Improving Communication Through Overlay Detours: Pipe Dream or Actionable Insight?

Stephen Brennan
EECS Department
Case Western Reserve University
Cleveland, USA
stephen.brennan@case.edu

Michael Rabinovich
EECS Department
Case Western Reserve University
Cleveland, USA
michael.rabinovich@case.edu

Abstract—**It has been long observed that communication between a client and a content server using overlay detours may result in substantially better performance than a native path offered by IP routing. Yet the use of detours has been limited to distributed platforms such as Akamai. This paper poses a question – how can clients practically take advantage of overlay detours without modification to content servers (which are obviously outside clients' control)? We have posited elsewhere that the emergence of gigabit-to-the-home access networks would precipitate a new home network appliance, which would maintain permanent presence on the Internet for the users and have general computing and storage capabilities. Given such an appliance, our vision is that Internet users may form cooperatives in which members agree to serve as waypoints points to each other to improve each other's Internet experience. To make detours transparent to the server, we leverage MPTCP, which normally allows a device to communicate with the server on several network interfaces in parallel but we use it to communicate through external waypoint hosts. The waypoints then mimic MPTCP's subflows to the server, making the server oblivious to the overlay detours as long as it supports MPTCP.**

Index Terms—**Download performance, overlay detours, MPTCP**

I. Introduction

Numerous studies have observed that communication between two Internet hosts can often be improved if the hosts sent traffic via relay hosts, rather than directly on a native path offered by IP routing [1]. The overlay *detour* paths produced by the relay hosts traversal often have less packet loss [2], lower latency [3], and higher bandwidth [4], due to inefficiencies in native IP routes. Moreover, past studies have shown that most performance benefits can be obtained by using a single waypoint between the end-hosts [2], [5]. However, despite these insights being known for almost two decades, the use of detours has been limited to communication within distributed platforms, notably, Akamai, which adopted detours as the foundation of their "SureRoute" technology [6]. At the same time, with emerging gigabit-to-the-home access networks, the communication bottleneck will shift upstream from the last mile, and likely into the core Internet. Utilizing well-connected waypoints promises significant performance benefits to Internet users in this environment, especially if users can utilize multiple detour paths in parallel to speed up their content accesses.

This paper poses a question – how can clients practically take advantage of overlay detours, when downloading content from the Internet, without modification to content servers (which are obviously outside clients' control)? We propose a framework to leverage overlay detours in an application-transparent manner and demonstrate its operation using an unmodified off-the-shelf networked application (iPerf). Our vision rests on the assumption we posited elsewhere [7] that to realize full potential of the emerging ultra-broadband Internet, home networks would benefit from a new appliance, potentially packaged within the wifi router form factor, which would have general computing and storage capabilities. Given such an appliance, we envision users forming cooperatives in which members agree to serve as waypoints to each other to improve each other's Internet experience – in other words, creating peer-to-peer systems where home appliances of some users serve as detour waypoints for end devices of other users. To make detours transparent to content servers, we leverage MPTCP, which normally allows a device to communicate with a server on several network interfaces in parallel, but we use it to communicate through external waypoint hosts. The waypoints then mimic MPTCP's subflows to the server, making the server oblivious to the overlay detours as long as it supports MPTCP.

We recognize that very few content servers currently support MPTCP. However, there are signs that this might be changing. For instance, Apple has employed MPTCP for its Siri service [8]. We assume that as compelling use cases such as the one addressed in the present paper emerge, the deployment by content servers will grow. In the meantime, IETF is working on a proposal to deploy MPTCP proxies within the network, which would allow MPTCP-adopting clients to benefit from MPTCP even with interacting with a non-MPTCP server, by proxying their communication through an MPTCP proxy in server's vicinity [9], [10]. Our approach can be used in this deployment scenario as well, by establishing subflows with the MPTCP proxy.

MPTCP has been used for bandwidth aggregation over available access links (most commonly, wifi and cellular) in a mobile device, for communication within a datacenter [11]

This work was supported in part by NSF through grant CNS-1647145.

and for Internet gateway aggregation [12] in a rural setting. Our approach represents a novel use of MPTCP, namely, to make overlay detours possible, so that a client can explore alternative wide-area routes and aggregate multiple such routes if possible.

II. Vision

We assume that ultra-broadband Internet home networks would deploy a server-like appliance that would maintain permanent Internet presence for the users at the residence and help realize full potential of ultra-broadband – the appliance we called "home point of presence" [7], or HPoP. With this assumption, we envision that users form a *detour cooperative* to improve each other's Internet performance. By joining the collective, they offer their HPoPs as waypoints which other members may use. In return, they gain access to the detouring services of the other members of the collective. In effect, the collective forms an overlay peer-to-peer network that unmodified TCP applications may use.

To join the cooperative, a member installs certain application-level components, as well as our patched kernel with modified network stack, on their client machine. In principle, a client can engage another member as a waypoint in its communication path with the server using an arbitrary custom protocol, since all coop members install our custom patch (we later discuss some particularly simple mechanisms for this engagement). The waypoint, however, mimics the behavior of an MPTCP subflow when communicating with the server.

When the data mostly flows from the server to the client, the client establishes subflows to the server through waypoints, but it is still up to the server to split the flow among the waypoint(s) and the original direct path to the client. The client can still have indirect control over waypoint use by withdrawing poorly performing waypoints and adding new waypoints during the TCP session, as well as by other manipulations (see Section VI). When the data flows mostly from the client to the server (e.g, in video clip uploads, video-conferencing, or other increasingly common cases of user-generated content), the client can directly explore different waypoints by sending a few data packets over new subflows and staying with whose waypoints that perform well.

In both cases, our mechanism is able to explore different waypoints to find efficient overlay detours, and further to aggregate bandwidth of several available paths. Most importantly, since MPTCP presents the same binary-compatible OS level API as TCP, unmodified applications may use this mechanism simply by using our patched kernel.

