HTTP: An Evolvable Narrow Waist for the Future Internet

Lucian Popa
Patrick Wendell
Ali Ghodsi
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-5
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-5.html

January 4, 2012

Copyright © 2012, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

HTTP: An Evolvable Narrow Waist for the Future Internet

Lucian Popa* Patrick Wendell* Ali Ghodsi* Ion Stoica*

Abstract

While the Internet is designed to accommodate multiple transport and application layer protocols, a large and growing fraction of Internet traffic runs directly over HTTP. Observing that HTTP is poised to become the de-facto "narrow waist" of the modern Internet, this paper asks whether an HTTP narrow waist, compared with the an IP-layer waist, facilitates a more *evolvable* Internet. Evolvability is highly desirable for the Internet, since communication patterns change must faster than the underlying infrastructure. Furthermore, the narrow waist plays in important role in enabling or preventing architectural evolvability. We argue that HTTP is highly evolvable, due to (i) naming flexibility, (ii) indirection support, and (iii) explicit middleboxes. We point to evolving uses of HTTP on today's Internet, and designing our own publisher/subscribe service, HTTP Relay Service (HTTP-RS), on top of HTTP.

1 Introduction

During the past decade, we have witnessed an explosive growth of HTTP traffic [28,37] and a massive increase in HTTP infrastructure in the form of Content Distribution Networks (CDNs), HTTP proxies, caches, and other HTTP middleboxes. A variety of applications, such as video and audio streaming, have migrated to HTTP, abandoning protocols specifically designed for their workloads (*e.g.,* RTSP, RTMP) [1, 7]. This trend is fueled by the ease of deploying new functionality on the data path via reverse and forward proxies, a wide distribution of HTTP client and server software libraries, and HTTP's ability to penetrate firewalls [32].

The growth of HTTP traffic also pushes infrastructure providers to expand their HTTP footprint, creating a positive feedback loop and accelerating HTTP traffic growth. Taking this trend to its logical conclusion, it has been argued [32] that HTTP has become the de facto "narrow waist" of the Internet, as the vast majority of traffic runs over HTTP instead of directly over IP.

A highly desirable property of a narrow waist is the ability to *evolve* with the ever-changing needs of new applications. This ability is important since it is nearly impossible to predict the needs of future applications. If the narrow waist can evolve and provide new functionality, then all applications can take advantage of such functionality. If the narrow waist is not evolvable, the applications have to either implement the functionality themselves, or wait for their protocol of choice to implement it. In fact, one could argue that the main motivation behind the flurry of recent proposals for new network architectures [9, 20, 26, 27, 39, 42] is a response to IP's inability to evolve and support features such as content dissemination, explicit support for middleboxes, and anycast. It should come as no surprise that evolvability has recently been singled out by several clean-slate proposals as the most desirable feature of a future architecture [8, 19].

In this context, we ask the following natural question: *Is HTTP evolvable?* Despite the fact that one could convincingly argue that HTTP is already an "ossified" protocol[1], we answer this question positively by arguing that HTTP is surprisingly evolvable and a "narrow-waist" well poised to serve the needs of the future Internet.

We argue that HTTP's evolvability boils down to some of the same properties that have led to its widespread adoption: (i) flexible names, (ii) support for redirection through name and address decoupling, and (iii) explicit middlebox support. These properties enables HTTP to evolve without protocol changes by enabling innovation *within* its framework. For example, HTTP allows one to encode arbitrary application information into names, use new name resolution mechanisms to implement anycast and redirection, and deploy new functionality at middleboxes.

As proof points, we observe that these properties have already enabled HTTP to provide services and functionalities not envisioned at the time of its design. For example, CDNs use HTTP redirection and middlebox support in the form of reverse proxies, chunked video streaming take advantage of naming flexibility, URL tokenization leverage both redirection and name flexibility, and, finally, anonymization and content-filtering services take advantage of the ability of clients to point the HTTP traffic to specialized middleboxes (proxies).

We explore the limits of HTTP's flexibility by extending the HTTP communication model with a new service that is fundamentally different from its original intended use. In particular, while HTTP implements a *client-server pull*-based communication model, where clients

*University of California, Berkeley

[1]Indeed, the HTTP protocol specifications have been stagnant since the specifications of HTTP 1.1 were published in 1999, and there are no imminent plans for a new version.

pull content from servers, we aim to implement a generic *client-to-client push*-based communication model. This model is highly general as it provides a variety of communication primitives, such as unicast, anycast, and multicast. These primitives would enable HTTP to support a class of applications that it does not support well today, *i.e.,* low-latency applications, including voice, video conferencing, and chatting. We believe that adding such a generic client-to-client communication model to HTTP represents not only a compelling proof point for its evolvability, but also allows HTTP to handle virtually any application that today uses TCP/IP, extending the benefits of HTTP to an entirely new set of applications.

To realize this vision we design, deploy, and evaluate *HTTP Relay Service* (HTTP-RS). The architecture of HTTP-RS is similar to that of the Internet Indirection Infrastructure (*i3*) [39], as it enables client-to-client communication through an indirection point. Like *i3*, this model enables a variety of communication models. With HTTP-RS, a client can subscribe to an URI using *S-GET*, an annotated GET request. Once it subscribes, the client starts receiving data subsequently published to that URI. The communication takes place through an HTTP infrastructure that relays data between clients. In response to demand, a number of ad-hoc solutions have already been proposed to implement push based communication from a single server to a single client [6,14], HTTP-RS standardizes these efforts and provides a more general client-to-client communication model.

Our HTTP-RS implementation requires no protocol changes, and leverages two key HTTP properties: (a) flexible naming to name the communication channels and the data units exchanged by the clients, and (b) HTTP redirection to perform relay selection. We evaluate the performance of HTTP-RS and find that it incurs only minor overhead in terms of latency and throughput. Furthermore, stateful relays can be implemented in a scalable manner. Adding a fundamentally different communication model to HTTP without protocol changes represents a strong proof point of HTTP's ability to evolve.

In summary, in this paper, we make two main points:

1. HTTP is a highly evolvable protocol. HTTP's evolvability, together with its widespread adoption, opens ample opportunities to deploy new functionality in the Internet. In fact, we believe that some of the functionalities proposed in the context of recent clean-slate designs can be implemented at the HTTP layer. We give one such example, HTTP-RS, which implements the indirection functionality proposed in [39].

2. HTTP's evolvability can be traced to three properties: flexible naming, support for redirection, and explicit middlebox support. We believe these properties can represent a source of inspiration for both researchers and practitioners designing new protocols, and/or new network architectures.

This paper is organized as follows. The next section (§2) describes the growth of HTTP traffic and motivates a discussion of evolvability. Sections (§3) identifies the fundamental properties of HTTP which promote evolvability and (§4) presents examples. We introduce HTTP-RS (§5) and describe its use (§6) and deployment (§7). The next section (§8) evaluates the performance of HTTP-RS. The final sections of the paper cover related work (§9) and conclusion (§10).

2 Background

In this section, we summarize the main arguments for HTTP becoming the de facto a narrow-waist of the Internet [32]: (a) it's explosive growths, and (b) its ability to provide features, not supported by the existing IP, and targeted by new Internet architecture proposals.[2] We then discuss the evolvability problem in the context of HTTP.

2.1 HTTP's Growth

With the advent of the web in mid-90's, HTTP became a significant fraction of Internet's traffic [31]. In the intervening years, a handful of other application-layer protocols have seen periods of popularity, including proprietary streaming protocols (*e.g.,* RTMP, RTSP) and peer-to-peer technologies. With time, however, these applications have mostly migrated to or been replaced by services operating over HTTP. HTTP traffic today, for instance, is driven by the growing popularity of video traffic (Cisco forecasts that by 2013, 90% of the consumer traffic will be video [4, 36]). To sustain such growth, content providers and aggregators have recently turned their attention to *HTTP chunking*, a technology pioneered by Move Networks [1]. HTTP chunking allows content providers to leverage their vast HTTP infrastructure, and helps clients improve their performance due to its ability to leverage corporate and ISP proxies, and traverse firewalls. HTTP chunking is today employed by all major video distribution platforms, including Microsoft, Adobe, and Apple, and it is rapidly growing to dominate the Internet video traffic.

