Architecting for Edge Diversity: Supporting Rich Services Over an Unbundled Transport

Fahad R. Dogar
Microsoft Research, Cambridge

Peter Steenkiste
Carnegie Mellon University

ABSTRACT

The end-to-end nature of today's transport protocols is increasingly being questioned by the growing heterogeneity of networks and devices, and the need to support in-network services. To address these challenges, we present Tapa, a transport architecture that systematically combines two concepts. First, it unbundles today's transport such that network specific functions (e.g., congestion control) are implemented on a *per-segment* basis, where a segment spans a part of the end-to-end path that is homogeneous (e.g., wired Internet or an access network) while functions that relate to application semantics (e.g., data ordering) are still implemented end-to-end. Second, it has an explicit notion of in-network services (e.g., caching, opportunistic content retrieval, etc) that can be supported while maintaining precise end-to-end application semantics. In this paper, we present the basic design, implementation and evaluation of Tapa. We also present diverse case studies that show how Tapa can easily support opportunistic content retrieval in online social networks, various mobile and wireless optimizations, and an in-network energy saving service that improves battery life of mobile devices.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

Keywords

architecture, transport, services, middleboxes

1. INTRODUCTION

Today's Internet is much different from the original Internet of the 1970s. Many of these changes have occurred at the network *edge*, in the form of diverse Internet access technologies (e.g., blue-tooth, ultra-wide-band), edge devices (e.g., cell-phone, PDAs, sensors), applications (e.g., content sharing, gaming, sensing apps), services supported by the network (e.g., caching, mobile users), and the nature of network deployments (e.g., unmanaged residential wireless networks [5], data center networks [6]). Supporting this diversity often requires help from the network, which is hard given the "intelligent end-points, dumb network" principle underlying the Internet architecture. As a result, ad-hoc solutions, such as transparent proxies/middleboxes [9, 11], or CDNs are used; these solutions are complex to manage, fragile, and often violate one or more architectural principles of the Internet [24, 15].

As diversity at the edges is only going to increase in the future, it is critical that we develop architectural mechanisms – rather than point solutions – to accommodate it in the Internet architecture. To this end, we introduce Tapa, a transport architecture that systematically supports rich in-network services on top of an unbundled transport.

The unbundling process moves several functions out of today's end-to-end transport into *segments*. Segments span a portion of the end-to-end path that can be considered homogeneous. Some possible segments include: the "wired Internet", a private network owned by an enterprise, or a wireless mesh network. Each segment provides *best effort data delivery service* to the upper layer – functions that may be required to provide this service (e.g., routing, error control, congestion control, etc) are internal to the segment and hidden from higher layers.

On top of segments is the *transfer* layer, which supports end-to-end data transfers overs multiple segments, similar to how IP supports connectivity in today's Internet. The transfer service runs on both the end-points and network elements, called Transfer Access Points (TAPs), that interconnect segments. In Tapa, the transfer and higher layers work at the granularity of ADUs [13] rather than byte streams or packets. The use of ADUs simplifies the deployment of in-network services, making it easy to leverage advances in technology (e.g., cheap in-network storage).

Finally, the *session* layer implements specific end-to-end application semantics over the transfer layer. The presence of the segment layer makes the session layer lightweight, isolating it from the details of specific network technologies thus making it easier to support heterogeneity in the system. The introduction of segments, combined with the use of ADUs, means that it is easier to implement end-to-end protocols with diverse semantics required by different types of applications (e.g., fully reliable, streaming applications, etc.) It also becomes easier to insert services into the end-to-end path, while maintaining specific semantics between the end-points and the (possibly third party) network service.

Like most architectural proposals, several building blocks used in Tapa are inspired by prior work, including DTNs [23], middleboxes [38], proxies for wireless and mobile users [9], and data/content oriented networking [37] to name a few (See §6 for a detailed discussion on related work). Our core contribution lies in reusing their well understood benefits, modifying them to match our needs, and most importantly, *synthesizing* them in a new architecture.

Another important contribution of this paper is to demonstrate the flexibility of Tapa using three case studies (§4). In Vigilante, we show how Tapa can be used to efficiently disseminate content in online social networks, resulting in improved performance for end users and lower cost for the provider. In Swift, we leverage Tapa to provide various optimizations for mobile and wireless users, such as multiplexing multiple interfaces or optimizing data transfer in a mobile scenario. Finally, Catnap is an in-network traffic shaping service that leverages Tapa to provide significant energy savings to mobile devices.

Finally, as Tapa makes in-network services visible to endpoints, legacy applications need to be rewritten if they want to make full use of Tapa functionality. While this may prove to be a stumbling block towards the adoption of Tapa, we show that this modification effort is manageable. Moreover, our experiences also show that new segments and services can easily be deployed inside Tapa. We discuss these experiences in §5.

2. REQUIREMENTS AND CONCEPTS

We first discuss two key requirements that are important for a new transport architecture.

1. Accommodating Diverse Networks and Devices The original ARPANET was designed to inter-connect heterogeneous networks and hosts. However, by today's standards, they were in fact fairly homogeneous. For example, it was assumed that links would provide "reasonable" reliability ($<$ 1% loss rate) [12] and that all end-hosts would be powerful enough to support the full TCP/IP stack. The networks that make up the Internet now are very diverse, ranging from very high speed optical backbones, to low speed, unreliable wireless networks that have very different properties from wired links (e.g., higher error rates, variable bandwidth and delay), causing problems for end-to-end protocols such as TCP [9, 22]. Similarly, devices, such as sensors and mobile nodes, are highly resource constrained, compared with traditional endpoints. Dealing with the heterogeneity of devices and networks affects how we distribute functions across devices. For example, when applying the end-to-end argument [35], we can no longer view the communication subsystem as a *single homogeneous module.*

2. Accommodating network services Most applications in today's Internet benefit from diverse network services, e.g., web caching, video transcoding, various services provided by load-balancers inside data center networks, services for mobile and wireless users, etc [21, 34]. Unfortunately, inserting services in an end-to-end path is hard in today's Internet for two reasons. First, the data plane of existing protocols (IP and TCP) works with packets and byte streams, requiring tight coordination between applications and services on how to interpret the data (e.g., object size, naming). Second, and more importantly, TCP's end-to-end semantics are very rigid – they do not allow a role for an intermediary or accommodate different delivery semantics.

Limitations of existing solutions. Today, the above two requirements are addressed in an ad-hoc manner, bringing more complexity and rigidity into the architecture. For example, a common way to deal with network heterogeneity is to use the transparent split proxy approach, with different transport regimes/protocols on either side of the proxy (e.g., I-TCP [9], PEP [11]). Similarly, network services are typically hidden from the end-points (e.g., transparent web proxies) or deployed as overlays. This is not ideal, since there can be poor interactions with other services, like firewalls, and sharing services between applications, users and providers is hard. These hidden, stateful middleboxes also create new failure modes and make it difficult to support mobility (e.g., migrating VMs in a data center or clients in a mobile environment).

On the other hand, a clean solution that accommodates diverse networks and devices, in addition to supporting in-network services, can potentially provide substantial benefits to users, application providers, and network service providers. As we show in this paper, end users and application providers can get improved performance with robust communication semantics. Having visible in-network services can simplify the job of the ISPs, making it easier to deploy new services, including having an explicit role as an advertiser. Moreover, they can also reduce their costs with the use of content centric features supported by Tapa.

2.1 Key Concepts

We now present the two key concepts that form the basis of the Tapa architecture and help in meeting the above requirements.

