Evolving the Internet with Connection Acrobatics

Catalin Nicutar‡ Christoph Paasch* Marcelo Bagnulo† Costin Raiciu‡

‡ U. Politehnica of Bucharest * U. catholique de Louvain † U. Carlos III de Madrid

ABSTRACT

The textbook Internet architecture revolves around the end-to-end principle with smart endpoints and a dumb network, while the actual Internet is far messier, with middleboxes pervasively deployed and affecting end-to-end traffic in many ways. Today's Internet is fragile as most of the communications are affected by transparent stateful middleboxes deployed along the path. In this paper we propose an evolution of the Internet architecture to make the middleboxes an explicit part of the Internet communications. We do so using the new Multipath TCP (MPTCP) protocol recently standardized at the Internet Engineering Task Force. MPTCP allows us to change the endpoints of the connection and by extension to explicitly add middleboxes in the middle of an ongoing communication. We show that the proposed solution accommodates nicely several widely used use cases including load balancing, DDoS filtering and anycast services. We implement selected use cases as a proof of concept.

Categories and Subject Descriptors

C.2.1[**Computer-Comms Nets**]: Network Architecture

1. INTRODUCTION

Middleboxes form an integral part of the Internet today : operators, major content providers and even certain applications use middleboxes to bypass the shortcomings of the Internet architecture including security, load balancing, in-bound connectivity behind NATs, and so forth. However, middleboxes also make the Internet brittle, difficult to debug and evolve. There is wealth of research to overcome these issues by properly including middleboxes in the Internet architecture, e.g. [20, 8]. However, these solutions depend on changes to the IP layer or upgrades to all applications, none of which seem feasible in the near future.

In this paper we ask the pragmatic question of how to incorporate middleboxes into the Internet architecture in a deployable way. We survey existing proposals and find that deployability severely restricts possible changes. That is why we restrict ourselves to a) making operator-deployed middleboxes explicit in the data path (as opposed to transparent), and b) making provider middleboxes' flow handling more flexible.

HotMiddlebox'13, December 9, 2013, Santa Barbara, CA, USA.

http://dx.doi.org/10.1145/2535828.2535834 .

To support these goals, we leverage the newly standardized and recently deployed Multipath TCP protocol (MPTCP) [7]. To achieve multipath operation, MPTCP implements two key constructs: a per-host opaque connection identifier (not related to IP addresses), and a way to add new addresses to existing connections. These two constructs enable **connection acrobatics**: the ability to explicitly redirect connections via a waypoint, and the ability to migrate the endpoint of a connection. We show that connection acrobatics are *sufficient* to achieve both our goals. Further, we propose the "sticky bit", an optimization to MPTCP that can reduce the redirection overheads for short connections.

Flexibility has a cost: attackers can abuse these same mechanisms to easily hijack connections and move them off-path, especially when the sticky-bit is used. We discuss these vulnerabilities and potential solutions to mitigate them. Finally we explore and implement a number of useful scenarios that are enabled by connection acrobatics in section 6.

MPTCP has been recently implemented and deployed by Apple (though only Siri is using it at this point). This gives us confidence that the acrobatics described here can be deployed in practice. Further, we believe our minor updates to MPTCP should also be easily deployable.