A. Security and Privacy

Inserting a waypoint into a communication path introduces potential security and privacy concerns. However, modern Internet applications increasingly use TLS. The client should perform TLS handshake on the initial subflow, over the direct path, before establishing any other subflows. Subflows through detours will then be encrypted, so the waypoints will be unable to read any communication. A waypoint still learns the IP addresses with which the client is communicating. While this information already flows openly through the Internet in the unencrypted IP headers, our approach makes it readily available to the waypoints involved. This is an inherent cost of our approach.

A malicious waypoint could also disrupt its subflow in arbitrary ways, but as long as TLS is negotiated before any auxiliary subflows are established, the application would notice, and presumably withdraw this waypoint, while transparently recovering (within TCP) the affected packets over remaining subflows. Furthermore, the misbehaving peer can be expelled from the collective to avoid affecting future communication of any member.

III. Related Work

Historically, the communication bottleneck has been most commonly located at the last mile, and extensive effort has been spent on alleviating this bottleneck, going back to *channel bonding* technology for ATM networks [13]. However, with the emergence of gigabit access networks, the bottleneck shifts upstream, and it has been shown the effective download TCP throughput that gigabit users obtain is orders of magnitude lower than their access link capacity [14]. We are trying to address this new reality.

Our approach essentially builds one-hop overlay paths between clients and servers at the transport layer. Numerous previously proposed overlay or peer-to-peer networks typically operate at the application layer and are not transparent to the application at both sides of the connection. Even when their functionality is encapsulated within a runtime library, such as in RON [15], applications still have to link to the custom library and thus undergo a change. The same holds for mTCP [16], a pre-MPTCP multipath variant of TCP built in user space on top of RON. Gummadi et al. describe a system for single-hop source routing (SOSR) [2] that is transparent to websites. Unlike our approach, it is unable to aggregate bandwidth across multiple paths or switch paths in the middle of a TCP session.

Some applications provide means for parallelizing communication at the application level. For example, a client could use members of the collective we envision as proxies for parallel download of a static HTTP object using HTTP range requests. By enabling detours at the transport layer, we allow our framework to be used by *any* application. Even in HTTP, accesses to dynamic resources are often non-idempotent, making these resource not amenable to parallelization via range requests.

IP source routing [17] allows path exploration at the IP layer. However, due to security concerns, it is recommended that routers and firewalls drop packets with source-routing options [18], and many routers on the Internet do.

IV. Architectural Framework

DCol involves three types of hosts:

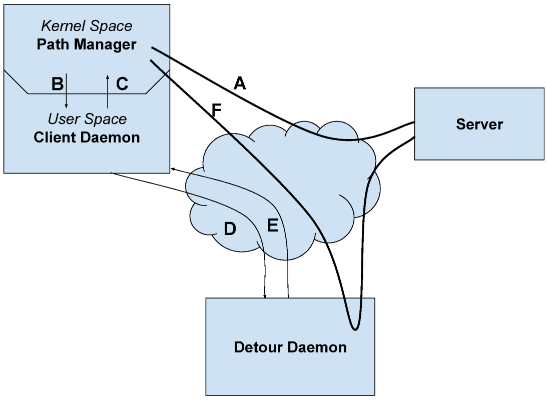

Fig. 1. High-level architecture of DCol. In step A, the initial subflow across the default route is created. In step B, the path manager requests an overlay route. In step C, the client daemon reports overlay routes back to the path managers. In steps D and E, the client and detour daemons negotiate an overlay route. In step F, a subflow is created through the detour.

- The *client* is the active opener of the MPTCP connection. In our system, the client actively requests all detours and initiates all subflows.
- The *server* is the passive opener of the MPTCP connection.
- The *waypoint* is a host which acts as an intermediary along the overlay detour from the client to the server. There may be more than one detour at a time, and there are multiple ways to implement the detour service. However, each detour only passes through a single waypoint.

We assume that the server supports Multipath TCP but employs no additional customization. The clients and waypoints are customized to use our system. While we describe clients and waypoints as separate entities, DCol members would typically combine both roles. This paper focuses on basic architectural framework for exploiting overlay detours. The take-away point from the discussion in this section is that key building blocks for DCol already exist and setting up such a collective involves mostly integration of existing technologies. Given the groundwork described in this paper, the main direction for future work is developing and implementing strategies for exploring candidate waypoints and for controlling how traffic is split among them.

Figure 1 depicts a high level view of DCol architecture. The DCol components on a client includes a modified MPTCP path manager in the kernel and a user-space *client daemon*. A waypoint deploys a user-space *detour daemon*. Engaging a waypoint for the communication between the client and server involves the following steps. When an application opens a TCP connection to an MPTCP-enabled server, the client creates an MPTCP connection to the server. The first subflow uses the Internet routed path. Once this connection is fully established, the path manager communicates with the client daemon to request overlay routes for additional subflows. The client daemon receives this request, and selects candidate waypoints for detours[1]. The client and detour daemons on selected waypoints negotiate their respective tunnels[2]. The client reports the successfully negotiated tunnels to the kernel, which creates subflows on these tunnels to explore the corresponding detours according to some policy[3]. We discuss each of these components in more detail below.

We have implemented a proof-of-concept prototype of our approach. The prototype is built on top of Linux MPTCP implementation [19] version 0.91 and includes an extension to the MPTCP path manager on the client side and user-space daemons on both the client and waypoint sides.

A. Client-to-Waypoint Tunneling

We have implemented two different mechanisms for tunneling MPTCP subflows between the client and waypoint, leveraging VPN tunneling and using network address translation at the waypoint. Furthermore, our DCol client incorporates both alternatives, so it can use either technology for individual detours depending on which tunneling type is supported by the corresponding waypoints.

1) VPN Tunneling: VPN technologies use packet encapsulation to tunnel VPN clients' traffic through a VPN server. We can leverage this functionality to tunnel traffic between our clients and waypoints. We use OpenVPN [20], an open-source VPN implementation that offers encapsulation of packets into UDP datagrams addressed between the VPN client and server among tunneling options, as the basis of our exploration of this mechanism.

Our setup is shown in Figure 2. A waypoint runs an OpenVPN server, which offers a DHCP service for its virtual subnet (its VPN). To use a waypoint for any detours, the client daemon creates a virtual interface and join the corresponding VPN by running DHCP to acquire an IP address on the virtual subnet. The client daemon further sets up a routing rule for this interface with high cost to all destinations. This prevents the route from being used for unrelated traffic but keeps it available for the path manager to route specific subflows as needed. The client daemon informs the path manager of the available VPN tunnels, which the path manager can use to create MPTCP subflows.