We unbundle today's transport to better accommodate the needs of heterogeneous networks and in-network services. The unbundling process has both vertical (across layers) and horizontal (across the network topology) dimensions. As a first step, we propose *decoupling network regions with very different properties*, such that the properties of one region do not affect the other. For example, adequate in-network buffering can hide the losses that may be experienced in one region from the other region. Decoupling facilitates the deployment of customized solutions for each region as solutions designed for one region need not worry about the properties of other regions. Decoupling affects modularity of the system in the horizontal direction as it requires the network to support some functions (e.g., buffering). Moreover, nodes that implement decoupling can be used to insert additional functions (e.g., data oriented and higher level services) inside the network.

As a second step, we propose to *raise the level of abstraction that the network provides to the end-to-end layer that implements specific application semantics (e.g., TCP)* as this may make it easier to "hide" diversity at the lower layers. So other than the functions that must be implemented with the help of end-points [35] (i.e., specific application semantics) all other functions are pushed down, affecting modularity of the system in the vertical direction. This allows separation of concerns: the lower half of transport can focus on network specific challenges (with the help of the network) while the upper half of transport can focus on the semantics of applications and higher level services.

The first concept, specifically the notion of decoupling, leads to our second concept: *rich, visible in-network services.* We propose that these services operate at the granularity of

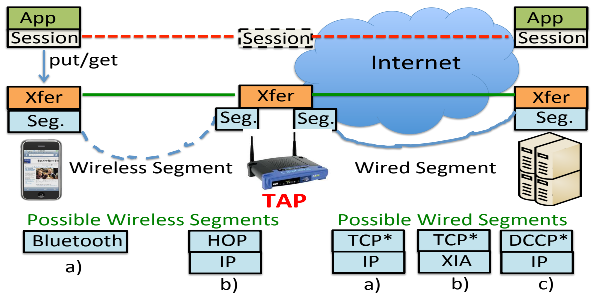


Figure 1: Tapa Overview.

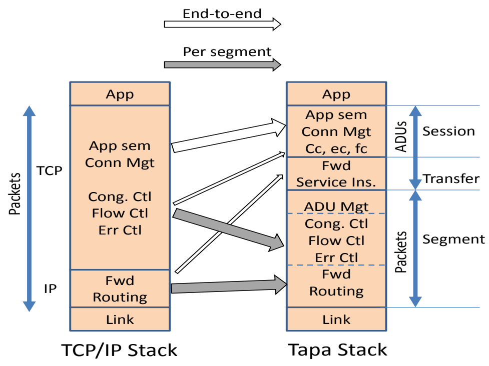


Figure 2: Comparison of protocol stacks.

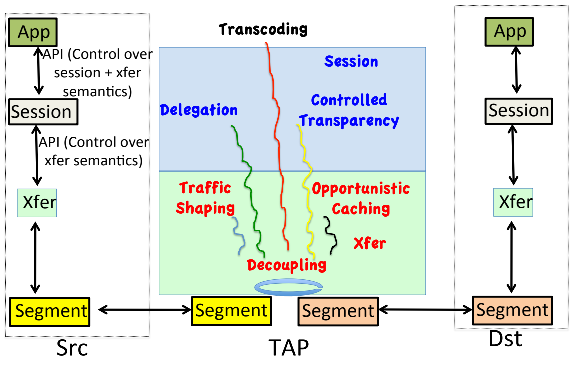


Figure 3: Different types of services in Tapa.

ADUs, which can be defined by applications to meet their needs. For example, an ADU can correspond to a file (e.g., Catnap[20]), a chunk within a file (e.g., DOT[37]), or an MPEG frame in a video transfer. ADUs can be identified using self-certifying labels i.e., the ADU identifier is a hash of the content [37]. This enables several data-oriented optimizations, such as opportunistically fetching content from arbitrary sources. In addition to the identifier, we also associate *hints* with an ADU, such as possible sources from which the ADU can be retrieved etc. As our case studies show, ADU based services play a central role in enabling applications to benefit from content centric functionality inside the network.

3. TAPA

We introduce Tapa, or TAP-based Architecture, which synthesizes the above two concepts into a coherent architecture. As a result of unbundling, most of the traditional transport functions are put inside the *segment* abstraction. As segments provide a best effort service to the higher layers (i.e., they deliver data with high probability, but delivery is not guaranteed), they appear as Internet-style "links" to the higher layers.

Figure 1 illustrates a typical end-to-end data transfer in Tapa consisting of two segments (wired and wireless). Similar to how IP supports connectivity in today's Internet, Tapa's *transfer* layer supports end-to-end transfer of application data units (ADUs) [13] over multiple segments. It supports two modes of ADU transfers: a push mode, which is useful for interactive applications (e.g., gaming, video conferencing, etc) and a pull mode, which is useful for data oriented applications as it allows an ADU to be retrieved from potentially any source based on the ADU identifier. Finally, a light weight *session* layer implements specific application semantics related to four functions: reliability, data ordering, confidentiality, and data integrity. It can support traditional semantics (e.g., end-to-end reliability, data ordering, etc) or richer semantics involving intermediate services (e.g., delegation).

Figure 2 shows how the transport functions that are currently bundled in TCP are distributed across the Tapa layers. The end-to-end application semantics and connection management function are placed in the session layer. Some connection management is also needed as part of the segment layer. The flow, error, and congestion control functions are distributed over the segment and the session layers. In the session layer, these functions are lightweight because they only have to operate over a limited number of segments/hops.

In the segment layer, the functions can be very lightweight (e.g., a point-point wireless link), or complex (i.e. an Internet segment). For Internet segments, this means that we can view Tapa as an overlay, as the segment layer can be broken up into an ADU management layer on top of a TCP (or TCP-like) and IP layers, as suggested by the dashed lines. For single-link segments, Tapa can be viewed as running native, as IP or transport functions are not required.

We now elaborate on the role of services, followed by a discussion on routing and addressing. Finally we describe each of the three layers of Tapa in detail. To illustrate the key functionality, we make use of a running example where a mobile and wireless user, Alice, uses Tapa to communicate with a fixed host over the Internet (similar to Figure 1).

3.1 Services

As shown in Figure 3, Tapa can support various types of services at the transfer and higher levels of the system. Transfer services enhance the data transfer function without affecting the typical end-to-end application semantics. The most basic example is a *decoupling* service that isolates one segment from the other by providing adequate buffering at the TAP. We can also deploy various data oriented services such as application independent caching at TAPs, which can help mask disruptions for mobile users (Swift, §4.2) or by acting as a storage service for content distribution (Vigilante, §4.1). Similarly, we can also have a traffic shaping service that can allow end-devices to sleep by temporarily buffering data (Catnap, §4.3).

Session-level services support end-to-end communication semantics that may be requested by the application. Examples include helping with error recovery for reliable data transfer semantics (i.e., delegation), or an in-network service inspecting encrypted data for performance or policy reasons (i.e., a form of controlled transparency [14]). We provide examples of several such services in §3.5. Finally, application-level services can provide application specific functionality, such as transcoding.

3.2 Routing and Addressing

Today's transport protocols (e.g., TCP) use IP addresses to identify communicating end-points and it is the responsibility of the underlying network layer (i.e., IP) to provide reachability to these identifiers. As IP identifiers are tied to a specific location/address, their use as identifiers makes it difficult for transport protocols to deal with mobility.

For Tapa, mobility is a common use-case, so instead of relying on ad-hoc solutions, the architecture should handle mobility in a graceful fasion. Consequently, similar to many other achitectural proposals [36, 38], Tapa requires that identifiers should be separate from locators, so even if an end-point moves, its identifier remains the same.