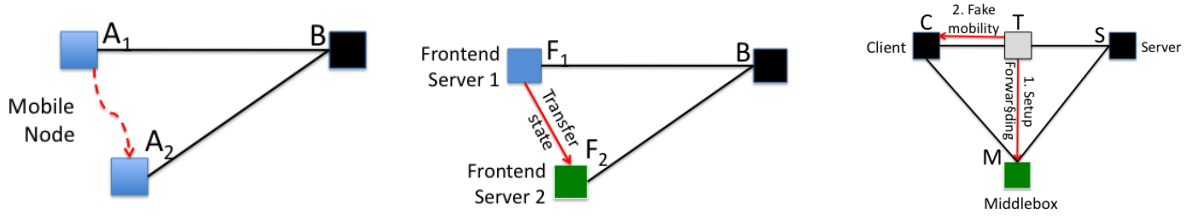
2. PROBLEM STATEMENT

Most of today's connections pass via two middlebox domains. First, "eyeball" networks use routing (e.g. OSPF) and tunneling (MPLS) techniques to steer customer traffic to middleboxes such as firewalls, NATs and performance enhancing proxies. Unfortunately, routing techniques are prone to misconfiguration and do not scale well. Additionally, such middleboxes are transparent to the end-hosts which makes it difficult to debug connectivity problems and reason about end-to-end behavior of protocol extensions [9].

Content-providers use DNS to attract customer connections to front-end servers that terminate the connections, authenticate the customers and load balance each request to the best server. This setup is inflexible: the front-end does not need to be on the data path after client authentication; a better front-end may exist if the client moves.

The research literature has proposed many ways to incorporate middleboxes into the Internet architecture including Delegation Oriented Architecture (DOA) [20] and NUTSS[8]. The key findings are that:

1. Explicit connection signaling is needed to negotiate connectivity through firewalls; this ensures policy compliance and network evolvability as it decouples the control traffic needed to setup the connection from the data traffic (which could use a different protocol).
2. Middleboxes should only process traffic explicitly addressed to them. Having explicit addressing would re-



(a) A canonical IP Mobility Solution (b) Migrating Connections (c) Connection redirection

Figure 1: Connection acrobatics

duce routing complexity and makes it easier to debug the network.

Despite their appeal, none of the existing solutions has been deployed. DOA requires changes to IP, and NUTSS requires changes to the socket API which implies all apps must be changed to take advantage.

Our focus is to implement these principles in a deployable system which has to meet the following two goals: a) applications should not need changing to take advantage of it, and b) it has to be compatible with the current network.

A fundamental problem of the current Internet is that the network does not know with certainty what application traffic it is forwarding. Ideally, port numbers should identify the end-host application but the trend of consolidating all traffic on few ports (in particular port 80) reduces accuracy drastically. However such knowledge is needed to decide whether traffic should be allowed, or to optimize the traffic, for instance by routing it via application-specific middleboxes.

Ideally, applications should explicitly signal to middleboxes what they want to do, but this is impossible without changing applications. Instead, so we have to make due with the status quo: networks have begun to rely on deep packet inspection boxes that use various heuristics to categorize traffic. In many cases, it is impossible to know during the three-way handshake what app a connection is serving: only after the DPI box sees enough traffic it can accurately identify the application and steer the traffic to the appropriate middlebox. Today redirection is done transparently, without the end-host knowing, inflating the round-trip time and creating mysterious failure modes. Instead, we would like the connection to be explicitly redirected via the middlebox, with the end-hosts' knowledge.

Pragmatically, we need a way to **move the middle of a connection** via a specified waypoint (i.e. a middlebox). This scenario is shown in Figure 1c, where the transparent box T redirects the established connection C-S via M, effectively transforming it in two connections: C-M and M-S. To implement the redirection, box T must change the data plane by re-routing the flow, and will also need to control middlebox M to instruct it to proxy the new flow. Note that the difficulty is in changing the data plane, as the control plane operations in this case are rather simple: as long as M trusts T, SNMP, OpenFlow or any other network configuration protocol can be used to configure the forwarding state at M. Once M sees the traffic, it should be possible to reroute it via another waypoint if it wishes, and so on.

On the server side, DNS will direct the client's traffic to a nearby datacenter where a front-end proxy that will terminate the mobile connection (and maybe TLS) and authenticate the client before passing the request to the real server. There are two issues with this setup: first, the front-end does not need be on the data path after the authentication, and keeping it there increases its load and general complexity. Second when a front-end needs to be taken offline (e.g. for maintenance) all the connections it serves will simply die.