HTTP traffic is also increasing at the expense of peer-to-peer (P2P) traffic [4, 30, 36]. One of the main reasons for this has been the dramatical decrease in cost of CDN delivery (*e.g.,* by a factor of 10 between 2006 and 2010), which had considerably decreased the appeal of P2P distribution. Today, virtually all major content providers use CDNs instead of P2P for content delivery. With the advent of HTTP chunking and with

[2] For an in-depth discussion see [32].

a continuous expansion of the HTTP infrastructure, we expect that this trend will only intensify.

Given these trends and the massive investment in HTTP infrastructure on today's Internet, we believe that HTTP traffic will dominate (at least in volume) Internet traffic for the foreseeable future.

2.2 Evolvability

A highly desirable characteristic of any network architecture, Internet included, is the ability to *evolve* and adapt to future applications with unexpected needs. Indeed, many of the recent clean-slate proposals of network architectures emphasize evolvability as the centerpiece of their design [8, 19]. Given its position as the common denominator in the protocol stack, the narrow waist is the key to architecture evolvability. If the narrow-waist doesn't enable evolvability, the developers have no choice but try to implement new functionality at a different layer. If the new functionality starts being used by the majority of applications, the layer providing that functionality may become the new (de facto) narrow waist. Arguably this is the path followed by HTTP. As IP could not evolve to provide support for content distribution, explicit selection of middleboxes, and improved security, HTTP filled these gaps!

There at least two dimensions to evolvability. The first dimension is the ability to easily transition from one version of the protocol to another, *e.g.,* from IPv4 to IPv6. The second aspect is the ability to implement and deploy new functionality without protocol changes. By new functionality we mean functionality that protocol designers did not consider at the onset. In this paper, we focus on the second dimension, as it is arguably more general and easier to evaluate. We emphasize that evolvability does not imply protocol changes. For instance, at one extreme, an active network is highly evolvable without changing its specification. Similarly, in this paper, we show that HTTP is highly evolvable despite the fact that arguably HTTP is already an "ossified" protocol. In particular, we argue that HTTP's evolvability stems from three basic properties that we discuss in Section 3.

3 An Evolvable Architecture

In this section, we identify three key properties that we argue are responsible for HTTP's evolvability.

3.1 Properties

Flexibility in naming: HTTP uses flexible naming to identify resources. An HTTP resource is identified via a URN; in practice, the most common use of URNs are HTTP URL's. HTTP leaves open how servers should interpret a URN once a request is received. This lack of semantic strictness allows flexible and evolving naming schemes. Indeed, HTTP naming has changed dramatically in the last ten years, from URL's commonly identifying file names to a more content-centric approach (*e.g.,* http://cnn.com/politics).

Similarly, HTTP does not specify the mechanism by which names are resolved into destination addresses. This is an important decision, because it leaves the door open for future resolution mechanisms, should the need for them arise. In principle, it would be possible to introduce an entirely different resolution step, along with its own supporting architecture, and stay within the bounds of the HTTP specification. While this would be a monumental task, it would not necessitate altering HTTP's content retrieval infrastructure. Since name resolution and content retrieval are decoupled in HTTP's design, one can evolve without affecting the other.

Indirection primitives: HTTP can leverage DNS to provide indirection. This allows applications such as CDNs, to use dynamic DNS updates to offer anycast functionality for content delivery. Indirection is not tied to DNS however, as HTTP offers a robust, explicit redirection primitives. HTTP indirection is generally more powerful, because redirectors base decisions on a full URN (not just a hostname) and see true client IP address. The latter is important since network topology may be a consideration in such schemes [38].

Prior work argues that indirection, specifically anycast, is the single most important enabler of network evolvability [33]. While earlier projects considered such redirection at the network layer, we observe that application-layer redirection provides many of the same benefits. In particular, HTTP redirection allows late-binding per flow connection decisions. Several recent projects have proposed global control planes (4D, SDN, Ethane) and these all require such functionality. Redirection also enables incremental deployability, since client requests can be routed to a subset of servers with the corresponding functionality. Finally, redirection supports inter-service delegation and allows dispatching of clients from one HTTP service to another.

Explicit support for middleboxes: The HTTP protocol provides explicit support for named middleboxes. This facilitates end-middle-end applications, where third parties explicitly participate in client-server exchange. Since proxies are a first-class entity in HTTP, they need not be deployed on the datapath for the clients and servers to take advantage of their functionality. Since they are explicitly named, a client can choose whether or not to engage a particular middlebox (forward proxy) for each request. Similarly, servers can use DNS or URL redirection to make sure that the requests addressed to it are processed by a particular middlebox (reverse proxy). Furthermore, upon receiving a request, a server knows whether that request has arrived via a particular middle-

box, and can generate responses accordingly.

Explicit middlebox support has been identified by several previous works as an essential property to provide flexible deployment of new functionality [20, 39, 42].

3.2 Discussion

As argued in [32], HTTP has three properties which feature prominently in clean slate proposals. First, HTTP is a content-centric protocol, where several architectures have advocated for content identifiers in packet headers [12,26,27]. Second, HTTP offers explicit middlebox support, a feature argued for in numerous designs [9,20,27,39,42]. Third, via DNS naming, HTTP allows limited support for anycast communication and single host mobility (as proposed in [15, 24, 35, 39, 42, 44]) Not surprisingly, there is a significant overlap between these three properties and the evolvability properties discussed in this section. However, there are two important differences which we emphasize next.

First, we single out the name flexibility as one of the key evolvability properties. Virtually all evolvability examples we discuss in Section 4, as well as HTTP-RS, leverage the name flexibility property. In contrast, we do not list the content-centric nature of HTTP among the evolvability properties, as this property follows at a large extent from the name flexibility property, *i.e.,* the ability to name the content.

Second, while decoupling the host identifiers from addresses is necessary to provide indirection, this decoupling captures only part of the power of the indirection primitives. The other critical aspect is that HTTP does *not* mandate the resolution mechanism, *i.e.,* how names are mapped to addresses. This is a very important decision, as it allows one to provide arbitrarily anycast services by controlling the resolution mechanism. For comparison, some of the previous proposals that decouple names/identifiers from addresses dictate the resolution mechanism, *e.g.,* i3 uses a DHT mapping [39].

Finally, one legitimate question is whether the three properties we have listed in this section are either necessary or sufficient for achieving evolvability. Unfortunately, it is hard to give a definite answer to this question given that the evolvability itself is not well defined. Short of giving such an answer, we argue that these properties are highly desirable, and the lack of any of them would hamper the evolvability. As we have already mentioned, the ability to provide indirection has been singled out as the key property to provide evolvability by [33], and [39] provides a convincing proof-point for this property. Together with indirection, the ability to explicitly name middleboxes allows the developers to incrementally deploy arbitrary functionality at proxies. In the absence of this property, it would be much more difficult for clients to direct their requests to proxies that implement the desired functionality. Finally, name flexibility allows clients to encode application specific information to be interpreted by proxies and servers. As demonstrated by various HTTP services, such as CDNs and our own HTTP-RS, name flexibility is critical to implement flexible anycast functionality.

4 Evidence For Evolvability

In the decade since HTTP 1.1 was officially codified, the nature and quantity of Internet traffic has changed dramatically. The Internet has welcomed a bevy of highly interactive applications (*e.g.,* online document editing) which deviate from the model of static content retrieval. Static content, in turn, has become orders-of-magnitude greater in size, driven by the advent of high-definition and often realtime video. Content is increasingly user-customized, requiring unprecedented interoperability between websites to preserve user identity and capabilities. One might expect these dramatic workload changes to drive developers *away from* HTTP. Instead, the opposite has happened. New applications have leveraged HTTP's flexibility to build such functionality. This section explores several representative examples of applications which use HTTP in unanticipated ways. Throughout, we refer back to the principles outlined in section 3.

Content distribution networks (CDNs): The last decade has seen an an increase in Internet bandwidth and availability. In turn, aggregate demand for online content has increased several fold. The bulk of this burden has fallen to CDNs, who must maintain robust and highly available infrastructure for serving HTTP content to millions of simultaneous users. In this role, CDNs have become a necessary backbone, serving a large proportion of all Internet traffic. The HTTP specification does not mention or explicitly support CDNs; the industry was still in its nascency when the protocol was standardized.

