPtables symmetric NAT (see figure 6). Therefore, the nodes running on different VMs have to rely on the relaying service (TURN) because the symmetric NATs do not allow for UDP hole-punching thus precluding direct TinCan P2P connections. In practice, typical usage scenarios are unlikely to be as constrained by symmetric NATs, nor is the use of a proactive all-to-all policy recommended for all but small-scale VPNs, since it does not scale well.

For this experiment, we utilized the "social vpn" controller to represent the use case where end users would like their personal devices (e.g. desktops, laptops, tablets, smartphones) along with their friends' devices to belong to the same SocialVPN and thereby having secure network access to each other. In this model, social relationships are mapped to TinCan VPN connections; for example, if Alice has a friend Bob, then if they run TinCan in SocialVPN mode on their devices, these devices will automatically join each other's social virtual private network. Hence, the TinCan P2P links in the SocialVPN mode will resemble the edges of a social graph because each VPN link represents a social link (see figure 4). To simulate this social graph environment, we used the Barabasi-Albert model from the NetworkX graph library [32] and generated 300-node graph with 1475 edges (or TinCan links).

Table 1. Experimental Setup Summary

Parameter	Value
Number of VMs	6
Number of Containers per node	50
Number of nodes	300
Number of connections	1475
Bandwidth Cost for TinCan connection	1 KB/s
Average Traffic at XMPP server	19 KB/s
Average Traffic at STUN server	27 KB/s
Average Traffic at TURN server	145 KB/s

6.1. Bandwidth Costs

During this deployment, the average bandwidth consumption at the XMPP server is about 19 KB/s; this shows that our protocol incurs very little traffic on the XMPP server. This traffic is primarily the periodic ping messages that each node in the network send to each other to indicate that they are still alive. In the case of the reflection (STUN) server, the TinCan implementation running on the end-nodes sends a 64-byte STUN *binding request* and receives a 72-byte STUN *binding response* every 15 seconds per connection. Therefore, the bandwidth cost on the STUN server for supporting our deployment of 1475 TinCan connections is about 27 KB/s (or 0.18 KB/s per

Table 2. VPN Network Performance

	Latency	TCP	UDP
LAN	0.5 ms	325 Mbps	320 Mbps
TinCan DTLS	1.07 ms	64 Mbps	47 Mbps
TinCan no DTLS	1.07 ms	84 Mbps	128 Mbps

connection). Also there are numerous freely accessible STUN servers on the web hosted by Google and others meaning that these resources can also be leveraged for the reflection service. Table 1 summarizes the bandwidth costs of the deployment.

In order to calculate the bandwidth cost on the TURN relay server, it is important to understand the maintenance cost of each TinCan connection. Libjingle sends a STUN *user request* every 500 ms and expects a *Success Response*; the average size of these packets is 130 bytes. These ping packets help libjingle keep track of the state of the TinCan link in terms of latency, jitter, and link failure. Therefore, each TinCan connection consumes about 1040 bytes per second as connection maintenance overhead. When a TURN server is used for relaying, these ping messages are routed through the relay. According to Google research, about 10% of P2P connections require a TURN relay and therefore, supporting a 300-node network with 1450 edges would necessitate a relay service for about 145 connections costing about 145 KB/s for connection maintenance. It is important to note that the TURN service would also have to relay IP traffic between the nodes that it supports and therefore deploying a TURN service requires thoughtful planning and proper access control. TURN implementations provide user authentication making it possible to identify different connections and apply bandwidth limitations per user. For instance, it is possible to configure a TURN server to only allow a maximum of 50 KB/s throughput per connection and limit the number of connections. However, since the relay service is required in the face of symmetric NATs (i.e. less 10% of the time) it is possible to support up to 1000-node network on a single TURN server. There are also commercial offerings such as turnservers.com that provide this relay service for a fee if users do not want to deploy their own TURN service.