We observe that on the server side the limitations come from TCP's reliance on a fixed set of addresses: a connection cannot be efficiently migrated to a different front-end unless in the same layer 2 domain. Thus, a solution must **move the endpoint of a TCP connection** to allow better load balancing and traffic optimization.

3. SOLUTION SPACE

We have found that explicit signaling cannot be obtained in a deployable way—it requires application changes. Luckily, the goal of making middleboxes explicit should be achievable as long as we can move both the middle and the "ends" of a connection.

We start by observing that any mobility solution can also easily support connection endpoint migration, as long as it is implemented at layer 4 or below. This is quite surprising, as mobility solutions were only built to cope with the devices changing their point of attachment to the network.

To understand why mobility can help, consider a canonical mobility solution as shown in Figure 1a that offers support for a mobile node to continue connectivity despite changing its network attachment point from A_1 to A_2 in the figure. Endpoint migration is easy to implement using this mechanism as shown in Figure 1b:

1. Transfer state from the origin server (front-end 1 in the figure) to the destination server (front-end 2) including TCP connection state (sequence numbers, IP addresses, ports, send and receive buffers, etc) and mobility mechanism state (e.g. keys used for the association with the remote node).

2. Pretend that the Front-end Server 1 has changed attachment point from A_1 to A_2, by using the mobility mechanism. The exact details depend on the mobility mechanism used.

Moving the middle of a connection (as shown in Figure 1c) can also be implemented on top of mobility. The middlebox must trigger two mobility events: to the client, it will appear that the server has moved to middlebox M, while to the server the client will appear to move to M. In effect, T has to fake two mobility events; of course, the protocol mechanisms depend on the mobility solution used, but it should be feasible to do this with any mobility mechanism.

The one caveat for redirection is that the middlebox T may not know the keys used by the client and the server to secure IP mobility. In this case, T has to ask for help from one endpoint, asking it to fake a mobility event (e.g. client moves to M). We discuss a few concrete security issues in more detail in section 5.

3.1 Choosing a mobility mechanism

Mobile IP [15], Locator-ID Separation Protocol (LISP) [5], Shim6 [12] or Host Identity Protocol (HIP) [11] are only a few of the technologies that have been devised to support mobility, so there is a wealth to choose from. More recently, Serval [13] proposes to use service names instead of IP addresses in a bid to optimize the Internet for serving content; Serval also allows connection migration and mobility by design. All these solutions function (mostly) transparently to the transport layer. The upshot is they support redirections for all traffic, not just TCP; the bad part is that, at least for traditional mobility mechanisms, redirections are done in bulk, and not for specific connections, but this should be possible to fix. Even worse, IP-level solutions are oblivious to the transport protocol which can cause TCP performance to drop significantly when mobility events are frequent [17]. Crucially, none of these solutions is deployed—or showing signs of imminent deployment.

There are many proposals that offer mobility at transport level, including SCTP, Migratory-TCP [6] and TCP Migrate [18]. All of them support mobility by adding a per-connection unique identifier independent of the endpoints' IP addresses; this identifier allows the connection to resume when the address changes. SCTP is a mature and feature-rich protocol, yet it has not seen deployment because middleboxes (e.g. NATs) block it. The TCP-based solutions should be deployable, yet they haven't seen adoption partly because of lack of maturity, and partly because of wrong timing—when they were proposed, mobility problems were infrequent in practice. Surprisingly, the newly standardized and deployed MPTCP protocol seems the best bet for an easily deployable solution.

4. ACROBATICS WITH MULTIPATH TCP

Multipath TCP's [7] is an extension of TCP that allows it to efficiently utilize multiple network paths within the same transport connection in a way that is transparent to applications and the network [16]. Because of this requirement, MPTCP allows endpoints to add new addresses to existing connections: every MPTCP connection is given a unique identifier by each endpoint upon creation, which is a hash of a key chosen by that endpoint. The keys are carried in the initial handshake in new TCP-options.