CDNs leverage two of the previously mentioned properties of HTTP to construct robust third party content-serving infrastructure. First, CDNs make extensive use of HTTP's support for explicit middleboxes, relying heavily on both forward and reverse caching proxies to serve data with minimal end-user latency. Second, CDNs leverage flexible name resolution to provide an "anycast" abstraction to HTTP clients, thus minimizing deployment complexity. Redirection comes both via extensive DNS dynamism (matching hostnames to edge-caches) [17, 38] and HTTP-level redirection between caches [43]. Unlike IP-layer anycast, however such indirection lets CDNs consider server load, cache locality, access control, and other application-specific criteria in directing clients to replicas.

URL tokenization:. This technique leverages HTTP's flexible naming and indirection primitives to enables the

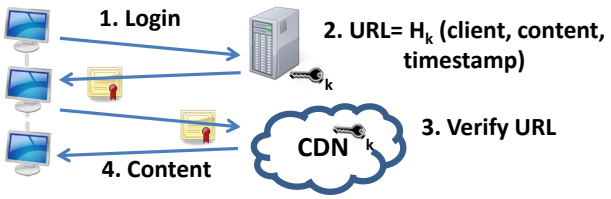


Figure 1: URL Tokenization. A web client is authenticated by an application server, then retrieves content via a certified URL. The server and CDN share a secret key, k.

transfer of access control credentials between websites. For example, assume a media company, `foo.com`, hosts content at `cdn.com`. `foo.com` authenticates its own users but wants that only authenticated users to access the content hosted by `cdn.com`. One solution to this problem is as follows: `foo.com` and `cdn.com` share a single secret key, k, which is disseminated out-of-band. When an authenticated `foo.com` user X wants to watch a video, `foo.com` embeds and signs $[X$,time, movie$]_k$ in the HTTP URL. The client is then redirected (via explicit redirection or content embedding) to that URL, which the CDN can verify. This method, which is detailed in Figure 4, has developed into a de-facto standard offered by multiple CDN's.

Embedding privileges in the URL has several advantages. First, it eliminates the need for realtime coordination between applications and content distributers. Since CDN's operate in a highly distributed fashion with hundreds or thousands of cache locations, this coordination can pose high overhead (or limit performance, if the cache pool is decreased). Tokenization requires only a single out-of-band agreement on keys, which can then be propagated amongst caches. Second, this strategy limits the possibility that a third-party middlebox or cache will hold onto a piece of privileged content and serve to unauthorized users, since URL names are unique to the combination of content and access. Naming obfuscation therefore provides a defense against unauthorized dissemination of content. Variants of the strategy have also been employed other contexts where it is necessary to encode temporary privileges in URL names, such as password recovery links.

HTTP chunking: As mentioned in Section 2, several companies, including Move Networks [1] and Swarmcast, [2], have pioneered HTTP chunking, which enables the delivery of video and audio over HTTP instead of traditional streaming protocols. This technique chunks a video stream into blocks of a few seconds each and then distribute these blocks as individual files by leveraging existing CDNs and HTTP proxies. Taking advantage of HTTP's flexible naming, file names convey both video identifiers and chunk offsets. A client downloads chunks, stitches them together, and plays the original stream.

HTTP chunking has several advantages over traditional

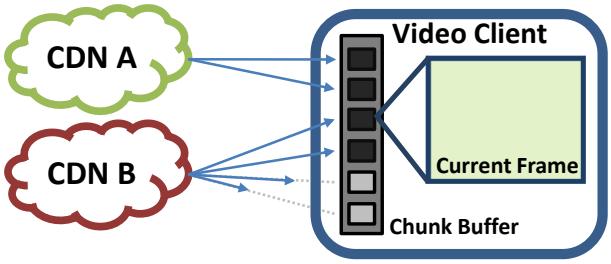


Figure 2: HTTP Video Chunking. An HTTP client retrieves and stitches video splices from multiple CDNs.

streaming protocols. First, it increases the distribution scale and reduces the cost, as CDNs have more HTTP servers than streaming servers, and they do not incur licensing costs for the HTTP servers (these servers are typically based on open-source software, unlike the streaming servers). Furthermore, using HTTP to distribute video can leverage the HTTP caching proxies deployed by ISPs and enterprises. Second, it improves availability: if an HTTP server fails, the client can mask such a failure by requesting the subsequent chunks from a different server or CDN. Third, it improves quality, as a client can request multiple chunks simultaneously, which leads to aggregating the throughput of multiple TCP connections. In contrast, traditional streaming protocols use one TCP connection for data transfer.

Middleboxes: Because HTTP middleboxes need not be on the data path, they are easily deployed by third parties. While this functionality was designed with simple proxying and caching in mind, it has been extended to include a litany of intelligent content-transforming middleboxes. These include anonymizing HTTP proxies, content filters, intrusion detection boxes, web accelerators, layer-7 load balancers, and transcoders for mobile devices. Because HTTP explicitly names these middleboxes, their use is not at odds with the correct functioning of the protocol. Indeed, these services are often deployed only for certain sets of HTTP clients, such as mobile users or those seeking privacy, and specific clients can use them intermittently request-to-request. In contrast, IP-layer middleboxes represent an "all or nothing" proposal, touching all traffic on the data path, often unbeknownst to the sender or receiver.

In each of these examples, the properties of HTTP discussed in section 3 are leveraged to enable unanticipated functionality. These examples, while only a small sample of HTTP's modern use, suggest that the migration towards HTTP traffic has not prevented a dynamic and evolving Internet.

5 HTTP RELAY SERVICE

HTTP has evolved to meet the changing demands of Internet applications *to date*, and this presents strong evidence that an HTTP infrastructure is, in fact, evolv-

able. We next ask whether HTTP will evolve gracefully in response to *future* application demands. This question is difficult to answer, as we cannot definitively characterize the workloads of future application. One strategy would be to anticipate which application will dominate tomorrow's traffic, and then assess HTTP in the context of that application. We take a more general approach, and develop a new communication service between two or more HTTP clients that supports a wide variety of communication patterns. Our service, HTTP-RS, offers a publisher/subscriber primitive on top of which a diverse range of applications can be deployed, including those for which HTTP has historically been a bad fit, such as low-latency or client-to-client services. Since HTTP-RS differs drastically from HTTP's current communication model, it presents an extreme experiment in evolvability.

5.1 HTTP-RS Communication Model

HTTP's architecture is based on a client-server communication abstraction. However, many of today's services—such as chat or VoIP—are in essence client-to-client. There is a clear trend toward these services moving to HTTP, with Facebook chat as a popular example. In our model, two or more HTTP clients send and receive updates through intermediary HTTP servers, *e.g.,* one client sends a chat message that is relayed by the server to another client. We use the term "client" in a general sense, in that clients may send or receive data. A server or relay is a host which relays traffic between clients.

To support client-to-client communication, we need to address two distinct problems. First, the server needs to provide a relay service between clients. This is supported within today's HTTP semantics: a client can publish data to an HTTP server (via PUT or POST commands), and another client can receive data from that server via GET.

Second, to support real-time applications such as voice and video conferencing, there should be a *low-latency* mechanism that allows a client to get the data as soon as it is published to the server. Unfortunately, using GET to periodically poll the server for new data may be insufficient. Assuming the receiver checks for content every T ms, the end-to-end latency may exceed T ms. If an application wants to achieve an end-to-end latency on par with cross country latencies, it needs to poll about every 50 ms, which can be prohibitive both for the client and the server.

The need for providing a low-latency fetching mechanism in HTTP has long been recognized by industry, and several approaches commonly referred to as "HTTP push" or "Comet" [6, 14] have been proposed. These approaches typically leverage Javascript to send asynchronous HTTP requests, which either delay the response to a GET until data is available, or send a stream of never-ending data. However, these solutions are focused on server-to-client communication; they provide no standardized mechanism to relay the packets from one client to another.

In this paper, we build on these solutions to provide a *client-to-client low-latency* service. The key abstraction to implement this service is Subscribe-GET (S-GET), which allows a client to subscribe to a URI. Unlike existing "HTTP push" approaches, S-GET provides end-to-end semantics: any data that is published to a URI is forwarded by the HTTP server to all clients subscribing to that URI via S-GET. The HTTP servers do not store the data published to an S-GET URI, they just forward the data. This differentiates our solution from client-to-client communication relying on traditional PUT/POST and GET commands. It also removes one of the main hurdles in providing a client-to-client service today, *i.e.,* the reluctance of HTTP servers to allow storing data to avoid the risk of storage DoS attacks. We refer to this as the *HTTP Relay Service* (HTTP-RS).