Similar to several recent proposals [8, 26], Tapa uses self-certifying flat labels as identifiers, although other options that also distinguish between locators and identifers can also be considered [27]. A self-certifying label corresponding to a host corresponds to the hash of the public key of the host. We can also have similar certifying labels to identify services and content [26]. The use of self-certifying identifers provides several desirable security properties, such as host/service authentication and data integrity verification [8, 26].

Host/Content identifiers are provided by the application and passed down to the segment protocol which is responsible for routing on these identifers. While routing on flat-labels is still a challenging problem, we rely on prior and ongoing work that aims to provide practical solutions to this problem [8, 26]. For example, XIA adds a topological identifier (i.e., identifer of the autonomous domain) to the host identifier to form the host address.

A related question is the role of TAPs in routing. We note that TAPs are not required for end-to-end reachability – even without TAPs, applications can still communicate directly by establishing a direct segment between end-points. However, we still need a TAP discovery mechanism if we want to make use of TAPs and the services they offer. While we leave the design of a generic TAP and service discovery protocol for future work, we do discuss some point solutions that we have built as part of the case-studies, and also point out relevant solutions from prior work. This includes TAP discovery that uses help from lower layers (e.g., similar to AP discovery in WiFi) as is the case in Swift (§4.2), making use of social network information (Vigilante - §4.1), via service discovery protocols [3], resolution services [38], or the application providing this information (e.g., as part of the end-point address, similar to a DAG-based address in XIA [26]).

3.3 Segment Layer

The segment layer is responsible for transferring ADUs across a segment, e.g., client - TAP or TAP - server. Segments can choose the data granularity they use internally (e.g., frames, bytes, etc.) allowing them to optimize communication as appropriate. Segment endpoints do not move, so they *can* use network-specific locators as addresses. For example, in the wired Internet IP addresses based on CIDR may provide the necessary scalability, while MAC addresses may be more appropriate in wireless networks.

The segment layer must be able to translate the identifiers corresponding to the TAPs and end-points into "locators" that can be used to establish the segments; locators may only have meaning locally within a network, i.e. they do not have to be globally unique. Higher layers of Tapa use identifiers and it is the responsibility of the segment layer to resolve them internally to locators. This can be done using either a name service (e.g., DNS for wired Internet segments), or a locally maintained mapping (e.g., MAC addresses for a bluetooth based segment).

3.3.1 Basic Operations and API

As shown in Table 1, the segment API allows higher layers to transfer ADUs across a segment. Similar to today's Internet, the segment layer supports a push model. Senders use a `send` call while receivers `register` a callback to receive incoming ADUs.

Recall that the basic motivation behind segments is to decouple different types of networks in an end-to-end path and to make them fairly homogeneous, so that higher layers need not worry about network specific functions. To achieve this, segments must implement error, flow, and congestion control as needed. Tapa's transfer service expects segments to be *reasonably* reliable, but segments can use very different ways for achieving that, e.g. TCP style retransmissions and ACKs over wired segments versus in-network coding or opportunistic forwarding and block acknowledgments on a wireless segment [30, 10]. Segments reassemble ADUs and deliver them to the transfer layer. In some scenarios, applications may want to send or receive ADUs piecewise for performance reasons; we give an example in §4.3.

The benefit of tailoring the congestion control algorithm to the type of network has similarly been recognized [9, 30]. Some segments may need specialized routing protocols for delivering data over multiple hops within a segment (e.g., wired Internet or mesh network). Some segments (single hop, point-to-point) may not require routing or congestion control. The important point to note is that all these mechanisms are hidden from the transfer and session layers, which focus on application oriented functions.

Figure 1 shows some possible ways to implement these functions for wired and wireless segments. On the wired side, we can use variants of existing transport protocols (e.g., TCP, DCCP) over different network layers (IP, XIA [26]), while on the wireless side we can use completely different protocols. In §5.2, we describe various segment protocols that we use in the Tapa prototype.

Alice and the segment layer: Alice can use various segment protocols on the wireless side. If she is connected to a wireless mesh network, she can use a segment protocol like HOP [30], which relies on per-hop mechanisms for reliability and rate control. She can also bypass IP and traditional link layers, using options as diverse as blue-tooth or wireless USB. This is a direct benefit of raising the level of abstraction – not all networks and devices need to implement heavy-weight functions of TCP/IP (e.g., congestion control) in order to be part of the Internet.

Function	**Description**
Session Layer Interface for Applications	
`get(ADU-id, hint, src, session + xfer options)`	Call used to pull an ADU.
`put(ADU, dest, session + xfer options)`	Call used to push/publish an ADU.
Transfer Layer Interface for Session/Higher Layers	
`get(ADU-id, hint, src, xfer options)`	Call used to pull an ADU.
`put(ADU, dest, xfer options)`	Call used to push/publish an ADU.
`register(ID, handler)`	services/apps registering with the xfer layer to receive ADUs destined to them.
Segment Layer Interface for Higher Layers	
`send(ADU)`	Call used to send the ADU across a segment
`register(ID, handler)`	registering to receive incoming ADUs

Table 1: Interfaces used at different levels of the Tapa system. Identifiers (e.g., src, dst, ADU id, etc,) are self-certifying labels. Session and xfer options are a list of (type, value) pairs.

3.4 Transfer Layer

The transfer layer is the "inter-segment" layer of Tapa and its role is somewhat similar to that of IP in today's Internet: providing a best effort data delivery service over multi-hop paths. There are however some differences in both the control and data plane. For example, Internet routing needs to establish routes in large scale but fairly stable networks; in contrast, the Tapa transfer layer establishes short (e.g., two-segment) paths but paths can be very volatile due to the dynamics of the edge network (e.g., mobility and wireless dynamics, volatility in data center networks). The introduction of a transfer layer allows for a separation of concerns. The segment layer can focus on the challenges associated with specific networks (e.g., scalability within core Internet) while the transfer layer can focus on dealing with higher level challenges (e.g. selection of segments based on content or service availability, or mobility).

3.4.1 Basic Operations and API

The service supported by the transfer layer is the transfer of ADUs based on a push/pull model as described below. The push and pull modes of the transfer layer are used by the higher layers through the put and get interfaces, respectively. The same API is used by both the end-points as well as by services running at the TAPs. As shown in Table 1, the API takes in an ADU, which is mandatory, hint, src/dst, and *xfer options*, which is an *optional* list of (type, value) pairs that are used to provide more control to higher layers so that they can choose different transfer semantics, if they desire so. The semantics are optional – if they are not specified, default transfer semantics are used. We first describe the working of the two calls with default semantics and then elaborate on optional semantics.

Pull Mode: The `get` interface allows higher layers to retrieve an ADU based on its identifier, irrespective of which source or transfer mechanism (e.g., choice of segment or interface) is used to retrieve it. The flexibility means that the lower layers are free to choose any source, including local storage,

Applications can aid this decision by providing hints regarding hosts who *may* have the data (e.g., friends in a social network - see Vigilante (§4.1), or a previously used TAP in case of a mobile user – see Swift (§4.2)). We do require the higher layer to specify a source (src) that is *guaranteed* to have the ADU (e.g., a server), so hints are completely optional and are just intended as an optimization. The combined use of a hint and src is somewhat similar to XIA's use of intent and fallback [26]: hints serve as optimization, but if they don't work (or are not provided) then the source is used to retrieve data. This ensures that data is retrieved even if the network cannot "route" on the ADU identifiers or the hints fail. Having the src information, in addition to the content indentifer, is a pragmatic decision given that scalable Internet-scale routing on flat content identifiers is still an open problem.