When a new subflow is being added to an existing connection, the MP_JOIN option in the SYN informs the remote end that this is a part of an existing connection, rather then a new one; the option is cryptographically signed using a concatenation of the connection keys. The aim of these mechanisms it to ensure that only the endpoints can add new subflows; however boxes that are on-path for the initial handshake know the keys and can add subflows at will.

The MPTCP standard also allows endpoints to advertise addresses by adding an *ADD_ADDR* option to any segment; the receiving end would then start a new subflow using the advertised address. Note that *ADD_ADDR* options are not protected by the connection key: thus anyone on-path can add an address, not just the endpoints. This can result in a security breach that we address in Section 5.

Multipath TCP's address management mechanisms are sufficient to implement the needed redirection mechanisms. For redirection, the *ADD_ADDR* functionality can be used to redirect traffic as follows. Say an MPTCP connection is setup between C and S (see Figure 1c). T sets up a forwarding rule at M instructing it to proxy all traffic it receives from C. Then, T sends an *ADD_ADDR* message to C advertising M's address. As a result, C will send a SYN+MP_JOIN message to initiate the three way handshake. M receives the SYN+MP_JOIN message, it changes the source address to M and the destination address to S and forwards the message to S. An analogous processing is applied to subsequent packets of the three way handshake. The result is that Both C and S have a new subflow with M, which acts as an explicit middlebox for the MPTCP connection. T can now close the C-S subflow, effectively forcing the endpoints to use only the path via M. Once M sees the traffic, it can direct the traffic to S or D via another proxy by simply using *ADD_ADDR* and setting up the appropriate proxy rules. We have implemented this type of redirection and tested it in practice. The experimental results are discussed in Section 6.

The IOS implementation of MPTCP does not implement *ADD_ADDR* as it perceives it as a security threat [4]. However, redirection with MPTCP is still possible by having the middlebox explicitly initiate two subflows to the endpoints, as follows: M learns the connection keys from T and then sends a SYN+MP_JOIN message to S, mimicking standard subflow creation with MPTCP. S will reply, finalizing the handshake. Similarly, M has to setup a subflow to C by sending a SYN+MP_JOIN. As long as C is not behind a NAT, this type of redirection works equally well.

Connection migration. Connection migration is easy to implement by migrating the MPTCP connection state and opening a new subflow from the new location. However migrating the related application logic is quite difficult. Here, we distinguish two cases: in the first case, the application itself is migrated to the new machine, either by process migration or virtual machine migration, and it just keeps running. In the second case, the connection alone is migrated from one instance of an application to another. If the connection is moved mid-stream, synchronizing the application state obviously requires non-trivial application updates in the general case, and service continuations could be used for this purpose [19]. However, such application changes defeat our deployability goal.

We observe that there are scenarios were connection migration does not require application changes: one example is migrating just after connection startup and before the application receives notification that the connection is ready (i.e. before connect or accept returns). In this case, it is safe to migrate as the application hasn't had the chance to change its state. This functionality can be useful to implement anycast, for instance: the initial handshake is terminated by the load balancer, which then selects a target server and migrates the whole connection there. We have implemented this functionality and show experiments in Section 6.

Another example targets applications that use TLS with the TLS API (*ssl_connect*, *ssl_accept*): it is safe to migrate the connection after the SSL handshake completes but before any data is sent by the application. This use-case re-

quires changes to the TLS library, though, so that the connection keys are migrated too. It can be used to offload the expensive TLS handshake (because of public-key operations) to specialized boxes, and then move the connection to the actual servers.

Optimizations. Redirection with MPTCP requires setting up an additional subflow, which incurs a non-negligible overhead and also delays application traffic. In practice, once a redirection has been made, it is quite likely that traffic to the same server and port will be redirected in the same way in the future—for instance, a large number of HTTP connections are made to the same server in a short period of time, so they should be redirected in the same way.