We note that HTTP-RS is similar to *i3* (*e.g.,* an S-GET request is similar to a trigger), which was a DHT-based architecture that could provide mobility, multicast, and anycast [39]. While *i3* never enjoyed wide adoption, we believe HTTP-RS to have greater adoption potential due to today's massively deployed HTTP infrastructure and associated business incentives.

5.2 Subscribe-GET (S-GET)

The format of an S-GET request is similar to that of a traditional GET. However, unlike GET requests, HTTP servers *store* S-GET requests up to an expiration timeout associated with the request. As long as the S-GET is stored at the server, any updates (through PUTs) to the URI of the S-GET are sent to the client that issued the S-GET. A server removes an S-GET request after the timeout expires or when a client closes the underlying TCP connection. Each update is sent to the client through a regular HTTP response.

S-GET only returns content published after the S-GET has been received by the server. Thus, S-GET provides a publish/subscribe abstraction, where a receiving client subscribes to a URI through S-GET, and a sending client publishes application data units (ADU) to that URI using either POST or PUT. Note that there can be multiple senders as well as receivers for the same URI.

The S-GET request contains the desired timeout as a header attribute. For security purposes, the server may not accept large timeout values, in which case, it returns an error response containing the maximum allowed timeout. To extend their duration, S-GET requests need to be "refreshed" before their expiration. If the server accepts the timeout of the refresh S-GET, it updates the timeout of the stored S-GET. If it cannot accept the timeout of a refresh S-GET, it sends an error response

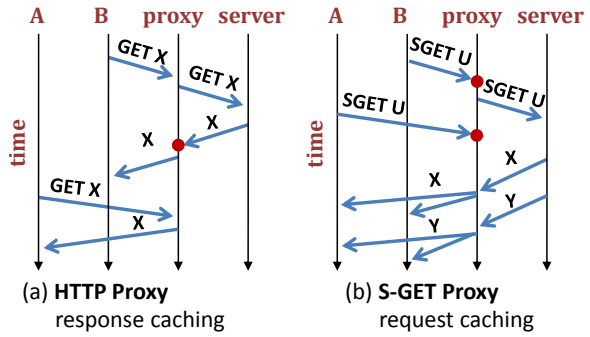


Figure 3: Dual proxy modes. (a) GET responses are cached for future requests. (b) S-GET requests are cached for future pushes. Circles represent cache events.

but keeps the stored S-GET unmodified.

Low-latency communication through HTTP-RS is naturally implemented using S-GET. Consider communication between two clients, `A` and `B`. To start communicating, client `B` sends an S-GET request to a URI, and subsequently client `A` publishes ADUs to that same URI though HTTP PUT requests. All ADUs can share the same URI, because S-GET provides a named-pipe abstraction. For example, the URI can be `S.com/from/A/to/B`, where `S.com` is the DNS name of server `S`. Since publishing a new ADU through PUT is equivalent to modifying the content at `S.com/from/A/to/B`, the server will forward each ADU to `B`. For clarity, throughout this section, we structure URIs similarly to above, but in practice `A` and `B` can use arbitrary URIs. Also, for brevity, we typically use `S` for the server's DNS name (instead of `S.com`).

The efficiency of delivering HTTP-RS content can be increased with the use of caching proxies. Whereas, today's proxies cache GET *responses* and deliver them to subsequent GETs for that URL, S-GET, proxies cache S-GET *requests* rather than responses, *i.e.,* the dual of what is done today (see Figure 5.2). If the proxy receives multiple S-GETs for the same URI, it need only forward the first to the server. The remaining S-GETs can be served using copies of the reply; this behavior is similar to that of an IP multicast router, creating a (reverse path) distribution tree. When a proxy receives a new S-GET request with a timeout expiring later than its current subscription, it will send a refresh request to the server with the newly requested timeout.

Today's HTTP is deployed on top of TCP. In our current implementation, an S-GET request maintains an open TCP connection to the server, thus allowing the server to send data even when the subscribing client is behind firewalls or NATs. While S-GET partially departs from the stateless model of HTTP, it represents a design pattern relied on heavily by nearly all of today's interactive web applications, including GMail and Facebook Chat. These applications scale gracefully to large numbers of simultaneous users. Finally, we note that S-GET uses soft state, which leaves the HTTP's failure semantics largely unchanged.

5.3 Buffering Semantics

Buffer management is an important consideration in HTTP-RS, since senders and S-GET receivers for the same URI may operate at different rates. HTTP-RS allows two approaches to buffering: (1) reliable relay, and (2) real-time relay. Reliable relay is useful when there is a single receiver, while the real-time relay is useful for live audio/video streaming to multiple receivers. In both cases, the server maintains a queue for each S-GET request and enqueues ADUs to be sent to the respective request. Having one queue per S-GET request serves to isolate the performance of different receivers.

For reliable relay, a server always enqueues the ADU in the queues of all the S-GET receivers registered for the URI of the PUT request. The server does not read new resources from the sender until it has enqueued the current request to all receivers. Therefore, in case of multiple receivers, the sender is transmitting at the rate of the slowest receiver. Note that the reliable relay is not guaranteed to be reliable in the strict sense, since an HTTP server may fail. End-to-end reliability still needs to be implemented by applications, though failover is assisted by a rendezvous layer discussed in Section 7.1.

For real-time relay, the server only enqueues the ADU of a PUT to receivers with room in their queue. A receiver with a full queue will not get ADUs sent on that URI. Thus, for real-time relay, only the receivers that are able to keep up with the sender's rate receive all ADUs. In addition, the server reports back in the HTTP response header how many requesters the URI has, as well as how many of them it enqueued the ADU to. The sender may then adjust its rate to ensure that enough ADUs are being enqueued.[3] This model is useful in the case of delay sensitive traffic, such as live video or VoIP, where it is preferred to drop ADUs instead of incurring large latencies. A slow requester may then choose to switch its S-GET to a slower stream, as is currently done in commercial HTTP video solutions.

The two buffering options HTTP-RS are distinguished through a header field alongside PUT requests. If the field is unspecified, real-time relay is assumed.

6 Applications on top of HTTP-RS

Our primary intention in designing HTTP-RS is to enable new communication patterns over HTTP. Therefore,

[3]Note that this is similar to ICMP source quench messages, but with information about multiple receivers.

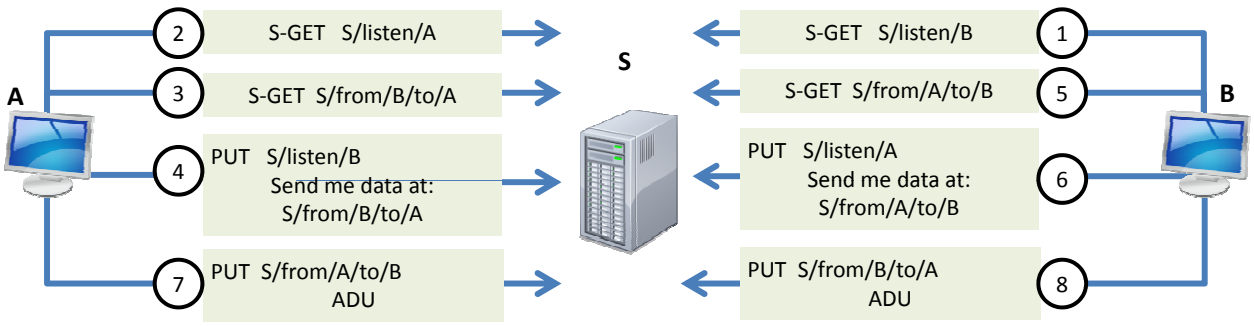


Figure 4: Connection through HTTP-RS with S-GET calls.

HTTP-RS's success is, in some sense, tied to the number of applications contexts which it supports and how much those applications deviate from HTTP's functionality today. In this section, we present increasingly complex examples of how connection oriented communication can be built on top of HTTP-RS.

6.1 Connection Oriented Primitives

Connection oriented communication primitives such as listen, communication management (*e.g.,* open, close)m and reliability can be implemented on top of HTTP-RS. Figure 5.3 illustrate this process for two clients `A` and `B`.

Listen: A client `B` can emulate the listen function by registering a *listen channel* through an S-GET request with a public URI, *e.g.,* `S/listen/B` (Figure 5.3 steps 1 and 2). Every other client can contact `B` through its listen channel, `S/listen/B`, to open a connection.