Once a waypoint receives and decapsulates a packet from a client, the waypoint must rewrite the packet's source IP address to contain the waypoint's own address before forwarding the packets to the Internet. This ensures that the source address is routable and that the return traffic will be routed back to the waypoint. While this functionality is typical for VPNs that provide Internet access, it is not offered by OpenVPN itself. Consequently, we configure waypoints (via netfilter)

[1] The waypoints are currently selected from a static configuration file but in reality would be selected according to some policy from the current members of the collective

[2] Waypoints behind NAT boxes make themselves available for incoming tunnel requests using standard UPnP or STUN mechanisms.

[3] The prototype currently uses a simple policy that creates a subflow across every available waypoint, until a predefined limit is reached.Our implementation uses two subflows as the default limit, but it may be changed by an administrator using the `sysctl` tool.

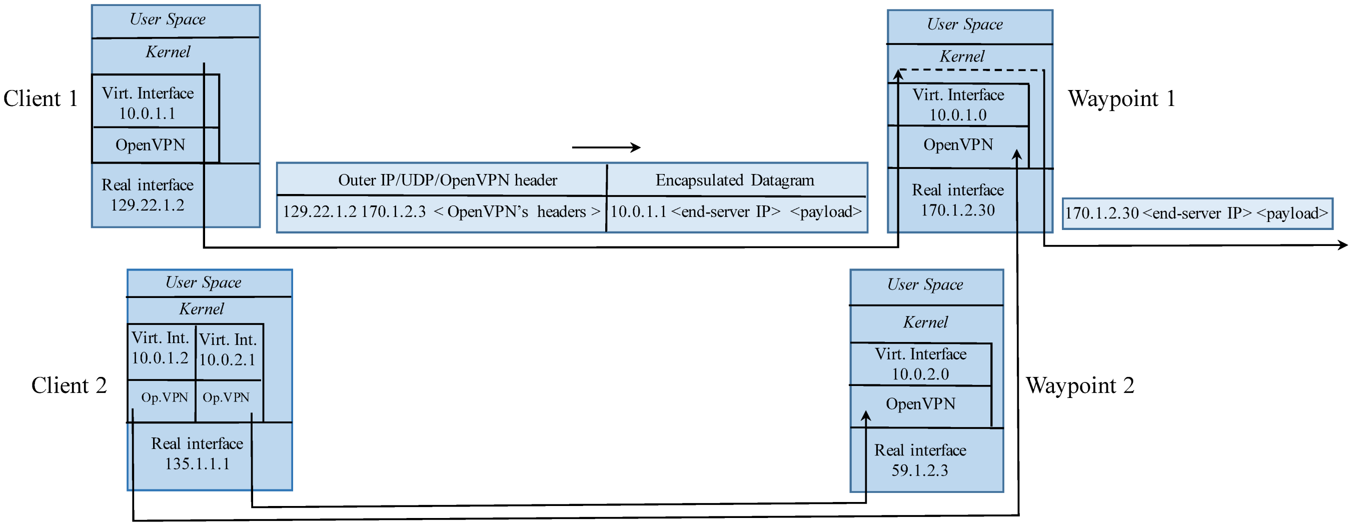

Fig. 2. VPN-based client-to-waypoint tunneling. Client 1 is connected to waypoint 1 and client 2 to both waypoint 1 and 2. In the tunneled traffic from a client to the waypoint, the inner datagram headers carry the client's private IP address as source, and the end-server's IP address as destination IP addresses. The outer datagram header carry the IP addresses of the tunnel end-points.

to perform network address translation (NAT) on outgoing packets. Similarly, the incoming packets get their destination addresses rewritten from the waypoint's real address to the client's private address.

To connect to multiple waypoints simultaneously, the client would create multiple virtual interfaces and join multiple VPNs. Thus, care must be taken to avoid addressing conflicts between the VPNs. The easiest way to achieve this is to make each waypoint use a distinct address block for its private IP subnet. For instance, using the 10.0.0.0/8 unroutable address block, if each waypoint were to establish its own /26 subnet (sufficient to detour traffic of 64 clients simultaneously), there would be capacity for 256K non-conflicting subnets hence members of the collective. In a large collective, these subnet allocations could be handled by a centralized collective management server, or an appropriate distributed protocol among the members. In our prototype, we assign subnets to waypoints manually.

Finally, note that OpenVPN requires the client and server to exchange certificates during the initial connection negotiation. This mutual authentication is actually beneficial for DCol because it can be used to (a) protect both the client and waypoint from working with an imposter counterpart and (b) provide accountability for misbehaving clients and waypoints. However, DCol's OpenVPN configuration does not encrypt or sign subsequent messages, since these protections are orthogonal to DCol's functionality. Presumably the payload is already protected through end-to-end TLS negotiated between the client and server as discussed earlier.

2) NAT Tunneling: The second approach directly modifies packet source and destination addresses. Rather than encapsulating packets and sending them to the waypoint, the client daemon instead informs the waypoint's detour daemon of the intended destination, and requests that it perform NAT on its packets. The waypoint sets up a NAT rule and allocates a port on which it would receive packets from the client intended to the specified destination. Then, the client's kernel addresses packets directly to the waypoint, on the agreed upon port. The waypoint forwards these to the final destination, and forwards reply packets to the client. On packets destined to the server, the waypoint rewrites the destination address and port number to be the server's, and rewrites the source address and port number to be its own. For reply packets, the waypoint rewrites the destination address and port number to be the client's, and rewrites the source address and port number to be its own. We utilize standard NAT facilities offered by Linux within its netfilter framework to set up the NAT rules, with no custom code needed for packet forwarding.

We reiterate that the NAT solution requires the client daemon to use a signaling protocol to communicate to the waypoint the ultimate destination's address and port number and get from the waypoint the port number allocated for the tunnel and a conformation from the waypoint that the corresponding NAT rules have been installed. Only after this signaling completes can the client daemon make this waypoint available to the client path manager.