To inititate an ADU transfer, the transfer layer establishes an end-to-end path between the client and the server passing through the TAP(s). The underlying segment protocol can create a new segment or reuse an existing one, if one already exists. The ADU request is sent to the other end-point and data is returned as a result. In the typical scenario, the data plane of the transfer layer is relatively straightforward: TAPs read data from one-segment and write to the other, ensuring that adequate buffering is provided.

We can also have scenarios where segments may have high volatility or TAPs use more sophisticated services that change the typical behavior of the data plane (e.g., Catnap - §4.3). In such scenarios, suitable buffering at the TAP becomes critical for end-to-end performance. Our experience so far has been limited to various kinds of wireless segments and the challenges they pose for provisioning buffer space at the TAPs. In our case studies, we discuss how additional buffering at the TAP is used in various scenarios.

Finally, the `get` call returns the actual ADU or a failure notification if the ADU cannot be retrieved. Note that because the higher layers are letting the transfer layer make several transfer decisions, they are likely to observe fewer failures compared to using today's socket API because the transfer layer has more options for failure recovery.

Push Mode: The `put` interface is used to publish an ADU to a specific destination (dest), which could be a storage service (local or remote) or any other service running on a host. Applications can use this API to publish their data *once* and a generic storage/transfer service can later take care of serving it to future client requests (similar to DOT [37]). This enables temporal decoupling between the publisher and the consumer, so both of them need not be present at the same time. The push mode sends an ADU to a given destination, similar to how IP datagrams are pushed from a source to a destination. Its overhead is also comparable to that of sending UDP datagrams in today's Internet (see Figure 5.3).

A key reason for supporting a push mode is to facilitate real-time applications that cannot benefit from data oriented optimizations. Unlike the pull mode, which requires an extra round trip time for the ADU request, the push mode directly sends an ADU to the intended recipient. Even though Tapa's data oriented optimizations are not applicable in this mode, there are still benefits to using this mode compared to using today's protocols. For example, using appropriate segment protocols, instead of a single end-to-end transport, can provide better performance for real-time applications. We expect the push mode to be used in scenarios where the pull mode is not applicable, such as transferring dynamic data, publishing content or exchanging control messages. The protocol followed in the push mode works similar to the pull mode except that no ADU request precedes the data ADU.

Different Semantics: Higher layers can choose different transfer semantics by extending the basic get and put calls with (type, value) options. Some types that are relevant for the transfer layer include: TAP, service, and segment. Higher layer could use a specific TAP for its communication (e.g., an enterprise TAP for policy reasons) or it can specify a specific type of segment that should be used. For example, a "slow" segment could be used for background transfer of data that is not immediately required, thereby reducing the load on the server.

Alice and the transfer layer: Tapa's transfer layer can help a mobile user like Alice in several ways. First, it can improve performance of data transfers through caching and intelligent use of segments. Consider an example where Alice is downloading a large file from a slow server. Initially she is using her home TAP but in the middle of the download she moves to Starbucks. Her home TAP will continue to download ADUs from the server and cache them in its storage; after she moves to Starbucks, the transfer layer will re-establish a segment to her home TAP via the Starbucks TAP and retrieve the missing ADU (by re-issuing requests for the missing ADU ids). This will be a much faster option compared to going all the way to the slow server to retrieve the missing ADUs. Note that the get call is used for this purpose and the Starbucks TAP information is provided as a hint.

Second, ADUs and segments simplify the use of wireless optimizations that allow multiplexing of multiple interfaces or access points (TAPs in this case). Such optimizations are not only difficult to implement in today's Internet, but are also tied to a specific transport protocol (e.g., TCP) or specific underlying technology (e.g., WiFi) [28]. In Tapa, multiplexing of segments is naturally supported: the transfer layer decides an appropriate segment for each ADU, so for the same application, it can fetch some ADUs using one particular segment while other ADUs can be retrieved using a different segment.

Third, TAPs can pre-establish long-lived segments with popular websites or frequently accessed WAN locations, providing Alice low latency access for interactive communication. Alice will only establish a segment with her local TAP, which will be close-by in most cases, and the TAP will use a pre-existing segment to communicate with the other end-point, gaining vital latency savings on the long RTT part of the end-to-end path. Such techniques are already used by CDN providers to accelerate dynamic website access [2]; with Tapa, such techniques can be used at users' TAPs.

3.4.2 *Resource Management*

Although congestion control is implemented inside segments, the transfer layer also needs to ensure that TAP buffers do not overflow. We include this broadly under *resource management*, which includes congestion control across a multi-segment path as well as protection against malicious sources (e.g., DoS attacks)

Several Tapa features support a holistic approach towards resource management, enabling TAPs to consider a variety of techniques for congestion prevention, avoidance, and control. For example, Tapa's use of ADU hints, which include the length of the ADU, provides information to the TAPs about future traffic load, which can be used to do admission control. TAPs strategic location at network edges means that they can also coordinate with end-points to install filters against potential DDoS attacks, thereby acting as a first line of defense [31]. The limited number of segments in an end-to-end path means that it is easier for TAPs to coordinate amongst each other in order to avoid congested segments. For example, many techniques that are difficult to use in Internet settings, such as hop-by-hop flow control based on back-pressure (implemented in our prototype) or end-to-end congestion control based on feedback from the network, become feasible in Tapa.

3.5 Session Layer

Similar to IP in today's Internet, the segment and transfer layers offer a best effort service to the transfer and session layer, respectively. This means that they deliver data with high probability, but delivery is not guaranteed. The motivation for the best effort nature of the segment and transfer layers is the same as for IP [35], i.e., full network-level reliability is expensive and not always needed. So the reliability provided by the segment layer is purely a performance enchancement, similar to how, in today's Internet, we have reliability at the link layer (e.g., 802.11) despite having TCP's end-to-end reliability. This follows the end-to-end arguments [35], as lower layers of the system *may* have reliability for performance reasons, but we still need end-to-end reliability for correctness/robustness reasons.

Tapa's session layer provides reliability and other application semantics (e.g., ordering, confidentiality, data integrity, etc) to the end-points, albeit over a very short path consisting of a small number of segments. An important thing to note is that we can easily implement support for different semantics (e.g., partial reliability or out-of-order delivery) as the session layer need not re-implement network specific functions, like congestion control, which are coupled with application semantics in TCP. Another important consideration is that TAPs are visible to the end-points, opening the door for richer semantics that explicitly capture the role of services running on the TAP.

3.5.1 *Basic Operations and API*

The session API is similar to the transfer API; the only difference is that it takes additional options that are relevant to the session layer semantics. In our design, this includes semantics associated with the four functions discussed earlier: reliability, confidentiality, ordering, and data integrity. If applications do not specify any additional option then the default semantics associated with these functions are chosen. We now describe a specific session protocol that supports semantics associated with these four functions. We briefly present the traditional semantics as well as the new semantics that emerge with the use of in-network services.

Reliability: Applications can choose a fully reliable data delivery (default) or a best effort delivery service. We focus on the reliable delivery case because it is by far the most common use case. For a reliable service, the session layer holds on to the ADU until it is acknowledged by the other end-point. If required, the session layer has to undertake recovery in the face of different kinds of failures (e.g., TAP failure). These are the traditional end-to-end reliability semantics that are offered in today's Internet as well. However, because we also have reliability within segments, Tapa's session layer is likely to experience fewer failures compared to today's transport (TCP) and such instances are mostly limited to TAP failures. In §5.3.2 we quantify the overhead of such end-to-end recovery, showing that we can efficiently recover from TAP failures.