Connection Management: To receive ADUs, `A` and `B` need to create communication *endpoints*. For example, `A` can send an S-GET request for the URI `S/from/B/to/A` (Figure 5.3 step 3) to open its communication endpoint. Then, `A` sends this URI to `B` by sending a PUT request to `B`'s listen channel at URI `S/listen/B` (Figure 5.3 step 4). Upon receiving this URI, `B` will make an S-GET request to `S/from/A/to/B` (Figure 5.3 step 5) to open its communication endpoint. This URI is sent to `A` via `A`'s listen channel, `S/listen/A` (Figure 5.3 step 6). Once `A` receives this URI, `A` and `B` can start sending data to each other through the endpoints (Figure 5.3 steps 7–8).

There are two points worth noting. First, `A` and `B` can register the two URIs associated to the connection at different servers, *e.g.,* `S1/from/B/to/A`, and `S2/from/A/to/B`, respectively. Second, both `A` and `B` can piggyback their first data units on the URI exchanges.

Timeouts and handshaking can be used for failure-detection and connection-termination. For brevity, we omit the details.

6.2 Advanced Communication Patterns

Multicast and Large Scale Data Distribution: With S-GET, one can easily implement a multicast channel with multiple senders, by having each receiver in the group register an S-GET request for the same URI, which plays the role of the multicast address or identifier. When a sender PUTs an ADU to this URI, the server will forward the response to any client which previously registered via S-GET. By default, this implements an open group multicast model, similar to IP multicast.

HTTP has already been proven highly successful for streaming data to large audiences. The proposed S-GET method can further improve this service for *live* streaming by reducing the end-to-end latency. S-GET can be used to receive the chunks with lower latency, or can be used as a control channel for signaling the availability

Mobility: Since ADUs are relayed through indirection points, both end-hosts can change their address and still be able to continue their current HTTP-level communication. Thus, HTTP-RS supports simultaneous mobility.

Multi-homing: Multi-homed clients could setup different S-GET receive channels for each network interface, in this way using multiple links for the same communication. Thus, if a link goes down, rendering one of its addresses unreachable, a multi-homed host can be contacted through its other receiving channel. To further increase reliability and ensure multiple paths, the sender can choose different servers for its different receiving channels. This approach can also be used to increase throughput and reduce latency.

Multiple paths: Even clients with a single network interface can use multiple paths by registering multiple S-GET requests at different servers. Such an approach could be used to increase reliability or throughput by leveraging uncorrelated failures and disparate performance of multiple links. Implementations with highly distributed relays would increase the likelihood of clients finding distinct paths. Support is limited, of course, if a bottleneck link exists at the edge of the network.

NAT/Firewall Penetration and a Default-Off Architecture: All HTTP requests are client-initiated and hence automatically traverse NATs and firewalls. The server

used in HTTP-RS is a mediator that enables two hosts behind firewalls to communicate, making techniques such as hole punching unnecessary. More generally, HTTP-RS defines an architecture with two types of entities, "clients", which are off-by-default (behind NATs/firewalls) and "servers", which can be contacted by anyone. This is similar to the architecture envisioned in [21], with the addition that in our proposal, clients can still communicate among themselves and can use access control mechanisms to only receive packets from approved senders (see §7.2 for restricting access to sending/receiving ADUs). In this way, DDoS attacks can be alleviated since resource-weak hosts can be off by default and receive data through HTTP-RS channels opened at multiple resourceful servers and data centers.

7 Deploying HTTP-RS

HTTP-RS offers a simple, but powerful abstraction to application developers. The benefits of HTTP-RS are only realized, however, if the service can be effectively deployed at the HTTP layer. At a minimum, we can identify two requirements for a deployment of HTTP-RS: First, HTTP-RS requires an indirection layer to intelligently assign client requests to relays. Second, as HTTP-RS enables new modes of communication between clients, it must present corresponding security features. In this section, we discuss how HTTP-RS leverages the underlying flexibility of HTTP to provide these features.

The design of HTTP-RS does not constrain which party operates relay servers. In today's Internet, CDNs seem to offer the most viable candidate given their existing role as providers of third-party infrastructure. We therefore consider deployment in the context of CDN infrastructure, though we emphasize this is only one possible deployment scenario.

7.1 Relay Selection and Migration

HTTP-RS leverages HTTP's redirection support to perform relay selection and migration. Server selection is a well studied problem [18] and is facilitated today in CDNs using massive indirection frameworks. Factors such as network proximity, server load, and access control influence which server will handle a request for a given client. Solutions to the server selection problem can be loosely categorized along two axes: First, they vary in which protocol is used, relying on either DNS-based or HTTP-based indirection. DNS-based server selection is more transparent but suffers from inaccuracy, whereas HTTP-based server selection is more accurate but incurs additional latency. Second, solutions opt for different policies governing how to dispatch clients to servers. Relay selection in HTTP-RS introduces additional constraints, as a CDN must minimizes latency (and maximizes throughput) between *multiple* clients. In addition, the relay assigned to a URI must stay constant on short time horizons, since the relays are stateful.

Our current implementation of HTTP-RS performs HTTP-layer server selection via a stateful rendezvous layer. The rendezvous layer tracks the current set of live relays and stores as soft state the mapping of URIs to relays. Figure 7.1 depicts an HTTP client performing an S-GET for `foo.com/ch1`. The domain name, `foo.com`, resolves to one of many caching rendezvous servers, RV, via DNS delegation from its owner to the CDN. The rendezvous servers keeps track of (channel→relay) mappings via a shared datastore. In this example, RV has a cached entry mapping from `foo.com/ch1` to relay $R3$, since the channel is already in use by two other clients (far right), RV responds with a `302 Redirect` explicitly forwarding the request to $S3$.

If, alternatively, RV did not have an existing entry for this channel, it would forward to the nearest relay and add a corresponding entry to the shared state store. Prior work has observed that choosing a relay close to one party provides a good approximation for the optimal relay [39], and that policy serves as the default in our current implementation.

HTTP-layer indirection offers us many advantages over a more transparent DNS approach. For instance, a client can actively request a particular relay via an HTTP header in the S-GET request, and the rendezvous will direct to that relay if the URI is otherwise unassigned. This could be because the client has a better idea than the rendezvous service which relay will offer it best performance or that the client wants explicit control over paths (such as with multipath HTTP) but still wants to store state in the rendezvous layer for future subscribers. In the limit case, a client can completely ignore the rendezvous layer and perform relay selection using some out-of-band mechanism. This is possible since relay hosts are explicitly named and promotes flexibility for applications which prefer their own relay selection models.

HTTP-RS also allows for client-initiated migration of relay assignments. A special HTTP header present in either an S-GETor POSTconveys intention to migrate a URI to another relay. Granting this request requires the relay coordinate with rendezvous layer (not pictured in Figure 7.1) to update the URI mapping. If the migration request is accepted by the relay, it redirects all participants to the new relay location.

Relay failures are recognized by the rendezvous layer via heartbeat timeouts. After failure, clients must re-establish their connection with the rendezvous service in order to find the new relay. This preserves the failure semantics of current HTTP clients.

We further find that, for the long-lived channel connections of HTTP-RS, the startup cost of an extra TCP connection with the rendezvous is outweighed by the

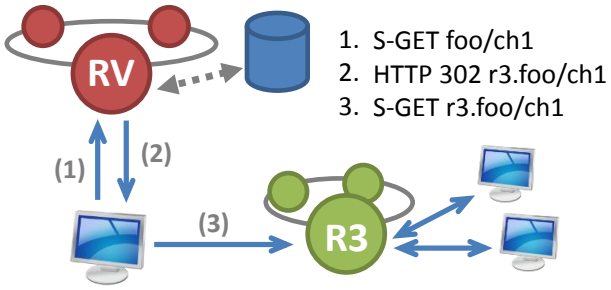


Figure 5: A client performs an S-GET for foo/ch1 and is redirected to the correct relay (R3) via a rendezvous node (RV). Two other clients are already assigned to R3.

benefit of finding a good relay server. This tradeoff is explored in more detail in Section 8.