6.2. Network Performance

One of the drawbacks of the TinCan design is that, instead of dedicated virtual switches and routers, each node runs their own virtual router that tunnels IP packets to the appropriate TinCan links. Therefore, every IP packet has to be encrypted, decrypted, and translated. This user-level packet processing can greatly constrain network performance. In this experiment, the iperf network benchmarking suite measured the maximum bandwidth achievable by TinCan between two nodes in the same gigabit LAN. As shown in table 2, TinCan achieves 64 Mbps for TCP and 47 Mbps for UDP with DTLS encryption, but without encryption the bandwidth increases to 84 Mbps for TCP and 128 Mbps for UDP. A possible optimization for LAN environments is to bypass the overlay and allow TinCan nodes in the same LAN to directly route packets to each other without encryption, as described in [33]. TinCan supports this router-mode of operation; in this mode, containers or VMs on the same host can all share a single instance of the TinCan router. Local nodes can therefore communicate directly with each other and only use the TinCan pathway to private connect with remote nodes outside of their LAN. It is also possible to run TinCan in OpenWRT-enabled routers, this approach would also make it possible for local nodes to communicate directly without the overhead of the local processing and they would also have connectivity to nodes in the P2PVPN since the OpenWRT router would now have a connection into the network.

Understanding the time it takes to create a TinCan connection is crucial in designing a controller when considering a proactive connection policy versus an on-demand connection policy. As shown in figure 7, the median connection setup time is about 6.3 seconds, with the 75% percentile at 7.8 seconds but in the worst case, it may take up to a few minutes to bootstrap a connection due to dropped connection requests. Therefore, it may not be ideal to use an on-demand connection policy if an application generates bursty traffic that is sensitive to high latency start-up times. For the proactive connection policy, this connection setup only occurs once when a node joins the network; afterwards, TCP/IP connections through this TinCan link will not be subject to this long setup time. For the on-demand policy, the controller has to determine when to create or trim TinCan connections. For instance, one option might be to trim a TinCan connection if a link has been idle for five minutes; in this case, nodes will have to re-experience the connection setup-time in order to re-establish a connection with a trimmed link.

6.3. Encapsulation Overhead

The proposed design incurs packet overhead due to the additional headers necessary for IP encapsulation. Another source of overhead is the selection of a relatively small MTU for the vNIC. Ethernet devices typically have an MTU of 1500 bytes, but using an MTU of 1280 bytes minimizes the probability of UDP packet fragmentation. Moreover, the TinCan implementation uses a 40-byte header for each packet consisting of a

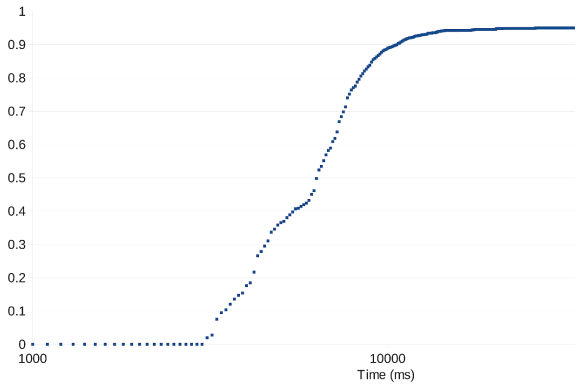

Figure 7. CDF of 1450 Connection Times. 75% of connections take less than 8 seconds.

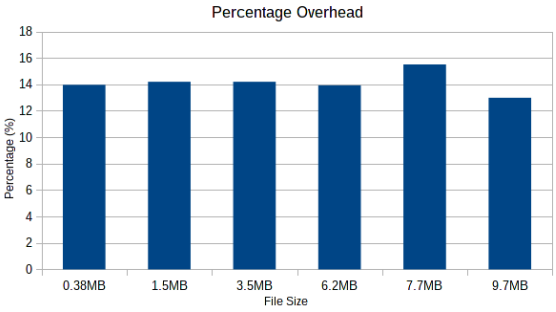

Figure 8. File Transfer Percentage Overhead. Due to MTU of 1280 and extra 40-byte header for IP encapsulation, there is a 14% overhead in the extra number of bytes sent over the network for the same file size when compared to WiFi.