The detour daemon is implemented in our prototype as a Python script.

3) Discussion: There are interesting tradeoffs between VPN and NAT tunneling. On one hand, an established VPN connection may be used as a detour for any TCP connection to any end-server, without any additional setup. The NAT mechanism requires signaling with the waypoint for every new server address and port number combination. On the other hand, VPN adds 36 bytes of per-packet overhead for IP encapsulation and UDP and OpenVPN headers[4]. The NAT

[4] To avoid fragmentation, OpenVPN accommodates this overhead by automatically reducing the maximum segment size accordingly.

mechanism adds no extra bytes to a packet. Further, VPN tunneling provides a built-in mechanism for mutually authenticating the client and waypoint. The NAT implementation offers no such protection. In fact a naïve signaling protocol consisting of a simple exchange of UDP request/response messages that we use in our DCol prototype would expose a number of obvious security vulnerabilities. Our simple protocol suffices to assess the efficacy of the NAT-based approach, but a more secure signalling mechanism would be needed in a realistic implementation. We discuss this issue further in Section VI.

B. DCol Client

A DCol client consists of a modified MPTCP path manager (a kernel module) and a client daemon (a user-space component). The two components communicate over a generic netlink, a commonly used Linux facility for communication between the kernel and user space [21]. Our implementation allows a given DCol collective to mix and match VPN and NAT tunnels, depending on waypoint preferences.

The DCol path manager performs the following basic functions:

- It maintains two lists of available waypoints, one for waypoints reachable through NAT tunnels and the other through VPN tunnels. As explained earlier (Section IV-A), NAT tunnels are specific for a given MPTCP connection and VPN tunnels are generic, good for any connection detouring via the corresponding waypoint.
- It requests new detours from the client daemon as needed. In our current prototype, the path manager requests new detours any time a new MPTCP connection is established; a real implementation would use a more flexible policy, which could request new detours whenever the available detours become scarce.
- It selects and adds detour subflows to MPTCP connections, as well as withdraws existing subflows according to a detour exploration strategy[5].

The client daemon receives requests for detours from the path manager, negotiates tunnels with the waypoints, and reports successfully established tunnels back to the path manager. In a real implementation, the client daemon would need to discover available waypoints in the collective and choose the ones to set up tunnels with.

Since the same VPN tunnel can be (re)used for any connection to any server, the client daemon can establish it either in response to request from the path manager or proactively, and inform the path manager of its availability ahead of the need. In contrast, a NAT tunnel can only be used for a particular destination's IP address and port number, which must be communicated to the waypoint during the tunnel establishment. Thus, NAT tunnels are established on an explicit request from the path manager.

The above considerations may lead to intricate policies for detour selection and tunnel establishment. With our focus on the basic groundwork, our prototype uses the following simple setup. The daemon has a configuration file which lists the IP addresses of waypoints to use for both types of detours.

On startup, the daemon creates a netlink socket to communicate with the path manager, starts an instance of the OpenVPN client process for each available VPN waypoint, and waits for all VPN tunnels to fully initialize. A new thread is used to monitor each OpenVPN client, logging any relevant messages.

Next, the daemon opens a UDP socket corresponding to each NAT waypoint in its configuration file. These sockets will be used to negotiate tunnels upon a request from the path manager.

Finally, the daemon reports all OpenVPN clients to the manager, and begins waiting for requests. For each request from the path manager, the client daemon sends a UDP request to every NAT waypoint. For each received response, the daemon sends the final NAT detour information (the IP addresses and port numbers of the tunnel's endpoints) over the netlink socket to the path manager. Note that a client can use the same NAT waypoint for multiple connections; however, a separate tunnel negotiation is required for each connection.

In summary, our prototype implements a primitive client daemon that makes available to the path manager *all* available VPN detours on a startup, and sets up NAT tunnels to *all* NAT waypoints on a first request from the path manager. For another connection, the path manager will send another request, and the daemon will set up another set of NAT tunnels to the same waypoints.

C. DCol Waypoint

A DCol waypoint consists of a detour daemon that listens on requests from clients and invokes local executables to perform requested actions. The detour daemon can operate in two modes - VPN or NAT. In the VPN mode, the detour server simply starts an OpenVPN server. When a client wants to establish a VPN tunnel with the waypoint, the client connects to the VPN server running on this waypoint directly.

Establishing an OpenVPN tunnel requires the two parties to authenticate each other. To this end, each member of a DCol collective has a certificate signed by the a root certificate authority common to the collective. All members store this root certificate – which represents the shared trust among the collective – so they are able to verify the certificates of each other. Once both parties perform mutual authentication using their certificates, the VPN is available for the client to tunnel packets to any IP address.

In the NAT mode, the detour waits for client detour requests on a UDP socket. A request contains the IP address and port number of the end-server the client wishes to communicate with. Upon receiving a request, the detour daemon allocates a unique port to this tunnel, creates the corresponding NAT mapping rule using IPTables, and sends a UDP response with the allocated port back to the client. The client will use this port and the waypoint's IP address as the destination address/port number for the corresponding subflow; in the

[5] This strategy is a major direction for future work. Our prototype implements a trivial policy where the path manager attempts to create and maintain two subflows for each connection.

meantime, the waypoint will use standard Linux NAT functions to rewrite the address and port information as it forwards the packets between the client and the end-server, as described in Section IV-A2.

V. Preliminary Evaluation

It has been well established that one-hop overlay routes can improve connection latency, loss rate, and throughput [1], [2], [15]. Further, discovering and exploring waypoints for advantageous detours is a major next topic and is outside the scope of the present paper. The goals of our preliminary evaluation are as follows:

- Given a network where an alternative path with better connection properties exists, demonstrate the ability of DCol to automatically and transparently leverage the alternative path to improve communication throughput;
- Given a network where the core rather than access links, is the bottleneck, demonstrate the ability of DCol to automatically and transparently aggregate the throughput of alternative paths;
- Assess the overhead imposed by DCol compared to standard TCP across the default route or a detour;
- Examine the difference in performance of the NAT and VPN detours.