7.2 Security of HTTP-RS

Broadly speaking, HTTP-RS faces three types of attacks: impersonation, eavesdropping and DoS attacks. First, a malicious node could attempt to impersonate the sender by putting ADUs at the URI on which the receiver gets the ADUs. Second, an attacker could eavesdrop on a communication by simply issuing an S-GET request to the receiver's endpoint URI. With a similar attack mounted on the receiver's listen channel, the attacker can impersonate the receiver by responding with its own receiving endpoint to connection requests. Finally, a denial-of-service (DoS) attack could be launched against the HTTP-RS service by registering many S-GET requests to use up memory and bandwidth resources at servers. We discuss solutions for all these challenges which can be implemented in today's HTTP.

7.2.1 Impersonation and Eavesdropping

We consider two contexts: (1) unencrypted traffic and (2) encrypted traffic. When traffic is not encrypted, hosts can protect against impersonation and eavesdropping attacks by using URIs that contain sufficiently long, hard-to-guess (*htg*), random strings. Initial exchange of these *htg* URIs must occur over a secure channel. For this purpose, the endpoints can use encrypted HTTP-RS channels (described below) or any other secure channel.

Encryption prevents impersonation and eavesdropping attacks if the endpoints are authenticated. Encryption can take place at the application level or at the transport layer to/from the indirection server, depending on whether the application trusts the relay service.

For transport layer encryption, we introduce the notion of a *self-certified URI*, in short *sc URI*. A *sc URI* is one that contains a public key, *e.g.,* `S.com/pk/P` where `P` is the public key; in practice, a hash of the key could be used for efficiency. *sc URIs* can be used to authenticate the receiver or the sender in an HTTP-RS communication; this is similar to YURLs [3] today.

Application level encryption can be set up using digital certificates or public (or secret) keys. Public keys can be learned directly from *sc URIs* or through out of band mechanisms. After knowing the public key of the other endpoint, traditional encryption and key establishment techniques can be used on top of HTTP-RS (*e.g.,* similar to those used by TLS but at the application level).

Note that encrypted ADUs can be used on the control plane and *htg URIs* on the data plane. For example, host `A` can listen on a *sc URI*, and host `B` can send it an *htg URI* on which `B` is receiving messages, encrypted with the public key used by `A` in the *sc URI*.

7.2.2 DoS attacks

We identify two types of DoS attacks in the context of HTTP-RS: (1) DoS attacks directly against clients and (2) DoS attacks against one or more HTTP-RS relays.

DoS clients: HTTP-RS allows a client-to-client communication even when hosts are behind NATs and firewalls. This enables a default-off architecture [10, 22] and limits the effectiveness of any DoS attempt against a client gateways (they will simply drop unrecognized flows).

DoS the HTTP-RS service: Attackers might also try to DoS HTTP-RS servers (on bandwidth, memory, CPU) to disrupt the HTTP-RS service. We contend that DoS attacks on an HTTP-RS server do not fundamentally differ from attacks on today's servers. While HTTP-RS servers sore more state than a basic HTTP proxy (multiple open TCP connections and small per-connection buffers), they are on-par with RTMP servers and existing HTTP servers supporting long-polled connections. HTTP-RS servers are therefore amenable to current techniques for detecting and suppressing DoS behavior which are widely used in CDNs. Although HTTP-RS servers fare no better or worse than the status quo, the broader HTTP-RS architecture does migrate the DoS threat from the clients to relay providers. Historically, infrastructure providers such as CDNs have proved best equipped to detect and mitigate such threats, so this division of labor arguably provides maximal security.

8 Evaluation

Applications using HTTP are often performance conscious. As a result, HTTP servers are heavily optimized to handle many concurrent connections [5], while content caches are pervasively deployed inside CDNs and large scale ISPs [11, 40] to mitigate request load. Since HTTP-RS data is not persistent, it is not amenable to caching. Therefore, HTTP-RS relays must themselves offer good performance, at very least comparable to an equivalent service implemented over raw sockets. We now evaluate the proposed HTTP-RS service, and the overhead HTTP-RS clients experience in terms of

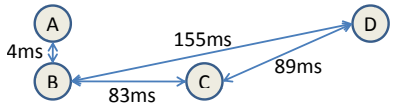


Figure 6: Topology of test setup.

latency, throughput, scalability, data overhead, jitter. We also evaluate the performance of one example application, VoIP, on top of HTTP-RS. Last, we briefly analyze the role of relay selection in improving performance.

Our evaluation provides two important results about HTTP-RS. First, latency-sensitive applications can achieve latencies on the order of the IP-level latency through the relay server by using small ADUs (< 5-10kB). Second, throughput-intensive applications, can achieve throughput on the order of the available bandwidth through the relay server by using large resources (> 50kB).

8.1 Implementation and Setup

We have implemented S-GET support in our own web-server—which we call 3S (Simple S-GET Server)—as well as in the popular open-source HTTP server Lighttpd. We use 3S to assess the upper bound of the performance for HTTP-RS and we use Lighttpd demonstrate that it is easy to build S-GET support incrementally in an existing commercial web server, without changing HTTP. The latter shows what performance can be obtained for HTTP-RS without optimizing or modifying existing web servers. We have also built support for S-GET in an custom-built HTTP proxy.

S is implemented in C++ and in roughly 3000 lines. The server responds to an S-GET request using the same TCP connection on which the S-GET request was received and keeps the connection open up to the expiration of the S-GET request. Each S-GET request is associated with an in-memory output queue. When the server receives a PUT request, it enqueues the PUT resource into the output queues of the matching S-GET requests.

The Lighttpd implementation builds on the FastCGI module[4]. FastCGI extends the CGI concept to an event-based model, where multiple requests can be multiplexed by a process that handles the requests. The implementation of our FastCGI application is very similar to 3S, in which each SGET requests has its own queue, and PUT requests are enqueued to the corresponding queues.

We have also implemented multiple HTTP clients that use S-GET. The clients are also implemented in C++ (*ca.* 2500 lines). The clients refresh their S-GET subscription by sending new requests on the same TCP connection.

We evaluate HTTP-RS in two contexts. First, we evaluate in a cluster environment to understand the fundamental limitations and upper bounds on the performance of HTTP-RS and S-GET. Our second setup consists of several machines distributed around the world as depicted in Figure 8.1. Machine `A` is located at a university on the west coast of US, `B` is a server located in the geographical proximity, `C` is located inside the data center of a cloud provider on the east coast of US and `D` is located in Europe.

[4]FastCGI support exists for most major web servers, *e.g.,* Apache

Scenario	Direct	Indirect	HTTP-RS - 3S	HTTP-RS - Lighttpd
Cluster	0.114	0.228	0.455	1.303
A-B-C	84.3	87.6	88	88.3
A-C-D	152	176.5	178.3	176.7
B-C-D	156	172.5	173.5	173.7

Table 1: Ping RTT Comparison (ms)

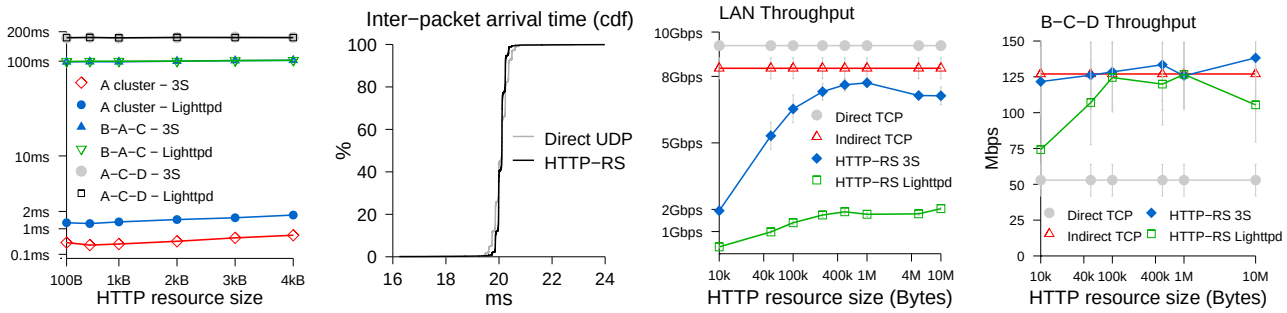
8.2 Latency

We now compare the end-to-end latency of HTTP-RS with that of IP datagram communication. There are three types of latency that HTTP-RS adds: (1) the extra network latency due to using an off-path relay server, (2) the HTTP processing overhead at end-hosts and the relay server, and (3) the delay introduced by the "store–and–forward" model used by the relay server (*i.e.,* the entire HTTP resource needs to be received before being forwarded). We focus to the second and third type of latencies described above (for the first see §7.1).