Tapa can also accommodate different reliability semantics where an end-point can *delegate* data transfer responsibility to an intermediate storage service. So even if the sender disconnects after completing the transfer to the intermediary, the transfer is not affected. In such scenarios, applications may be interested in differentiating between when the data is received by the TAP compared to when it is received by the end-points. Of course, this information can be used by the user or application in a number of ways. For example, the application may discard the data if it is received by the server but may like to hold on to the data if it is only received by the intermediary. Our implementation provides allows applications to choose the desired semantics from these different options.

Confidentiality: Tapa can make use of existing security protocols (e.g., SSL/TLS) to provide end-to-end confidentiality and data integrity. Moreover, it opens up new semantics that are not supported in today's Internet. For example, Tapa's use of ADUs allows applications more flexibility in implementing their confidentiality requirements as they can make fine-grained decisions on whether some data needs to be encrypted or not. The main benefit of Tapa, however, comes with the use of in-network services combined with the confidentiality semantics, allowing them to selectively look at certain data with the explicit permission of the application. This is important for both policy and performance reasons. For example, some intermediaries, like an enterprise or government, may require looking at certain types of data. We allow applications to explicitly state whether certain ADUs can be seen by intermediaries or not providing a form of "controlled transparency" [14]. So applications can hide ADUs that contain sensitive information, like credit card number or social security number, while allowing the company TAP to look at other data ADUs.

Similarly, the use of an intermediary with the confidentiality semantics can also result in performance benefits. One advantage is that it can reduce the burden of maintaining SSL connections for servers by shifting it to third party services running at TAPs. Specifically, the intermediary can maintain a *single* encrypted connection with the server and multiple clients can have their individual encrypted connections with the intermediary. Note that as the intermediary service could be provided by a third-party (e.g., an ISP), the semantics of this communication are very different from those of end-to-end encryption as the application/user is also trusting a third party and explicitly involving it in the confidentiality semantics.

Another advantage is that it can improve the effectiveness of redundancy elimination techniques, which are ineffective if the traffic is encrypted [7]. Specifically, applications/user can explicitly involve a trusted intermediary (i.e., ISP) in implementing confidentiality, such that the intermediary can decrypt the data as it enters its network, use RE techniques inside its network, and then encrypt the data again before sending it to the destination/next ISP. Of course, users will only choose weaker confidentiality if ISPs give them additional incentives to do so.

Data Integrity: In addition to verifying end-to-end data integrity, we also allow intermediaries to be involved in these semantics as there are scenarios where there could be legitimate reasons why an intermediary may change the bit-stream (e.g., transcoding, virus scanning proxies, etc). By explicitly involving the intermediary, applications can verify that the data was changed *only* by a legitimate party (i.e., the intermediary) and not by someone else. Data integrity is also closely tied to confidentiality as an application may allow intermediaries to look at the encrypted data but not to modify it (e.g., government or an ISP). With flexible data integrity and confidentiality semantics, applications can choose how an intermediary can read or modify the data.

Ordering: Tapa's use of ADUs helps the session layer in supporting different ordering semantics; these semantics determine whether the session layer presents these ADUs to the application in the same order in which they were generated by the other application end-point, or presented in the order in which they are delivered by the network. Of course, this is based on the assumption that the network does not provide any guarantee with respect to the ordering of ADUs. In fact, the use of Tapa may cause greater reordering compared to today's Internet because of Tapa's use of multiple segments for transferring ADUs.

Alice and the session layer: While Alice can benefit from all the rich semantics offered by Tapa's session layer, the one feature that is specifically useful for mobile users is *delegation*. Alice can benefit from delegation in the following way. Suppose Alice is chatting with her friend, with both users on the road, in a highly mobile setting. With end-to-end TCP, they can only communicate when both of them are simultaneously connected. Using Tapa's delegation semantics, the chat application can delegate the transfer responsibility to the TAP; in this case, both Alice and her friend would be interested in differentiating between whether their messages are actually received by the other end-point or just the TAP. Similar functionality is already offered by a popular messaging application, WhatsApp[1], which lets mobile users know whether their message was received by the intermediate server or the other end-device. With Tapa, any application can make use of this generic delegation service while maintaining correct semantics.

4. CASE STUDIES

We now present three case studies on how Tapa can be used to support diverse services. To support these services, we designed and implemented a proof-of-concept prototype of Tapa that focuses on the data plane of Tapa and a control plane that deals with two-segment paths: a wireless

[1] www.whatsapp.com

Vigilante
• put(ADU, serverAdd, reliability = "delegation", (first)segment = "fast", (second)segment = "slow")
• get(ADU-id, hint (friends), serverAdd)
Swift
• get(ADU-id, hint (previous TAP), serverAdd)
Catnap
• get(ADU-id, hint (ADU-length), serverAdd)

Table 2: Some sample API calls used in different case studies.

segment between the client and TAP and a wired segment between the TAP and server. We present the prototype evaluation in §5 while more details of the prototype design and implementation are available elsewhere [19, 17].

4.1 Content Distribution in OSNs

Online Social Networks (OSN) have massive popularity. Facebook (FB) alone has more than 500 million users spread all over the world. Supporting this huge scale is a major challenge for the OSN provider. Despite significant investments in network and data center infrastructure, ensuring a high quality user experience still remains challenging. In order to address the problem of scaling OSNs to billions of users, we have designed and prototyped Vigilante, a system that uses TAPs (e.g., home AP, media center, etc) for storing and distributing online social networking content.

Leveraging Tapa: Tapa facilitates the use of TAPs for storing and distributing OSN content. It provides support for *publishing*, *retrieving*, and *caching* content in an application independent manner. Social networking applications can *aid* these content centric functions of Tapa by providing *hints* or by pushing content in advance to suitable TAPs for performance or availability reasons.

Hints provide information to the TAP on the likely nodes that may have the required ADU in their cache. Hints can be generated and managed in a variety of ways. The simplest solution is to have application generated hints stored on a centralized server. For example, when Bob uploads content, it can include a hint that he is keeping a copy on his TAP. Similarly, after downloading Bob's photos, Lisa can store a copy on her TAP and notify the server. Alternatively, hints can also be managed in an application independent manner by Tapa, for example by having TAPs exchange information about the content (i.e., the self-certifying ADU identifiers) they are caching.

Tapa can also easily support a variety of roles for the TAP in the context of OSNs. In the simplest case, the TAP just maintains a cache of content stored at the server, as described above. However, it is possible to make the TAP the primary source of content, with the server simply keeping a backup copy. It is also possible to control when data is uploaded to the server, for example to reduce server peak loads. The precise role of the TAP can be controlled by the application by associating specific semantics with the transfer, using Tapa's API.

Photo Distribution in OSNs: As part of Vigilante, we have built a photo distribution application that allows users to publish and retrieve photos within a social circle. The working of the system is best illustrated through an example where Lisa wants to share her photos with her friends and

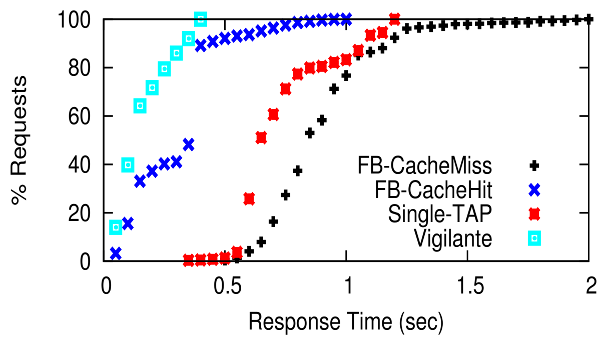


Figure 4: Performance comparison of various schemes for downloading photos.

family. Her social networking application uses Tapa's `put` API to publish the content to both the server and her home TAP. Data transfer from her device to the TAP is done using a normal/fast segment, while the data transfer from the TAP to the server is over a "slow" segment because the server only stores the data for backup purposes. All this is managed through a single put call by controlling the appropriate session and transfer semantics (see Table 2).