A. Methodology

We use network emulation based on Mininet [22], a tool that allows emulation of large networks on a single test machine using the native operating system networking stack. It has been noted that Mininet timing fidelity can suffer in situations with high load [23]. We verified that our test machine could emulate networks with throughput up to 24 Gbps. To ensure that our experiments did not approach the limits of Mininet's timing fidelity, we limited the bandwidth of each link to no more than 20Mbps, so that the aggregate traffic throughput will be an order of magnitude below the capacity of the emulation environment, and limited each emulated host to using no more than a tenth of the available processor time. As a result, CPU utilization of the test machine was no higher than 5% over the course of each experiment. The rather low absolute bandwidth values do not affect our findings because our goal is to study the relative performance of different approaches and not the absolute performance of any given solution. Further, the lower capacity core links actually reflect a realistic scenario where the network core can only provide a fraction of capacity to the connection in question.

Our network topology is illustrated in Figure 3. The routers, client, server, and waypoint are Linux hosts. The routers have statically configured routes, so that the default route from `client` to `server` traverses Link 2.

To study the effect of a core bottleneck, we consider the following basic network configurations:

- **Symmetric.** Each link has the same characteristics: 10Mbps bandwidth and default 5ms delay (unless overridden by a *delayed* configuration below for link 2).

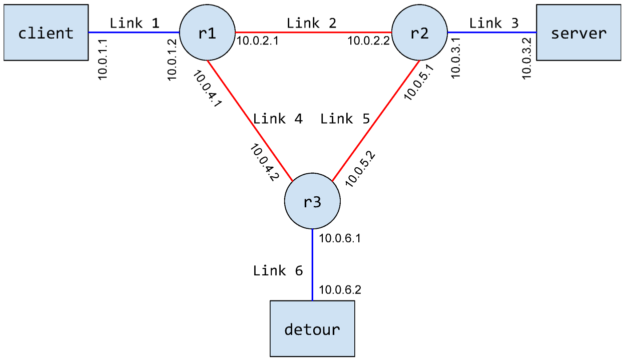

Fig. 3. Experimental network topology. Core links are highlighted in red. Access links are highlighted in blue.

- **Core-limited.** Each link within the network "core" (i.e. links 2, 4, and 5) have 10Mbps of bandwidth and a 5ms delay (unless overridden by the *delayed* configuation for link 2), while the access links (i.e. links 1, 3, and 6) are assigned 20Mbps, with the same 5ms delay. This aims to simulate a situation in which the access link can support more bandwidth than the default Internet path can support.

To consider the effect of a possible disadvantaged direct path, we further consider the following link 2 characteristics:

- **Lossy.** Link 2 is configured to drop packets independently and randomly with probability 0.01. This is within the range of packet loss rates Savage *et al.* observe over "bad" paths in their Detour study [1]. We briefly examined loss rates of 0.5%, 2%, and 5%, with similar results to the ones presented here.
- **Delayed.** Link 2 is configured to have a high (100ms) latency. We also tested latencies of 20ms and 50ms, although 100ms was the first to show observable differences in throughput.

The above settings result in the total of six network configurations: symmetric, symmetric/lossy, and symmetric/delayed, and the same variations of the core-limited configuration. We compare the performance the following communication scenarios:

- **1-Subflow.** MPTCP is enabled, but no detours are configured on the client. The connection uses only one subflow, across the Internet-routed path. This scenario will allow us to assess the inherent overhead of MPTCP over regular TCP. It also represents DCol communication when the client does not use any detours.
- **NAT.** In addition to a subflow on the Internet routed path, a second subflow is through a NAT detour. This scenario represents DCol communication with two subflows, one over the Internet-routed path and the other over a detour with a VPN tunnel.
- **OpenVPN.** In addition to a subflow on the Internet routed path, a second subflow is through an OpenVPN detour. Same as above, but with a NAT tunnel.

- **TCP.** MPTCP is disabled. A regular TCP flow is used, across the default Internet routed path. The default Linux congestion control (CUBIC) is used. This is the baseline case.
- **TCP(NAT).** A regular TCP flow is used, but it is directed through the NAT detour. This another reference case allowing us to see the inherent overhead of the NAT tunnel.
- **TCP(OpenVPN).** A regular TCP flow is used, but it is directed through the VPN detour. Same as above, but with a VPN tunnel.

We use TCP throughput between `client` and `server` to assess the network performance. Throughput is measured with a 10 second iPerf [24] session, which involves transmitting as much data as possible over a TCP (or MPTCP) connection[6]. In our experiments, the TCP flows reached their peak throughput within less than a second. Since MPTCP has a slower mechanism for closing a connection than standard TCP, MPTCP iPerf sessions lasted a few seconds longer than the TCP sessions. During these final seconds, the data throughput slowed down significantly. To focus on steady state performance, we filter out the first second and any remaining time after 10 seconds. Thus, for each trial, we measure the throughput as the number of bytes transmitted during these 9 seconds, divided by 9 seconds.

We ran 60 trials for each experiment and depict the results as box-and-whisker plots. The throughput of each considered scenario is represented by a box outlining the first and third quartiles of the results for individual trials, with a line through the center representing the median. The whiskers show the spread of the data, and points outside of 1.5 times the interquartile range are marked with circles as outliers.

B. Results

Figure 4 shows a performance comparison of the communication alternatives on the *symmetric* network without loss or latency. Since there is no bottleneck at the core in this scenario, MPTCP, as well as tunneling through a waypoint, can only add overhead to the regular TCP connection. The purpose of this experiment is to assess these overheads. We make the following key observations.

First, the overhead of single-subflow MPTCP with no tunneling ("1 subflow") compared to TCP is 130Kbps at the median in this scenario, amounting to a 1.4% loss in throughput. We traced this overhead to the presence of MPTCP's DSS option, the extra 20 bytes that the Linux MPTCP implementation tends to include into every data packet. This is the inherent overhead of our solution.

Second, by comparing the TCP and TCP (NAT) throughputs, we can observe the overhead of NAT tunneling. This overhead is due to processing at the waypoint and the higher-latency communication path, and in particular it gives an upper bound on the processing overhead at the waypoint. We see 200Kbps overhead, or just over 2%.