We compare the regular network latency, obtained using *ping*, with the latency of a ping application implemented on top of HTTP. The HTTP level ping between two machines `X` and `Y` is implemented as follows. Both `X` and `Y` make an S-GET request to a server `Z`. `X` sends a small PUT request matching `Y`'s S-GET; this represents the "ping". After receiving this HTTP message, `Y` sends a PUT response to `X`'s S-GET; this represents the "pong".

We do not differentiate between propagation delay and transmission delay and use small packets of 100Bytes with a 165Byte PUT HTTP header and 75Byte GET HTTP header.[5] Table 8.2 compares the direct client-to-client latency, the indirect latency through the relay server, and the HTTP-level ping latency. The first is most relevant, as it performed in a low-latency cluster and is indicative of the real latency added by HTTP-RS compared to the latency through the relay server. As expected, the latency added by the HTTP servers is small. The next lines present a few data points using the wide area topology from Figure 8.1. In the experiments labeled by three letters such as "`A-B-C`", the endpoints are `A` and `C` and the middle server is `B`. In general, we expect the additional latency added by HTTP-RS in any experiment to equal that observed in the cluster.

[5]We do not account for TCP setup and assume that S-GET requests have been set up. This is the common case we care about when evaluating latency, as setup occurs only once per communication.



(a) Ping RTT vs ADU size (b) VoIP Traffic (c) Throughput Cluster (d) Throughput B-C-D

Figure 7: Impact of HTTP Relay Service on latency and throughput

The other differences arise because of the routes taken (*e.g.,* the route between C and D exhibited a bi-modal distribution of latencies) and router queuing delays.

Figure 7(a) shows the results for the HTTP-level ping when we vary the size of the ADUs exchanged by the end-hosts. The additional HTTP-RS latency is small for ADUs $<$ 10kB. For larger sizes, the store–and–forward latency starts to matter. We note, however, that applications using large ADUs are typically throughput-intensive rather than latency-sensitive, and hence latency is less of a concern. Throughput is evaluated below.

Finally, we evaluate HTTP-RS' impact on latency for VoIP-like traffic. Most VoIP clients send packets at a constant packet rate, *e.g.,* one packet every 20ms. The size of the packets varies with the audio codec and between the talkspurt and quiescent periods. For example, the G.711 [41] codec adopted as a worldwide standard in fixed line telephony specifies a sampling rate of 8000 times/second using 8 bits per sample. VoIP clients which use G.711 aggregate the samples into packets of (at most) 160 bytes at a rate of 50 packets per second.[6] We use this traffic to assess the impact of HTTP-RS on VoIP applications. We use the hosts B and D as clients and compare the round-trip delay and inter-arrival times between direct UDP traffic and HTTP-RS sent through node C. We use instances of several minutes of simulated conversation. We compare against UDP since it has traditionally been the protocol used by VoIP clients[7]. In our experiments, the round-trip times were not affected (same as those presented in Table 8.2).

Figure 7(b) compares the inter-packet arrival times for the VoIP traffic. As one can see, in our experiments, the inter-arrival time of HTTP datagrams had actually lower jitter on average than for UDP. Very few UDP packets were lost, and less than 0.1% of the HTTP datagram packets exhibited inter-arrival times higher than 30ms or lower than 10ms.

8.3 Throughput

In this section, we evaluate HTTP-RS's throughput. Figures 7(c) and 7(d) show throughput for four scenarios: direct communication using the *iperf* utility, indirect communication using a TCP forwarder, and reliable-transfer relay using 3S and Lighttpd. To achieve higher throughput, we use pipelining with HTTP-RS.

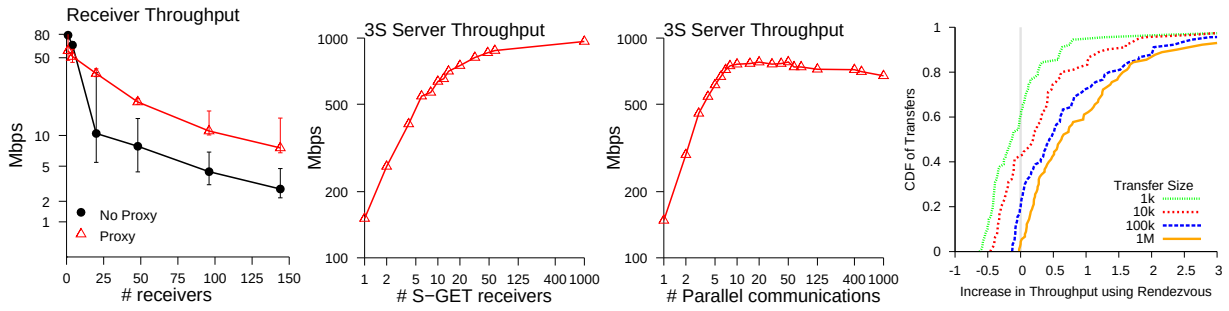
We first want to estimate the upper bound of HTTP-RS's throughput. For this purpose, in Figure 7(c) we use a controlled environment consisting of two machines (with Xeon X5560 CPUs) connected through a 10Gbps link. This experiment shows that a single communication over HTTP-RS can achieve throughput of over 7Gbps when using a server optimized for S-GET, and over 2Gbps even with an standard web server leveraging existing technologies. When smaller resources are used, the throughput decreases due to the overhead of parsing and processing each resource at the server. We expect these results to hold for the wide-area scenario as well.

Figure 7(d) presents a wide-area example where nodes A and D communicate through the relay server C (see Figure 8.1). Interestingly, in this setup, the bandwidth available through the indirection server was significantly higher than the direct bandwidth. As expected from the results in the cluster, for large ADUs, HTTP-RS can utilize the full available bandwidth, which is relatively low for wide-area networks.

We next explore how throughput is affected by S-GET-aware proxies. We let a set of clients receive the same data from a server through S-GET, *i.e.,* the server multicasts the data. Figure 8(a) compares the average throughput observed by clients when receiving the data directly from a server to the case when receiving the data through an S-GET-aware HTTP proxy. Receivers are located on a set of machines in the same organization as host A, while the server is located on host C. The proxy is co-located with the receivers. We send 50MB using ADUs of 50kB. As expected, the throughput is significantly higher when using the proxy because the

[6]Note that G.729, more popular for VoIP, has lower network bit-rate requirements and HTTP-RS' results can only improve.

[7]More recently, VoIP clients are migrating more and more towards TCP to handle firewall traversal and bandwidth elasticity.



(a) S-GET Proxy (b) Throughput Multicast (c) Throughput Parallel Comm. (d) Using Rendezvous

Figure 8: Throughput

bandwidth between the proxy and the receivers is much higher than the one between the source and the receivers.

8.4 Scalability

We now evaluate the throughput of HTTP-RS in the presence of multiple senders and receivers.

Figure 8(b) shows the total throughput for 3S in the case of multiple S-GET receivers and a single sender for several machines inside a cluster at our university connected through 1Gbps links. Figure 8(c) shows the server throughput in the case of multiple client-pairs using the same server; these transfers are executed in parallel at the same time. The communicating participants are distributed on multiple machines and senders are synchronized.

These results show that 3S performance degrades gracefully as the number of senders and receivers of HTTP-RS increases. That is, the total server throughput is mainly limited by the link bandwidth and CPU. As expected, we have found that the scalability of 3S is limited by the one-thread-per-client model rather than the support for S-GET. The throughput of the active HTTP-RS communications is not affected when there are 20k inactive S-GET requests (chart omitted for brevity).

8.5 Size Overhead

We now consider the overhead due to the HTTP header size. The typical HTTP header used in our experiments is 70-80Bytes for the S-GET requests and 160-170B for the PUT requests (*e.g.,* varying with resource name, host name, parameters). Thus, for example, the PUT HTTP header adds an 8% overhead on a 2 KB ADU. Since bandwidth is becoming cheaper, we argue this overhead is manageable in the vast majority of cases.