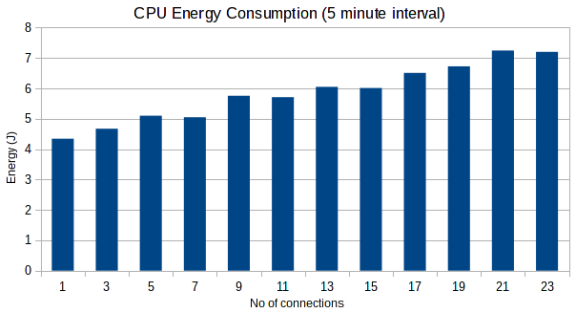

Figure 9. CPU Engergy Consumption on Mobile

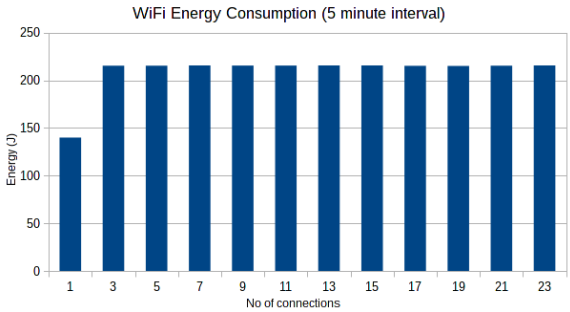

Figure 10. WiFi Energy Consumption on Mobile

20-byte source unique identifier (UID) and another 20-byte destination UID. The 160-bit UIDs creates an extra level of indirection which facilitates packet routing in the network. Consequently, a small MTU and the extra header has an adverse impact on network performance.

The following experiment quantifies the network overhead. For this setup, there is a TinCan network of just two nodes, a Samsung Galaxy Tab 10.1 and an Ubuntu 12.04 workstation. The tablet has a 1GHz dual-core Nvidia Tegra 2 processor and 1GB of RAM and the workstation is a 3.0GHz Intel Core 2 Duo with 8GB of RAM. By performing file transfers of different sizes as shown in figure 8 over both WiFi and TinCan, the results show an average network overhead of 14%. This overhead can be reduced by choosing a higher MTU closer to 1500 bytes and by using a smaller header size (e.g. 128-bit UIDs instead of 160-bit).

6.4. Mobile Power Consumption

Mobile computing support is an important aspect of the TinCan design; hence, an experiment on the Android tablet, with the number of connections scaled up from 1 to 23, provides insight in the power costs of TinCan P2P connections. PowerTutor [34], a software-based power measuring app available for Android, calculated both the WiFi and CPU energy consumption. In terms of CPU energy consumption, figure 9 shows a steady increase in energy cost which averages about 0.13 Joules (J) per 5-minute interval ranging from 4.3 J for one connection to 7.2 J for 23 connections. For comparison, the LinPack for Android benchmark on the same tablet consumes 64.7 J for the same time interval. The WiFi energy consumption in figure 10 shows a different pattern where there is a sharp energy increase from 1 to 3 connections followed by a steady state. As mentioned earlier, a TinCan connection generates about 8 network packets with sizes around 130 bytes per second consuming about 1 KB/s of bandwidth. The mobile WiFi card is able to handle the bandwidth requirements of one connection in low-power mode. However, once there is more than a single P2P connection, the WiFi enters high-power mode which increases the energy consumption by 1.5x from 144 J to 220 J (for a 5-minute period). Since the WiFi card remains in high-power starting with two connections, there is no significant change in energy consumption as the number of connections increases.