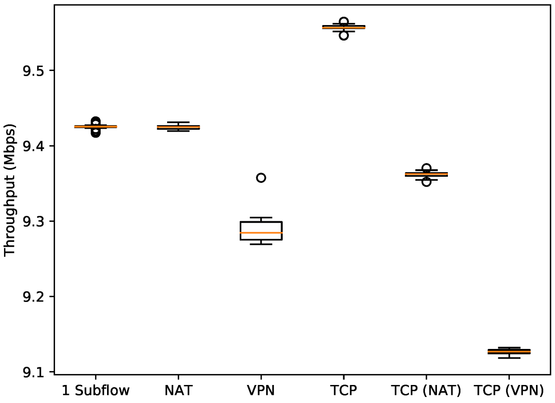

Fig. 4. Throughput comparison: Symmetric network

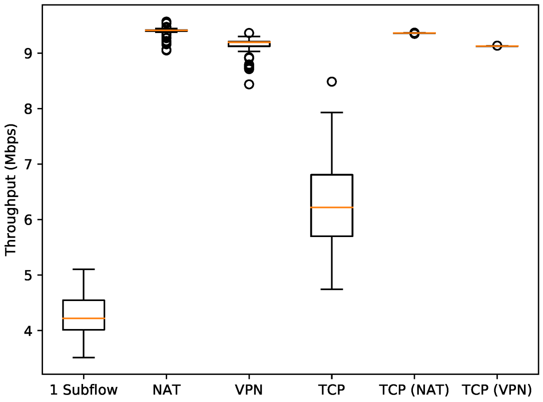

Fig. 5. Throughput comparison: Symmetric network with lossy direct path

Third, by comparing the TCP and TCP (VPN) throughputs, we can observe the overhead inherent in the VPN mechanism when compared to TCP over the default route. We see a greater overhead of about 430Kbps, or 4.5%. This overhead is largely explained by the per-segment extra 36 bytes of OpenVPN headers plus the OpenVPN processing on the waypoint.

Finally, by comparing the throughput of regular TCP ("TCP") with the two-subflow DCol scenarios using NAT and VPN tunnels ("NAT" and "VPN"), we can observe the DCol overhead when the direct path already utilizes full access link capacity and the detour is actually not needed. DCol with NAT tunneling exhibits 140Kbps throughput penalty at the median, or 1.5%, while the VPN tunnel has overhead of 270Kbs, or 2.8%. Note that the overheads are lower than for the TCP(NAT) and TCP(VPN) scenarios because the DCol scenarios use two subflows, one of which (the one utilizing the direct path) does not incur tunneling overhead.

[6] For convenience, we transfer data from the client to the server. Since our current client uses the standard MPTCP subflow scheduler, the performance of data transfer from the server to the client would be similar with the same subflow scheduler.

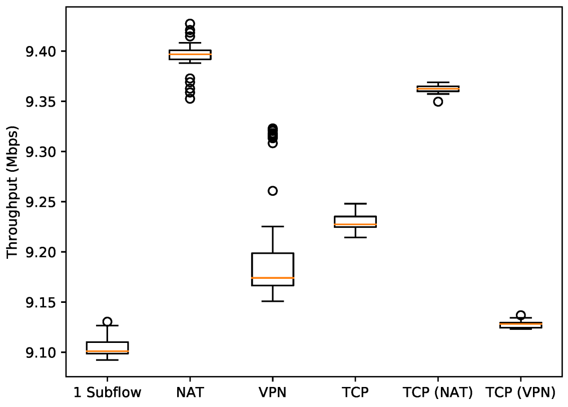

Fig. 6. Throughput comparison: Symmetric network with high-latency direct path

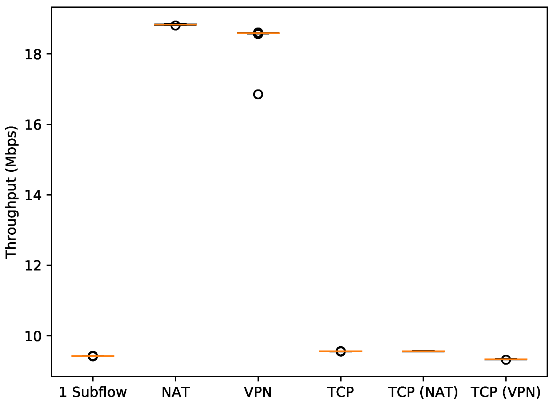

Fig. 7. Throughput comparison: Core-limited network

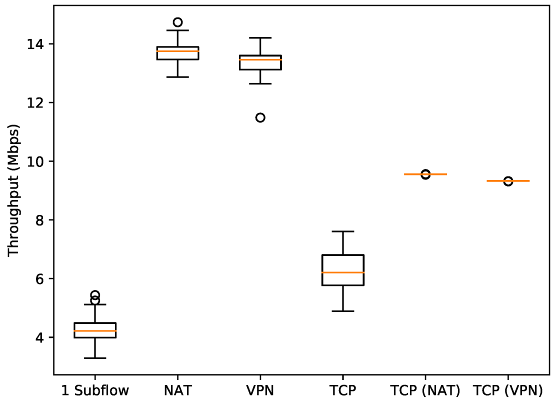

Fig. 8. Throughput Comparison: Core-limited network with lossy direct path

In addition, we note that MPTCP with the added NAT detour ("NAT" scenario) performs virtually identically to 1-Subflow MPTCP, because the overhead in both cases is dominated by the extra 20 bytes for the DSS TCP option. Adding a VPN detour to 1-subflow MPTCP ("VPN" scenario) decreases performance over TCP due to the VPN tunneling overhead, although the VPN scenario improves the performance over TCP (VPN), since the subflow on the default path makes up for some of the VPN tunnel overhead.