We can exploit such "connection-locality" by adding a *sticky bit* to the *ADD_ADDR* option. Upon reception of such an option, the receiver will save the address in its local connection cache and use it for future connections: in effect the client remembers to directly use the "optimal" connection. This optimization has security implications that we discuss next.

5. SECURITY ANALYSIS

Connection acrobatics allow redirecting MPTCP connections to middleboxes located anywhere in the Internet, so their implications need to be carefully considered from a security perspective. The design of MPTCP took into account a wide range of security threats [1]. More recently a residual threat analysis of the final MPTCP specification [2] identified a few remaining potential threats as well as the required countermeasures to prevent them.

The design goal for MPTCP security is simple: an ongoing connection can only be redirected by the connection endpoints or other parties explicitly authorized by the connection endpoints to do so. This is achieved by securing the messages that add a new address in the ongoing connection using cryptographic material generated at the beginning of the connection lifetime.

In the current MPTCP specification, the cryptographic material is exchanged in clear during the connection set up. All forthcoming control messages that are used to divert traffic are protected with keys generated from this cryptographic material The result is that MPTCP is vulnerable to attackers that are located along the path during the connection establishment phase. This is similar to the protection SCTP has and it is acceptable as long as only one MPTCP connection is at stake.

The notable shortcoming in the current MPTCP specification identified in [2] is that the *ADD_ADDR* message is not protected and can be used by an off-path attacker to blindly divert the communication to an arbitrary address in the Internet (basically using the same approach we propose for a middlebox to redirect the traffic). The solution to this security breach is to include an HMAC protection using the security token negotiated during the MPTCP connection establishment phase. This would enable the safe operation of the techniques we propose in this paper. The implication is that the middlebox must know the security token in order to generate an *ADD_ADDR* message, but this can be conveyed using the control protocol used to create the forwarding rule in the middlebox as described earlier.

However, when we consider the "sticky" bit as described earlier, it is not only the ongoing connection that is protected by the initial cryptographic material, but also all future MPTCP connections. Thus, the use of the "sticky" bit requires stronger security. This can be achieved by generating the keys used to secure the redirection messages in a more secure way. In particular, it is possible to use SSL keys to do that as described in [14]. Indeed, many data-transfers nowadays rely on TLS/SSL to secure the connection. TLS/SSL negotiate between the two end-hosts a key that is not sent in clear over the network and through server-certificates it is not even possible for a Man-in-the-Middle attacker to interfere with the connection. MPTCP could use the key from TLS to authenticate new subflows which would greatly increase the security of MPTCP as the keys are no longer sent in clear. A proposal to reuse the TLS key to secure MPTCP is described in [14].

In summary, as long as the "sticky" bit is not used, security can be achieved by HMAC-ing the *ADD_ADDR* options. The "sticky" bit is only safe to use when MPTCP's keys are truly secret (e.g. reuse TLS keys).

6. CONNECTION ACROBATICS AT WORK

Using the MPTCP kernel code as basis, we have implemented connection acrobatics. The implementation required very few changes to the MPTCP stack itself, and no changes whatsoever to the network boxes or to the application code, giving us confidence that the changes should be easily deployable. In this section we explore the usefulness of acrobatics by experimenting with three key use-cases and discussing their implications for the Internet at large.