6.5. Zero Infrastructure Experiments

The key advantage of the TinCan design is the ability for a user to create their own virtual private network by simply running the software on their end devices or

cloud instances and configuring it to use existing XMPP services (e.g. Google.com or Jabber.org). To demonstrate this, a virtual network consisting of two Android devices was created: a Motorola Photon Q smartphone and a Samgsung Galaxy Tab 7. The smartphone was connected via the Sprint 4G network while the tablet connected via WiFi network. Using the Google XMPP servers, the two devices created a SocialVPN and therefore had private IP access to each other as if they are connected on the same LAN. Using the CSipSimple Android app, the two devices could perform SIP calls between each other. The call was performed by simply using the devices' virtual IP addresses as the SIP address (i.e. sip@172.31.0.101). Consequently, secure SIP calls were conducted over TinCan IP links through both the 4G ISP firewall and WiFi NAT without any registration or signaling through a SIP server. By leveraging Google's XMPP service along with the dozens of publicly available STUN servers on the web, users can easily get private IP connectivity to each other at no cost.

7. Conclusion and Future Work

Collaborative environments are evolving to include more mobile and cloud resources that are geographically dispersed and mobile. The increased use of cloud and mobile computing as ad-hoc collaborative tools have created a need for more user-defined overlay virtual networks which enable both node mobility and information security. The proposed TinCan design leverages existing overlay and P2P technologies such as XMPP, STUN, and TURN thereby creating a solution where users can define and deploy their virtual networks without needing additional infrastructure. Additionally, unlike other existing solutions, this approach does not require special access to the hypervisor, nor do users have to configure virtual switches and routers. To provide layer 3 connectivity, each node in the network runs their own virtual router which maps IP addresses to TinCan connections. Analysis of the TinCan design shows that a network of 300 nodes incur acceptable bandwidth loads on the XMPP, STUN, and TURN servers. The experiments also show that it takes less than 10 seconds to create 75% of TinCan P2P connections. The additional headers for IP encapsulation and smaller vNIC MTU cause a 14% network overhead. In terms of mobile power consumption, it seems ideal to only maintain one TinCan connection at a time to avoid running the WiFi card in high-power mode.

The evaluations consider the overheads associated with a single link, and for small-scale VPNs that could be deployed with a very simple topology and connection policy. These are feasible for small-scale VPNs, e.g. for small virtual clusters; for VPNs scaling to larger number of nodes (100s to 1000s), it is clearly a requirement to reduce the number of links in order to reduce traffic at the notification/discovery and reflection services. One approach that scales well and has been used in previous work is to use a structured P2P routing overlay with on-demand shortcut connections [24]. The choice of a scalable overlay approach can be encoded in the logic embedded in the controller. Future work will consider different topology options (e.g. different structured P2P approaches, as well as social and random graphs) and different policies for on-demand link establishment/tear-down. We will also explore bootstrapping TinCan connections through friends of friends in the mobile device deployments rather than the XMPP server to facilitate more ad-hoc private networking.

8. Acknowledgements

This material is based upon work supported in part by the National Science Foundation under Grants No. 1339737, 1234983, and 0910812. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Gellman, B. and Poitras, L. (2013), U.s., british intelligence mining data from nine u.s. internet companies in broad secret program, http://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program.html.

[2] Rfc 5389 - session traversal utilities for (nat) (stun), http://tools.ietf.org/html/rfc5389.

[3] Rfc 5766 - traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun), http://tools.ietf.org/html/rfc5766.

[4] Rfc 5245 - interactive connectivity establishment (ice): A methodology for network address translator (nat) traversal for offer/answer protocols, http://tools.ietf.org/html/rfc5245.

[5] ejabberd - the erlang jabber/xmpp dameon, http://www.ejabberd.im/.

[6] About libjingle - google talk for developers, https://developers.google.com/talk/libjingle/.

[7] Ganguly, A., Agrawal, A., Boykin, P. and Figueiredo, R. (2006) Wow: Self-organizing wide area overlay networks of virtual workstations. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on* : 30–42.

[8] Htcondor - high throughput computing, http://research.cs.wisc.edu/htcondor/.

[9] Figueiredo, R.J., Boykin, P.O., Fortes, J.A.B., Li, T., Peir, J., Wolinsky, D., John, L.K. *et al.* (2009) Archer: A community distributed computing infrastructure for computer architecture research and education. In

Collaborative Computing: Networking, Applications and Worksharing (Springer Berlin Heidelberg), **10**: 70–84.

[10] Vmware nsx - network virtualization, http://www.vmware.com/products/nsx/.