We now turn to the scenarios where the direct route between the end-points is disadvantaged in some way. Figure 5, shows the effect of adding a 1% loss rate to the direct path in a symmetric network. Regular TCP shows significant degradation in throughput (by a third at the median compared to Figure 4), and much more variable throughput as well. 1-subflow MPTCP performs even worse, with further 2Mbps throughput loss relative to TCP, because its default LIA congestion control is more conservative than the CUBIC congestion control used by regular TCP. In contrast, all detour scenarios, whether using just a detour (which has no packet loss), or the detour in addition to the subflow over the direct path, are virtually unaffected by the lossy link. Any overhead noted earlier in Figure 4 pales in comparison to the advantage of bypassing the lossy link. Overall, the DCol scenarios achieve roughly 50% higher throughput than regular TCP in this environment.

In Figure 6, we see the impact of the high latency link along the direct route, still in a network with otherwise symmetric link capacities. The added latency does not affect the performance nearly as strongly as the packet losses, with all alternatives achieving between 9.1 and 9.4 Mbps throughput. Still, by sending a significant portion of traffic over the low-delay path[7] DCol with NAT tunneling ("NAT") slightly outperforms regular TCP. For the alternatives with VPN tunneling (both detour-only TCP(VPN) and DCol's two-subflows VPN), the benefit of bypassing the high-delay link does not fully compensate for the VPN overhead, resulting in slightly lower performance relative to regular TCP. Nonetheless, DCol's throughput with VPN tunneling is within 2% of DCol with NAT tunneling.

The results for the core-limited network are much different. Figure 7 shows a comparison of throughput on the core-limited network, with no loss or latency. Since the detour path traverses different core links than the default path, there is a potential to aggregate the capacity of both paths. The figure shows that DCol is able to effectively aggregate the capacity of both paths: the two-subflow scenarios using either NAT or VPN tunnels achieve almost double the throughput of regular TCP. Said differently, the two-subflow DCol with the NAT detour achieves 98.4% of the sum of the throughputs of TCP and TCP (NAT). The two-subflow DCol with VPN detour obtains a 98.3% of the sum of the throughputs of TCP and TCP (VPN).

[7] We use MPTCP's standard lowest-RTT-first scheduler. We checked the division of data between subflows in one of the runs and found that 82.3% of the data was across the lower-delay detour.

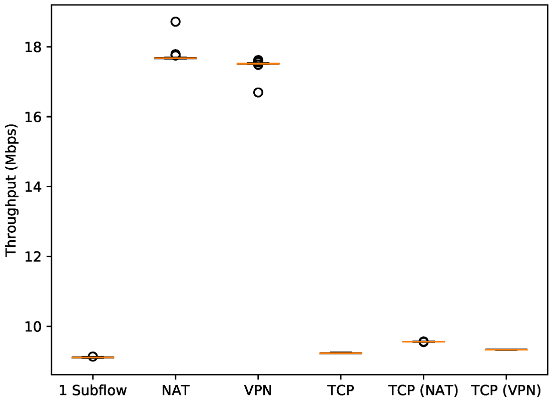

Fig. 9. Throughput comparison: Core-limited network with high-latency direct path

When loss is introduced, Figure 8 shows that all scenarios using a detour significantly outperform regular TCP over the direct path. When only using the detour path, TCP achieves roughly 50% higher throughput than regular TCP (detour's 9.33 or 9.56Mbps, depending on the tunnel used, vs. regular TCP's 6.27Mpbs at the median). With two subflows, DCol's performance gain exceeds the factor of two since, even though the direct path is lossy, it is able to contribute some throughput to the MPTCP connection.

Similarly, when latency is introduced, Figure 9 shows that DCol continues to aggregate throughput across the direct and detour paths. As in the case of the symmetric network, the effect of added latency on the throughput is slight, and the overall performance trends are the same as in Figure 7

C. Summary of Results

When no detour is needed, DCol exhibited an overhead over regular TCP in our experiments of just over 1.5% with NAT tunneling and 2.8% with VPN tunneling. This overhead is quickly overcome when a better detour than the direct path can be found. When path bandwidth aggregation is possible, DCol is capable of aggregating over 98% of the sum of the throughputs achieved by TCP across both paths.

Across all the scenarios, DCol with NAT tunneling has slightly higher performance than with VPN tunnels, due to the additional overhead of the IP and UDP encapsulation, along with OpenVPN protocol overhead and latency.

VI. Future Work

The proof of concept prototype an architectural framework for utilizing detours in an application is but the first step towards the vision this paper presents. In addition to a number of directions for future work mentioned throughout the paper, a fundamental question is the strategy in finding beneficial detours and selecting the ones to use for a given connection. There has been much work on routing in overlay networks and finding detours (e.g., [5], [25]) but our vision brings this question back to the fore. In particular, much of prior work in this area has focused on selecting detours based on predicted path latencies, while our preliminary study indicates that packet loss and capacity of a path, which are much harder to predict without an ad-hoc measurement (and harder to measure), play a dramatically greater role.

A related issue is data scheduling among subflows. While our prototype uses default MPTCP scheduler, one can easily see its limitations for our purposes. First, the current scheduler can only affect data sent from the client to the server, while data often flows in the other direction. Since the default schedules use RTT as a key factor in their scheduling policy, a custom client's scheduler can indirectly affect the server's scheduler by delaying subflow-level acknowledgements and thus modifying the RTT values seen by the server.

For the data sent by the client, one could explore a number of scheduling strategies, such as sending data redundantly over new subflows so that there is no performance penalty when assessing the characteristics of a new subflow, estimating the packet loss and capacity of the subflow from the traffic dynamics and using these factors in scheduling decisions, sending only small amount of data on new subflows until their performance has been assessed, communicating subflow performance to the path manager, which could decide to close underperforming tunnels and/or request new detours of the client daemon, etc.

Another issue concerns traffic attribution. Similar to exit nodes in Tor [26], waypoints act as exit hosts for someone else's traffic and appear externally as the traffic originators. Depending on the nature of the traffic, this can be undesirable for the exit host. Fortunately, unlike Tor, whose goal is to provide anonymity to the clients, DCol does not hide client identity, and a detour always knows the client responsible for a given connection. This should provide a basis for an extension to tunnel negotiation that would give the waypoint Irrefutable proof of attribution. In fact, OpenVPN authentication process already provides such proof for VPN tunnels. NAT tunnel negotiation would need to be endowed with a similar mechanism.