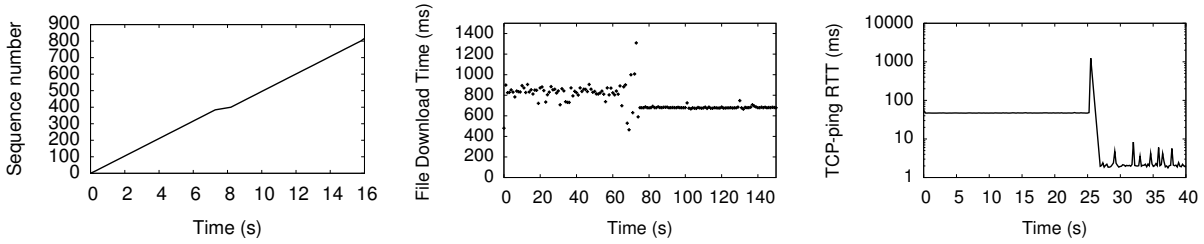
6.1 Connection Redirection

To move the "middle" of a connection, we use a setup where the client and the server have an established long-running TCP connection. Explicit redirection is impossible with vanilla TCP [1], but it is trivial with MPTCP and requires no protocol or kernel changes. Our transparent middlebox T (see figure 1c) runs on a plain Linux box, where our simple user-space program monitors interfaces using *pcap* and records connection details (sequence numbers, IP addresses, ports). To redirect the connection via middlebox M, the program takes these steps:

- T sends connection details to M, which installs forwarding rules to handle the traffic.
- Once M acknowledges the rules have been setup, T injects a TCP keep-alive for the client, carrying an *ADD_ADDR* option advertising the address of the second middlebox.
- The client will open a new subflow to M which uses simple netfilter rules to proxy the packet. This ensures M will be on the path of the new subflow.
- T closes the initial subflow by injecting TCP RST segments in both directions, effectively forcing the client and the server to use the other subflow.

Using this setup, an operator could dynamically redirect traffic to different middleboxes as required by policy. For instance, traffic containing certain keywords could be redirected to a full-blown intrusion detection system for more detailed checks.

[1] An operator can always use routing or tunneling for redirection, but this is both limited in scope and inefficient.



(a) End-to-end traffic is not affected as the connection is redirected from T to M

(b) Using Connection Migration for Load Balancing

(c) Live Migrating a Virtual Machine from Romania to Germany

In our experiment, T scans the TCP payload for the word "secure", and redirects the connection when a match is found. Figure 2a plots the sequence number progression at the receiver, showing that redirection has a negligible effect of performance. Redirection is quick: it takes two round trip times between the middleboxes plus the time needed for creating the subflow; in this example the total redirection time is around 200ms.

Connection redirection enables explicit middleboxes and allows arbitrary middlebox chaining, but it can do more than that. It also enables a few interesting features that are difficult to implement today:

- **Destination Routing Control:** load-balancing for inbound traffic is notoriously difficult to implement in the Internet, leading to hacks such as BGP path poisoning [10]. With redirection, **any box** receiving traffic can choose to redirect it via a waypoint. In figure 1c, M could choose to further redirect the traffic if it wishes, by sending ADD_ADDR itself.
- **Load balancing.** Using redirection, operators can load balance their networks in a fine grained manner, by targeting only long-running connections and redirecting them to proxies in different parts of the network. Alternatives include Equal Cost Multipath Routing (not good when routes are not equal cost) or MPLS-TE, which works on traffic aggregates.
- **Transition to IPv6.** IPv6 deployment is sluggish despite widespread OS support and major network upgrades. The problem is that it's difficult to know whether the IPv6 path works fine. Using redirection, an operator could advertise IPv6 addresses in its own network to make clients send traffic over IPv6. If the server supports IPv6, the connection can continue on IPv6 until the destination; otherwise the hop between M and the server could use IPv4. The advantage of this solution over Happy Eyeballs [21] is that traffic can be migrated back if the IPv6 network is overloaded.

6.2 Migrating connection endpoints

To support endpoint migration (Fig. 1b) we have changed the MPTCP stack, but not the protocol itself. The client connects to an MPTCP-enabled server F_1 which completes the three-way handshake but does not pass the connection to the application. Instead, F_1 transfers the crypto material—the 2 keys exchanged at connection setup—to F_2. At this point F_2 has enough information to accept a new subflow from the client; once the new subflow is setup it can pass the new socket to the waiting process, by returning from the accept syscall. Most importantly, our implementation does not require any application changes. Clients do a normal `connect(2)` to the first server which decides to redirect the connection without involving the application; in turn, the second server uses a simple `accept(2)` as if the client had connected directly.