The application also sends the meta-data associated with the photo to the server. In addition to other things specific to the photo, the meta-data also includes the ADU identifier corresponding to the photo and information that it is cached at Lisa's TAP. When Bob later contacts the server to get his latest news-feeds, the meta-data corresponding to Lisa's photo is sent to his social networking application. As shown in Table 2, his application uses Tapa's `get` API, specifying the ADU identifier as well as a *hint* that it is likely to be available at Lisa's TAP. Bob's TAP will retrieve the ADU from Lisa's TAP, cache it and serve it to Bob's application. A notification is also sent to the server so that it can update the hints i.e., that Bob's TAP also has a copy of the photo. Later, when Julie wants to retrieve the photo, the hints contain information about TAPs of both Lisa and Bob, so Julie's TAP can choose the one that can provide the better service (we base this decision on the RTT between the nodes). As more users access the photos, more copies are created, helping load balance future requests and also improving the fault-tolerance of the system. A final point is that when Lisa first published the photo, it was also possible to pre-load some other TAPs using the same put API; this further improves the performance and fault-tolerance of the system.

Evaluation: We have conducted an evaluation of Vigilante on the PlanetLab testbed, using more than 150 nodes spread all over the world. We compare the performance of Vigilante with a more centralized design, such as the content distribution infrastructure used by Facebook. Note that Vigilante's performance is comparable to what we might get with a customized P2P based solution, so the key point of this case study is to highlight the flexibility of Tapa in supporting a diverse content distribution model without requiring any changes to the underlying mechanisms.

We run the client application as well as the TAP on the same node. We focus on the performance of photo downloads and consider four schemes/scenarios in this regard:

1. Vigilante: This refers to the implementation described in the previous section but without any pre-loading.

2. Single-TAP: Only the publisher's TAP, which is located in the US, serves the content. We test this scenario by disabling updating of hints.

3. FB-CacheMiss: This refers to downloading the photo from the root photo server of FB. This is the worst case performance with using FB.

4. FB-CacheHit: This refers to the case when the photo is served from the nearest Akamai CDN. This corresponds to the best case for downloading a photo from FB.

We present the results of an experiment under a non-bursty traffic load scenario. We generate a schedule where the consumers make requests to download a 80kB image in a sequential order at roughly 1 photo request per second. This schedule is repeated 10 times and the same process is followed for all four schemes. We present the results for clients who are located outside USA: they have a latency of at least 100ms with the publisher. On the y-axis, we have the CDF of client requests and on the x-axis we have the response times.

As shown in Figure 4, performance is poor for both Single-TAP and FB-CacheMiss as the consumer has to retrieve the content all the way from US, which adds considerable latency. On the other hand, both Vigilante and FB-CacheHit are able to find a nearby cached copy. However, we observed that non-US sites, in general, had a higher latency to the nearest Akamai/Facebook CDN compared to US sites and therefore Vigilante provides greater benefits for such consumers.

4.2 Mobile and Wireless Optimizations

Through Alice's example in the previous section, we have already explained how Tapa can help a mobile and wireless user. We have implemented Swift, a system that implements these various optimizations. Here, we present a proof-of-concept evaluation of Swift that focuses on the optimization of using multiple segments, as this single optimization shows the underlying flexibility that Swift offers. For example, the multiplexed segments could correspond to different protocols (e.g., one segment uses HOP while the other uses TCP), different underlying technologies (e.g., one bluetooth based segment and another 802.11 based segment), and different ISPs/service providers. We have used multiplexing of segments in the following ways: to aggregate AP uplink bandwidth, to aggregate throughput of multiple interfaces, for efficient hand-offs and to to mask failures in highly volatile scenarios (see [19, 17] for detailed experiments and results). As a proof-of-concept, we present results corresponding to the last use-case.

We consider communication between a vehicle and multiple TAPs and how use of multiple segments can mask short disruptions that are difficult to handle if we use a single segment for end-to-end communication. We use link level traces from a real world testbed at Microsoft's campus in Redmond [4]. We pick the two APs in the MS testbed with the best connectivity with the van and emulate their wireless channels. For the experiments, we use an Emulab style testbed that also offers a wireless emulator [1]. The van downloads a 10MB file from a server located over the Internet and having a 60ms RTT with the APs. We compare the performance of Swift with multiple segments (`Swift-mult`) with the scenario where only a single segment is used (`Swift-single`). We make 20 requests for each scenario and start times for the requests are randomly selected.

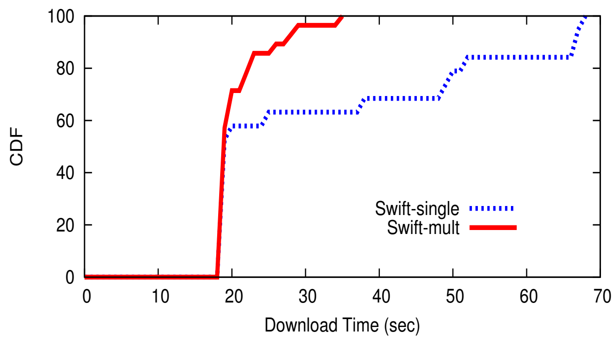


Figure 5: File Download Times in a Vehicular Communication Scenario. Use of multiple segments can mask disruptions that are common in such environments.

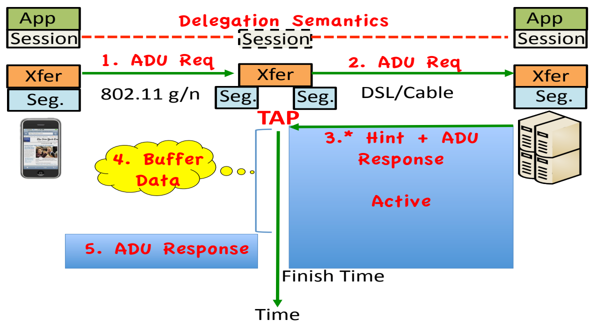


Figure 6: Overview of Catnap.

Figure 5 shows the cdf of the download times achieved in the two scenarios. Around 50% of the transfers do not experience any difference: these transfers are made during periods of good connectivity where there is no need to switch to the alternate segment. However, transfers that occur during bad periods experience severe performance degradation in the case of `Swift-single` whereas with `Swift-mult` switching to the alternate segment (if it is available at that time) significantly mitigates the performance degradation.

4.3 Energy Savings through Catnap

Catnap allows a mobile client to sleep *during* ADU transfers by intelligently shaping data before sending it over the wireless segment. Catnap targets settings where the wireless segment is much faster compared to the wired segment; in such scenarios, Catnap allows the wireless segment to remain inactive most of the time while still ensuring that the transfer finishes on time (Figure 6). During the time the wireless segment is inactive, the mobile device can enter into various sleep modes (e.g., 802.11 Power Save Mode, Suspend-to-RAM mode, etc), thereby providing significant energy savings to the mobile client. Catnap can provide up to 2-5x battery life improvement for real mobile devices under certain conditions [20].

Catnap can be viewed as a transfer service that leverages the key concepts of Tapa in the following ways. First, decoupling of segments, through use of different segment protocols and buffering at the TAPs, allows the wireless segment to remain inactive even though data continue to flow on the wired segment. This is not possible with end-to-end TCP as it relies on strict synchronization between end-points through the use of end-to-end acknowledgments. Second, the concept of ADU and associated hints (i.e., length in this case) allows the TAP to know when data will be *consumed* by the client application, so the initial packets of an ADU can be delayed as long as the finish time of the ADU remains the same.