A further optimization appears possible in regard to NAT tunnel negotiation, which in our current prototype is done as a separate UDP exchange before each subflow. Following the approach used in IETF's MPTCP proxying draft [9], we could instead piggyback this negotiation on TCP control segments involved in the subflow establishment. The initial SYN segment of a new subflow to a waypoint would be addressed to a default port on the waypoint and embed the final destination's IP address and port number into a special TCP option. When forwarding the SYN-ACK segment back to the client, the waypoint would include, also into a TCP option, the waypoint's port number for subsequent segments on this subflow.

Finally, a system for maintaining the membership in the DCol collective is needed for a real deployment, which would allow hosts to join and depart the system and would provide machinery to disseminate membership data to the individual members so they could use this information to select candidate waypoints. Given the amount of work addressing similar issues in peer to peer systems, this represents a significant but technical challenge, with rather clear ways to implement the required functionality.

VII. Conclusion

It has long been observed that communication between a client and a content server using overlay detours may result in substantially better performance than a native path offered by IP routing. With the emerging gigabit-to-the-home access networks, where the bottleneck is no longer at the last mile, the potential benefits of detours are likely to only grow. This paper presents our vision of how to achieve detour communication in a generic way, without requiring any changes to the applications or content servers.

We envision gigabit Internet users to form cooperatives in which members serve as detour waypoints to each other's Internet experience. To make detours transparent to the server, we leverage MPTCP, which normally allows a device to communicate with the server on several network interfaces in parallel but we use it to communicate through external waypoint hosts. The waypoints mimic MPTCPs subflows to the server, making the server oblivious to the overlay detours as long as it supports MPTCP. At the same time, because all functionality concentrates within the transport layer, network applications using TCP automatically benefit while being also unaware of this change.

We present our architectural framework, DCol (for "Detour Collective"), and a proof-of-concept prototype that shows that key building blocks for DCol already exist and setting up usch a collective involves mostly integration of existing technologies . Given the groundwork described in this paper, the main direction for future work is developing and implementing strategies for exploring candidate waypoints and for controlling how traffic is split among them.

References

[1] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorian, "Detour: Informed internet routing and transport," *Ieee Micro*, vol. 19, no. 1, pp. 50–59, 1999.

[2] P. K. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, D. Wetherall *et al.*, "Improving the reliability of internet paths with one-hop source routing." in *OSDI*, vol. 4, 2004, pp. 13–13.

[3] H. Zheng, E. K. Lua, M. Pias, and T. G. Griffin, "Internet routing policies and round-trip-times," in *PAM*, 2005, pp. 236–250.

[4] S.-J. Lee, S. Banerjee, P. Sharma, P. Yalagandula, and S. Basu, "Bandwidth-aware routing in overlay networks," in *INFOCOM*, 2008, pp. 1732–1740.

[5] C. Lumezanu, R. Baden, D. Levin, N. Spring, and B. Bhattacharjee, "Symbiotic relationships in internet routing overlays." in *NSDI*, 2009, pp. 467–480.

[6] Akamai, "SureRoute," https://developer.akamai.com/learn/Optimization/SureRoute.html.

[7] M. Allman and M. Rabinovich, "Rethinking home networking for the ultrabroadband era." [Online]. Available: https://www.nsf.gov/awardsearch/showAward?AWD_ID=1647145

[8] O. Bonaventure and S. Seo, "Multipath tcp deployments," *IETF Journal*, vol. 12, no. 2, pp. 24–27, 2016.

[9] M. Boucadair, C. Jacquenet, D. Behaghel, stefano.secci@lip6.fr, W. Henderickx, R. Skog, O. Bonaventure, S. Vinapamula, S. Seo, W. Cloetens, U. Meyer, L. M. Contreras, and B. Peirens, "Extensions for Network-Assisted MPTCP Deployment Models," Internet Engineering Task Force, Internet-Draft draft-boucadair-mptcp-plain-mode-10, Mar. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-boucadair-mptcp-plain-mode-10

[10] M. Boucadair and C. Jacquenet, "RADIUS Extensions for Network-Assisted Multipath TCP (MPTCP)," IETF Draft, Tech. Rep., Oct. 2017. [Online]. Available: https://tools.ietf.org/id/draft-boucadair-mptcp-radius-05.html

[11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 266–277.

[12] L. Boccassi, M. M. Fayed, and M. K. Marina, "Binder: A system to aggregate multiple internet gateways in community networks," in *Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access*, ser. LCDNet '13. New York, NY, USA: ACM, 2013, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2502880.2502894

[13] J. Duncanson, "Inverse multiplexing," *IEEE Communications Magazine*, vol. 32, no. 4, pp. 34–41, April 1994.

[14] M. Sargent and M. Allman, "Performance within a fiber-to-the-home network," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 22–30, 2014.

[15] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, *Resilient overlay networks*. ACM, 2001, vol. 35, no. 5.

[16] M. Zhang, J. Lai, A. Krishnamurthy, L. L. Peterson, and R. Y. Wang, "A transport layer approach for improving end-to-end performance and robustness using redundant paths." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 99–112.

[17] J. Postel *et al.*, "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc791.txt

[18] F. Gont, R. Atkinson, and C. Pignataro, "Recommendations on Filtering of IPv4 Packets Containing IPv4 Options," RFC 7126, Feb. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7126.txt

[19] C. Paasch, S. Barré *et al.*, "Multipath TCP in the Linux Kernel," http://www.multipath-tcp.org.

[20] J. Yonan *et al.*, "OpenVPN," http://openvpn.net.

[21] "generic_netlink_howto," Linux Foundation Wiki, Tech. Rep. [Online]. Available: https://wiki.linuxfoundation.org/networking/generic_netlink_howto

[22] "Mininet: An instant virtual network on your laptop (or other pc)," 2012.

[23] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[24] J. Dugan, S. Elliott, B. A. Mah, and K. Prabhu, "Iperf3." [Online]. Available: http://software.es.net/iperf/

[25] S. W. Ho, T. Haddow, J. Ledlie, M. Draief, and P. R. Pietzuch, "Deconstructing internet paths: an approach for as-level detour route discovery." in *IPTPS*, 2009, p. 8.

[26] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004.