This simple setup could effectively be used as a low-cost method of coping with flash-crowds. Consider a small web provider that suddenly becomes very popular. Once its server detects it is under load, a copy of the server could be spawned by renting a VM from a public cloud provider such as EC2. To balance the load across the two servers, the traditional approach is to use DNS-based redirection. Unfortunately, it takes a long time for the new DNS records to propagate because of caching in the DNS hierarchy. The web provider could choose very small TTL to limit the amount of caching, but this is costly as it would unnecessarily and permanently increase the load on the provider's DNS server.

Instead, the provider can use acrobatics to perform short term load balancing: with only a few CPU cycles per connection it can complete the three-way handshake; then it will immediately redirect a fraction of the incoming connections to the other server.

To emulate such a scenario, we ran a simple experiment where an Apache server with a gigabit link serves the same 25MB file to two customers that download it over and over again. The plot in figure 2b shows the file download times as time progresses. At around 70s, the server decides to load balance half of its requests to another similar server by migrating connections after the three-way handshake is finalized. The graphs shows that load balancing reduces both the average download time as well as the variance.

Connection migration also enables a few features missing in the Internet architecture today, including:

- **TLS Handshake Offloading.** A more secure version of endpoint migration is to use TLS keys to secure MPTCP, and to migrate a connection after the TLS handshake has finished. This would allow offloading the expensive TLS handshake to specialized boxes, and the connection would be then redirected to the final servers.
- **Off by default!** By using TLS Handshake Offloading, servers could become effectively *off by default* [3]: only successfully authenticated clients would be allowed to contact the destination server. Once a new connection is allowed and thus migrated, the destination firewall will be configured to accept SYN packets with the correct tokens, filtering out all other traffic; the token is similar to a capability in [3]. The TLS infrastructure needs be distributed to ensure it can withstand large scale attacks; however, such a deployment could be from a third party (e.g. Akamai).

- **Anycast.** Load-balancing proxies can move the connection to the chosen server, removing themselves out of the data path after the initial handshake. This allows a (two-stage) anycast that is friendly to TCP; the alternative of using BGP anycast is brittle as TCP connections will break when routing tables change.

6.3 Wide Area Virtual Machine Migration

Virtual machines help increase server utilization while isolating potentially distrusting users. They are one of the key enabling technologies in public clouds, and we are witnessing a shift to also have on-demand general purpose processing in operator networks, in the form of micro-datacenters. Content providers could rent VMs running in micro-datacenters to run frontend proxies as close to the customer as possible.

Virtual machine migration is widely used inside datacenters because it allows load balancing and planned hardware maintenance operations. Migration is great as long as the VM can keep its IP address: otherwise ongoing TCP connections will break. That is why today migration is restricted to the same L2 domain. It would be nice if providers could seamlessly migrate VMs between micro-datacenters in response to changing user behavior.

MPTCP allows wide-area VM migration out-of-the-box: we have successfully migrated an MPTCP virtual machine in the wide-area without any changes to Xen or to the MPTCP stack; we only needed to implement a simple user-space program that installs a Xenstore watch and monitors a key specific to the current Hypervisor. We monitor the Domain ID, which (typically) changes when migrating to a different hypervisor. When the machine is migrated, the monitored key changes and the program triggers an address renewal (a DHCP request). Upon receiving a new address, the MPTCP stack automatically opens a new subflow for each of the ongoing MPTCP connections.

Fig. 2c shows the application level round-trip as measured by a client running in Germany that sends periodic requests over a TCP connection to a virtual machine in Romania. 25 seconds into the experiment, the virtual machine gets migrated to Germany, and the RTT drops from 50ms to around 4ms. The perturbation induced by the migration lasts for a few seconds, increasing RTTs to 1.2s.