If we consider Catnap in the opposite direction i.e., to upload data from the client to server, then we can see how richer reliability semantics at the session layer can be used to *delegate* transfer responsibility to the TAP. For example, in order to upload a large file to a server, the mobile client can burst the data to the TAP using the fast wireless segment, go offline, and the TAP can take over the responsibility of sending data to the server over the slow wired link.

5. PROTOTYPE EVALUATION

In this section, we present our experience of using various segment protocols and legacy applications with Tapa. We also present micro-benchmarks to quantify the overhead of using Tapa in various scenarios.

5.1 Supporting Legacy Applications

It is important to consider the effort required in developing applications that can leverage Tapa. We focus on modifying *existing applications* – which were not developed with Tapa in mind – as they represent the hard case. We modified the open source `Mozilla Firefox` browser to make use of Tapa API instead of sockets. Specifically, we created a Tapa stub that provided a socket-like interface to the applications and was used by the browser. This greatly simplified the browser modification process once the code was *separated* from sockets. Although there was a general separation of socket communication code and application logic, some of the browser optimizations violated this separation and made the modification process non-trivial. However, the modification effort required is still manageable. Our experiences, as well as earlier similar efforts [37], suggest that typically it is in the order of *1-2 weeks* for reasonably sized applications. Overall, despite the huge code base of the browser the changes made to the browser code were small and required approximately 200 lines of code (LOC)).

5.2 Supporting Diverse Segment Protocols

We now discuss our experience in adding different segment protocols to Tapa – as we expect customized segment protocols for different types of networks, it should be easy to implement and use these protocols within Tapa. In addition to TCP/IP, our prototype supports the following segment protocols:

HOP: HOP is a possible replacement for TCP in multi-hop mesh networks and environments involving mobility and disruption [30]. It runs between a client and a mesh gateway and expects some kind of decoupling at the gateway, so that a TCP-like protocol can work on the wired side for end-to-end transfers. We added a light weight stub (*50 LOC*) that removed the differences between the HOP API and the interface that Tapa expects segment layers to implement. Overall, it took roughly *20 man hours* to fully add support for HOP as a segment layer for Tapa.

Blast: We have specifically designed a protocol for 802.11 based single-hop wireless networks where TCP features, like congestion control and ACK based reliability, are often an over-kill. Blast is built on top of UDP – it offers no congestion control, but provides light-weight reliability (in the form of NACKs) and flow control. The protocol was developed independently and later integrated with Tapa as a segment protocol, requiring approximately *10 man hours* for the integration effort.

Bluetooth: We also added support for Bluetooth RFCOMM transfer mode as a Tapa segment protocol. As this mode bypasses IP, it is an attractive option for small devices with limited capabilities who want to communicate over the Internet (through a TAP). The API exposed by the bluetooth library uses the socket API (unlike HOP). As a result it was straightforward to incorporate bluetooth as a segment protocol in Tapa, requiring approximately *5 man hours* and *20 LOC* for this task.

Performance: Table 3 shows performance of these different segment layers (with tcp on the wired side) in an end-to-end transfer. TCP, HOP, and Blast used WiFi on the Emulab testbed while the bluetooth (BT) experiment was on the emulator. The results show expected performance under the given conditions.

	TCP	HOP	Blast	BT
Xput	5.95Mbps	6.4Mbps	6.6Mbps	600kbps

Table 3: Download of a 10MB file with different segment protocols. End-to-end TCP throughput i.e., without Tapa, is roughly 5.9Mbps.

5.3 Overheads: Micro-benchmarks

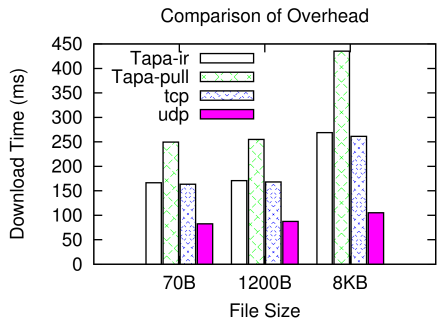
Tapa offers several optimization but not every application can benefit from them, so it is important to consider scenarios where it may hurt to use Tapa. We therefore conduct micro evaluation to evaluate the overhead of using Tapa under a scenario where no optimization is used.

5.3.1 Performance Overhead

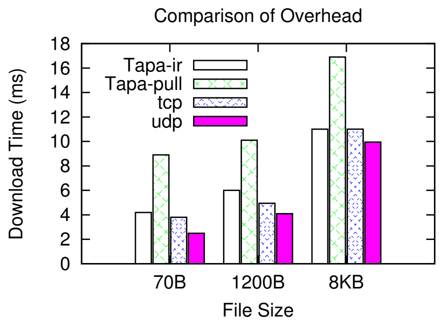
We consider a simple scenario involving a wireless client, TAP, and server. TAP and server are connected via a wired network – either LAN or WAN. We use TCP on both the wireless and wired segment. Also, even though Tapa allows reuse of segments which can eliminate connection set up delay, we disable this option for these experiments. This ensures a fair comparison with standard end-to-end tcp.

We compare the time required to complete a short request-response exchange in four different scenarios: `Tapa-ir` which refers to the mode where we push ADUs, `Tapa-pull` where we pull an ADU by first retrieving its id and then retrieving the data, `tcp` and `udp`. Figure 7(a) shows the results in a WAN setting with `80 ms` RTT. `tcp` and `Tapa-ir` perform the same whereas `Tapa-pull` requires more time because of the extra RTT involved in making a request for individual ADUs. Figure 7(b) shows that even in a LAN setting Tapa does not introduce any noticeable overhead and performs similar to the underlying segment protocols that it uses (TCP).

The above results for messages as small as 70 bytes show that the extra overhead introduced by Tapa in the form of ADUs and TAP is negligible. It also shows that `Tapa-ir` is a useful mode that can be used by interactive applications with short messages. This analysis suggests that over stable wireless links, Tapa will perform as well as today's protocol stack for a wide range of applications.



(a) WAN



(b) LAN

Figure 7: Tapa adds negligible overhead in both LAN and WAN scenarios. For transfers larger than 1MB (not shown) all scenarios have similar performance.

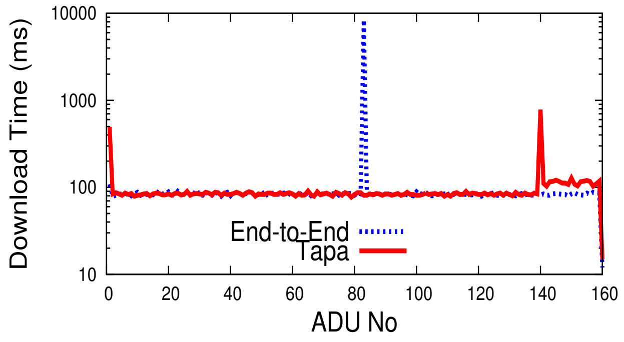


Figure 8: Case of TAP failure. Tapa recovers efficiently from TAP failures. Note log scale.

5.3.2 Reliability

As Tapa has to deal with ADU recovery in case of a TAP failure, we want to measure the overhead of this process. We use the LAN topology with tcp on both segments, and consider a scenario where the TAP fails after 15 sec and loses all its state. It comes up and starts serving the clients again after 5 seconds. This pattern is repeated and clients continue to make requests at random times for a 10MB file. We compare end-to-end tcp which naturally recovers from such AP failures with Tapa, where a TAP failure requires recovering the ADU that was being transmitted and all the ADUs that were yet to be transferred.