8.6 Server Selection

In our current deployment of HTTP-RS, a rendezvous step selects the optimal relay server based on client location. The rendezvous step itself, which consists of an HTTP request and redirection response, introduces the overhead of an additional round-trip. An alternative design could avoid this step (for instance by using a consistent hashing scheme) at the cost of converging at a sub-optimal relay. To evaluate this design decision, we studied the effects of the rendezvous step on transfer time between 90 (sender, receiver) pairs across five possible relay choices. In our experiment, a sender sends a single ADU to a receiver who is already listening on the channel. We evaluate the throughput of this transfer both (i) when the sender and receiver chose a random relay and (ii) when the sender and receiver first rendezvous then chose the optimal relay. This experiment is extremely simple, but conservative in its evaluation of using an up-front rendezvous. For longer-lived sessions, the RTT cost would be further amortized over subsequent transfers.

Figure 8(d) gives a CDF of the change in throughput using a rendezvous server compared with a random[8] relay. As expected, very small transfers fare worst, since the time spent on the initial rendezvous is comparable to the actual transfer time. At a transfer size of 1k, more than 50% of connections benefited from using an up-front rendezvous. At sizes above 100k, it is nearly always optimal to use a rendezvous. Since most throughput-sensitive applications transfer at least this quantity of data, we are confident the rendezvous service provides a substantial performance boost. The latency of the rendezvous step was determined entirely by RTT between the client and the state store (via the rv node). Distributing the state store would decrease this latency.

9 Related Work

Clean slate architectural designs have had a large influence on this work (*cf.,* §2 for a comprehensive overview). Recent clean slate proposals have argued that the most desirable feature of an architecture is the capacity for evolution [8,19]. This is achieved by designing protocols with disentangled features which can evolve in isolation from components. Upgrades to such features can be

[8]To qunatify the throughput of a "random" relay, we measured all possible relays, and used the relay with median performance.

implemented by a subset of the parties on the Internet, avoiding the need for forklift upgrades or wholesale replacement of protocols which have historically proved difficult. Viewed in this context, the properties of HTTP outlined in §3 can be seen as *disentangling* various protocol functions (via naming and resolution flexibility) and uses (via named middleboxes).

Our design and implementation of HTTP-RS is heavily influenced by work on publish/subscribe (pub/sub) architectures. Demmer *et al.* advocate for pub/sub as a network primitive [13]. In HTTP-RS, an S-GET to a URI represents a topic subscription and PUTs serve as publish events. Along these lines, our work can be seen as implementing *i3* [39] on top of HTTP. While *i3* depended on a global DHT infrastructure, we have essentially implemented *i3* triggers through S-GET.

Many have observed that functionality should move up the protocol stack. Some propose transport protocols have become the new narrow waist [34], while others suggest evolving transport will require moving pieces of their functionality to higher layers [16, 25]. In this paper, we argue that this is already happening through evolution on top of HTTP.

Finally, industrial efforts are transforming HTTP with modifications that share features with HTTP-RS. Examples of this include BOSH to enable avoiding long-lived TCP connections [23], reverse HTTP to enable easier configuration of HTTP servers [29], and Comet for reducing latency [6, 14]. While we share goals with many of these efforts, we offer a more general service, HTTP-RS, which partially subsumes more targeted approaches.

10 Conclusion

We have argued that the nascent narrow waist of the Internet, HTTP, provides an evolvable Internet architecture. Three fundamental properties of HTTP make it evolvable: (i) flexible names, (ii) support for redirection through name and address decoupling, and (iii) explicit middlebox support. We demonstrate that HTTP extends well to new communication patterns, through the deployment of HTTP-RS, a high performance HTTP pub/sub service. The success of HTTP-RS suggests that proposals advocating new Internet functionality might do well positioning themselves at the HTTP layer.

References

[1] Move Networks. http://www.movenetworks.com.

[2] Swarmcast. http://swarmcast.com.

[3] YURL. http://waterken.com/dev/YURL/.

[4] Cisco Visual Networking Index: Forecast and Methodology, 2009-2014, 2010. http://tinyurl.com/3p7v28.

[5] WhatsApp Blog - One Million, 2011. http://blog.whatsapp.com/index.php/2011/09/one-million/.

[6] A. Russell. Comet: Low Latency Data For Browsers. http://tinyurl.com/5st8ztz.

[7] Adobe Systems. Rtmp specification. http://www.adobe.com/devnet/rtmp/.

[8] A. Anand, F. Dogar, D. Han, B. Li, H. Lim, M. Machado, W. Wu, A. Akella, D. Andersen, J. Byers, S. Seshan, and P. Steenkiste. XIA: An Architecture for an Evolvable and Trustworthy Internet. In *HotNets*, 2011.

[9] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *ACM SIGCOMM*, August 2004.

[10] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by Default! In *ACM HotNets*, 2005.

[11] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *IEEE INFOCOM*, 1999.

[12] D. R. Cheriton and M. Gritter. Triad: A scalable deployable nat-based internet architecture. In *Stanford Computer Science Technical Report*, 2000.

[13] M. Demmer, K. Fall, T. Koponen, and S. Shenker. Towards a modern communications api. In *HotNets-VI*, 2007.

[14] Egloff, Andreas. Ajax Push (a.k.a. Comet) with Java Business Integration (JBI), 2007. JavaOne, San Francisco, California.

[15] K. Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *ACM SIGCOMM*, 2003.

[16] B. Ford and J. Iyengar. Breaking Up the Transport Logjam. In *HotNets '08*.

[17] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI '04*, Mar. 2004.

[18] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. Oasis: anycast for any service. In *NSDI*, 2006.

[19] A. Ghodsi, T. Koponen, B. Raghavan, S. Shenker, A. Singla, and J. Wilcox. Intelligent Design Enables Architectural Evolution. In *HotNets*, 2011.

[20] S. Guha and P. Francis. An End-Middle-End Approach to Connection Establishment. In *ACM SIGCOMM*, 2007.

[21] M. Handley and A. Greenhalgh. Steps towards a dos-resistant internet architecture. In *FDNA '04*, 2004.

[22] M. Handley and A. Greenhalgh. Steps towards a dosresistant internet architecture. In *ACM SIGCOMM Workshops*, 2004.

[23] I. Paterson and D. Smith D. Saint-Andre and J. Moffitt. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). http://xmpp.org/extensions/xep-0124.html.

[24] Internet Draft. RFC3344 - IP Mobility Support for IPv4, 2002.

[25] J. Iyengar, B. Ford, D. Ailawadi, S. Amin, M. Nowlan, N. Tiwari, and J. Wise. Minion—an All-Terrain Packet Packhorse to Jump-Start Stalled Internet Transports. In *PFLDNET*, 2010.

[26] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *CoNEXT*, 2009.

[27] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM*, 2007.

[28] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet Inter-domain Traffic. In *ACM SIGCOMM*, 2010.

[29] M. Lentczner and D. Preston. Reverse HTTP. http://tinyurl.com/627dmrq.

[30] Nate Anderson. P2P traffic drops as streaming video grows in popularity. http://bit.ly/eQse7V.

[31] A. M. Odlyzko. Internet traffic growth: sources and implications. volume 5247, pages 1–15. SPIE, 2003.

[32] L. Popa, A. Ghodsi, and I. Stoica. Http as the narrow waist of the future internet. In *HotNETS 2010*.

[33] S. Ratnasamy, S. Shenker, and S. McCanne. Towards an evolvable internet architecture. In *SIGCOMM '05*.

[34] J. Rosenberg. UDP and TCP as the New Waist of the Internet Hourglass. In *Internet Draft*, 2008. http://tinyurl.com/6fqy4mt.

[35] S. Deering. Multicast Routing in a Datagram Internet., 1991. PhD thesis.

[36] E. Schonfeld. Cisco: By 2013 Video Will Be 90 Percent Of All Consumer IP Traffic And 64 Percent of Mobile, 2009. http://tinyurl.com/nw8jxg.

[37] H. Schulze and K. Mochalski. Ipoque internet study 2008/2009. www.ipoque.com/resources/internet-studies/internet-study-2008_2009.

[38] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of dns-based server selection. In *In Proceedings of IEEE Infocom*, 2001.

[39] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *ACM SIGCOMM*, 2002.

[40] A.-J. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind Akamai (travelocity-based detouring). *ACM SIGCOMM*, 2006.

[41] I. T. Union. Recommendation G.711, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, Nov. 1998.

[42] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.

[43] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on cdn robustness. *SIGOPS Oper. Syst. Rev.*, 36:345–360, December 2002.

[44] W. Xu and J. Rexford. MIRO: Multi-path Interdomain ROuting. In *ACM SIGCOMM*, 2006.