Wide area VM migration enables a number of interesting use-cases including the ability to take down an entire datacenter, and the ability to shift processing to follow cheap electricity in diurnal patterns (i.e. follow-the-sun load balancing). Crucially, it allows content-providers to migrate their middleboxes (i.e. front-end servers) when needed without breaking connectivity.

7. CONCLUSIONS

This paper takes a pragmatic step towards embedding middleboxes into the Internet architecture, be they transparent operator-deployed machines or explicit proxies deployed by the content provider. To ensure deployability, we have limited ourselves to data-path changes—in particular, what can we do to make middleboxes explicit and more flexible?

We find that, surprisingly, any mobility solution can be used to solve our problem. Of the plethora of mobility solutions available we use Multipath TCP, a recently standardized protocol now available on IOS devices.

We show how MPTCP can be used to implement connection acrobatics, offering a degree of flexibility that is badly needed in the Internet. Our implementation and experiments show connection redirection, connection-migration and virtual machine migration are efficient and perform well in practice. These tools can help fix other problems of the Internet today including the difficulty of load balancing traffic and can help protect against DDoS attacks.

Acknowledgements

This work was partly funded by Trilogy 2, a research project funded by the European Union's Seventh Framework Programme FP7/2007-2013, (grant agreement number 317756), and the by IAP Bestcom network.

8. REFERENCES

[1] M. Bagnulo. RFC 6181: Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses, 2011.

[2] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, and C. Raiciu. Analysis of MPTCP residual threats and possible fixes. Internet-Draft draft-bagnulo-mptcp-attacks-00, 2013.

[3] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default. 2005.

[4] P. Eardley. Survey of MPTCP Implementations. Internet-Draft draft-eardley-mptcp-implementations-survey-02, 2013.

[5] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Rfc 6830: The locator/id separation protocol (lisp), January 2013.

[6] Florin Sultan and Kiran Srinivasan and Deepa Iyer and Liviu Iftode. Migratory TCP: Highly Available Internet Services Using Connection Migration. Rutgers University Technical Report DCS-TR-462.

[7] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. RFC 6924: TCP Extensions for Multipath Operation with Multiple Addresses, January 2013.

[8] S. Guha and P. Francis. An end-middle-end approach to connection establishment. In *SIGCOMM*, SIGCOMM '07, pages 193–204, New York, NY, USA, 2007. ACM.

[9] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proc. ACM IMC*, 2011.

[10] E. Katz-Bassett, D. R. Choffnes, Í. Cunha, C. Scott, T. Anderson, and A. Krishnamurthy. Machiavellian routing: improving internet availability with bgp poisoning. In *Hotnets*, page 11. ACM, 2011.

[11] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Rfc 5201: Host identity protocol, April 2008.

[12] E. Nordmark and M. Bagnulo. Rfc 5533: Shim6: Level 3 multihoming shim protocol for ipv6, June 2009.

[13] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: an end-host stack for service-centric networking. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[14] C. Paasch and O. Bonaventure. Securing the MultiPath TCP handshake with external keys. Internet-Draft draft-paasch-mptcp-ssl-00, 2012.

[15] C. Perkins. RFC 2002: IP Mobility Support, October 1996.

[16] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be?... In *NSDI*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[17] S. Schuetz, N. Koutsianas, L. Eggert, W. Eddy, Y. Swami, and K. Le. TCP Response to Lower-Layer Connectivity-Change Indications. Internet-Draft draft-schuetz-tcpm-tcp-rlci-03, 2008.

[18] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Mobicom*, MobiCom '00, pages 155–166, New York, NY, USA, 2000. ACM.

[19] F. Sultan, A. Bohra, and L. Iftode. Service continuations: An operating system mechanism for dynamic migration of internet service sessions. In *SRDS*. IEEE Computer Society, 2003.

[20] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, OSDI'04, pages 15–15, Berkeley, CA, USA, 2004. USENIX Association.

[21] D. Wing and A. Yourtchenko. RFC 6555: Happy Eyeballs: Success with Dual-Stack Hosts, April 2012.