[11] Vns3 overlay sdn product - cohesiveft, http://www.cohesiveft.com/products/vns3/.

[12] Blanquer, J. (2013), How rightscale supports virtual networking across clouds, http://www.rightscale.com/blog/rightscale-news/how-rightscale-supports-virtual-networking-across-clouds.

[13] Jamjoom, H. (2013) Virtualwire: system support for live migrating virtual networks across clouds. In *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, VTDC '13 (New York, NY, USA: ACM): 21–22. doi:10.1145/2465829.2465838.

[14] Williams, D., Jamjoom, H. and Weatherspoon, H. (2012) The xen-blanket: virtualize once, run everywhere. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12 (New York, NY, USA: ACM): 113–126. doi:10.1145/2168836.2168849.

[15] Jiang, X. and Xu, D. (2004) Violin: virtual internetworking on overlay infrastructure. In *Proceedings of the Second international conference on Parallel and Distributed Processing and Applications*, ISPA'04 (Berlin, Heidelberg: Springer-Verlag): 937–946. doi:10.1007/978-3-540-30566-8_107.

[16] Sundararaj, A.I. and Dinda, P.A. (2004) Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04 (Berkeley, CA, USA: USENIX Association): 14–14.

[17] Tsugawa, M. and Fortes, J.A.B. (2006) A virtual network (vine) architecture for grid computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06 (Washington, DC, USA: IEEE Computer Society): 148–148.

[18] Openvpn - open source vpn, http://openvpn.net/.

[19] Hamachi - instant, zero configuration vpn, http://secure.logmein.com/products/hamachi/vpn.asp.

[20] tinc wiki, http://www.tinc-vpn.org/.

[21] Vtun - virtual tunnels over tcp/ip networks, http://vtun.sourceforge.net/.

[22] n2n, http://www.ntop.org/products/n2n/.

[23] Ford, B., Strauss, J., Lesniewski-Laas, C., Rhea, S., Kaashoek, F. and Morris, R. (2006) Persistent personal names for globally connected mobile devices. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06 (Berkeley, CA, USA: USENIX Association): 233–248.

[24] Ganguly, A., Agrawal, A., Boykin, P.O. and Figueiredo, R. (2006) Ip over p2p: enabling self-configuring virtual ip networks for grid computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06 (Washington, DC, USA: IEEE Computer Society): 49–49.

[25] Open networking foundation, https://www.opennetworking.org/.

[26] Andersen, D., Balakrishnan, H., Kaashoek, F. and Morris, R. (2001) Resilient overlay networks. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01 (New York, NY, USA: ACM): 131–145. doi:10.1145/502034.502048.

[27] Juste, P.S., Wolinsky, D., Oscar Boykin, P., Covington, M.J. and Figueiredo, R.J. (2010) Socialvpn: Enabling wide-area collaboration with integrated social and overlay networks. *Comput. Netw.* **54**(12): 1926–1938. doi:10.1016/j.comnet.2009.11.019.

[28] Important concepts - google talk for developers, https://developers.google.com/talk/libjingle/important_concepts

[29] Turnserver - open-source turn server implementation, http://turnserver.sourceforge.net/.

[30] Fox, G., von Laszewski, G., Diaz, J., Keahey, K., Fortes, J., Figueiredo, R., Smallen, S. *et al.* (2013) *FutureGrid - a reconfigurable testbed for Cloud, HPC, and Grid Computing*, CRC Computational Science (Chapman & Hall).

[31] Linux containers, https://linuxcontainers.org/.

[32] Networkx - high productivity software for complex networks, http://networkx.lanl.gov/index.html.

[33] Wolinsky, D., Liu, Y., Juste, P., Venkatasubramanian, G. and Figueiredo, R. (2009) On the design of scalable, self-configuring virtual networks. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*: 1–12. doi:10.1145/1654059.1654073.

[34] Dong, M. and Zhong, L. (2011) Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11 (New York, NY, USA: ACM): 335–348. doi:10.1145/1999995.2000027.