Figure 8 shows the download time for each of the 160 ADUs present in the file (note log scale). The spikes show the increased time that is required to download the affected ADU – during whose transfer the TAP failed. Note that as requests are made at random time we pick two typical runs (one each for Tapa and end-to-end tcp) that depicts how the recovery process works in both scenarios. In Tapa, the client discovers that the TAP is down (using absence of hello messages) and establishes another segment with an alternate TAP or waits for the old TAP to appear again. As the transfer service knows the ids of the missing ADU, it sends a new request to the server using the new TAP. As the graph shows, Tapa can recover efficiently from TAP failures.

6. RELATED WORK

Tapa's design is inspired by a large body of work, including the design of the original Internet, wireless and mobility related proposals, overlay networks, and proposals that deal with making middleboxes first-class citizens [12, 35, 9, 36, 18, 16]. We give a brief overview of the most relevant proposals, focusing on the key differences with Tapa.

Visible Middleboxes: Tapa shares the concern of earlier work that hidden middleboxes can be a source of problems, although we focus on "flow middleboxes" that carry transport state (e.g., proxies or other middleboxes that use different transport regimes for an end-to-end connection). This is different from proposals like NUTSS[25] and DOA[38] that deal with network level middleboxes, i.e., ensuring that middleboxes become part of routing and addressing and can therefore process packets (e.g., NATs, firewalls, etc).

The work that is most relevant is the proposal by Ford and Iyengar [24] who break-up the transport layer to accommodate flow middleboxes in the end-to-end path. Tapa provides a more general form of decoupling of segments, allowing use of non-IP protocols within a segment (e.g., XIP [26]). Another key difference is that Tapa uses ADUs and supports various ADU based services (e.g., pull mode of data retrieval).

Delay Tolerant Networks[23] provide a similar level of decoupling as Tapa, but their focus on arbitrary disruptions results in fundamental differences with Tapa. First, Tapa supports reasonably reliable, homogeneous segments whereas DTN regions provide either full reliability or no reliability. Second, DTNs only support push mode of transfers whereas the pull mode is an important mechanism in Tapa. Finally, DTNs do not have a notion of visible services that support new end-to-end semantics.

Overlays: Tapa is designed to be complementary to the Internet as it focuses on application/session semantics while Internet deals with network issues (reachability, forwarding, etc). As a result Tapa has very unique characteristics compared to traditional overlays. For example, Tapa's topology is highly constrained, so we do not need a complex routing protocol. Edge segments can be short-lived, which can result in very dynamic topology. Segments in Tapa mostly line up with network boundaries and as a result, Tapa segments may use very different technologies, e.g., IP on the wired segment and custom protocols on the wireless/data-center segment. Finally, the role of the two end-points (e.g., client and a server) is very asymmetric. All the differences require new mechanisms not found in other overlays.

Transfer Services and Data Oriented Proposals: Tapa's transfer service leverages several concepts used in DOT [37]. However, Tapa focuses on transfer services within the network (TAPs) as well as end-points. Also, in order to provide rich end-to-end semantics, our transfer service is implemented *below* an end-to-end session whereas DOT works on top of existing transport layer protocols.

Recently, Popa et al [33] propose the use of HTTP as the narrow waist of the Internet, so it can be viewed as a transfer service. While the use of HTTP is certainly easier to use in the short-term, it is difficult to exploit many content centric and multi-path optimizations due to the inherent limitations of HTTP (i.e., naming, rigid semantics, etc), Moreover, we also define the roles of layers below and above the transfer service i.e., segment and session layers, which play a key role in the Tapa architecture.

Future Internet Architectures: Tapa's pull mode is inspired by data oriented architectures (e.g., CCN [27]), but there are important differences as we discuss below. First, unlike CCN, Tapa is not a "pure" content-centric architecture, so host and destination identifers are present in the packets even if we are retrieving an ADU based on its identifer. This results in different per-hop router operations as well as different failure modes in Tapa compared to CCN. Second, CCN operates at a per-packet granularity while Tapa uses ADUs. This difference mandates the need for different mechanisms to support traditional transport functions like congestion control, reliability, and data reassembly.

A better positioning of Tapa compared to CCN and other new network architectures is to view it as a transport architecture that can leverage these proposals as part of its segment protocol. For example, if Tapa is implemented over content centric network architectures [27, 26, 29], or service centric architectures [32, 26] then Tapa's transfer service can leverage the inherent features provided by these architectures for content/service routing. This will simplify Tapa's transfer layer and can also potentially improve performance due to late binding, native support for content/service discovery, and intrinsic security [26].

7. FINAL THOUGHTS

We presented the design, implementation, and evaluation of Tapa, a transport architecture that accommodates network heterogeneity and rich in-network services. Tapa unbundles today's transport and makes explicit use of in-network services that operate on ADUs. Our practical experience, as well as the case studies in this paper, confirms that Tapa offers great flexibility at multiple levels: customized solutions as segment protocols; diverse data oriented optimizations at the transfer level; and services with new semantics at the session level.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers, XIA team members, Ratul Mahajan, Ihsan Qazi, and Dina Papagiannaki for providing useful feedback on this work. This research was funded in part by NSF under award number CNS-1040801.

9. REFERENCES

[1] CMU Wireless Emulator. www.cs.cmu.edu/ emulator/.

[2] Dynamic site acceleration. http://www.akamai.com/html/solutions/dsa_curriculum.html.

[3] Service Location Protocol. RFC 2608.

[4] Vanlan. research.microsoft.com/en-us/projects/vanlan/.

[5] A. Akella, et al. Self-management in chaotic wireless deployments. In *MobiCom '05*, pp. 185–199. 2005.

[6] M. Alizadeh, et al. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010*, pp. 63–74. 2010. ISBN 978-1-4503-0201-2.

[7] A. Anand, et al. Redundancy in network traffic: findings and implications. In *Proc. of SIGMETRICS*, pp. 37–48. 2009.

[8] D. G. Andersen, et al. Accountable Internet Protocol (AIP). In *SIGCOMM*. 2008.

[9] A. V. Bakre, B. Badrinath. Implementation and performance evaluation of indirect tcp. *IEEE Transactions on Computers*, 46(3):260–278, 1997.

[10] S. Biswas, R. Morris. Exor: opportunistic multi-hop routing for wireless networks. *SIGCOMM CCR*, 35(4):133–144, 2005.

[11] J. Border, et al. Performance enhancing proxies intended to mitigate link-related degradations, 2001.

[12] D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM '88*, pp. 106–114. 1988. ISBN 0-89791-279-9.

[13] D. D. Clark, D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*. 1990.

[14] D. D. Clark, et al. Addressing reality: an architectural response to real-world demands on the evolving internet. *SIGCOMM Comput. Commun. Rev.*, 33(4):247–257, 2003.

[15] D. D. Clark, et al. Making the world (of communications) a different place. *SIGCOMM CCR.*, 35(3):91–96, 2005.

[16] J. Crowcroft, et al. Plutarch: an argument for network pluralism. *SIGCOMM Comput. Commun. Rev.*, 33:258–266, August 2003. ISSN 0146-4833.

[17] F. R. Dogar. Architecting for diversity at the edge: Supporting rich network services over an unbundled transport. PhD thesis. 2012.

[18] F. R. Dogar, P. Steenkiste. M2: Using Visible Middleboxes to Serve Pro-active Mobile-Hosts. In *ACM SIGCOMM MobiArch '08*, pp. 85–90. 2008.

[19] F. R. Dogar, P. Steenkiste. Segment based internetworking to accommodate diversity at the edge. *Technical Report - CMU-CS-10-104*, 2010.

[20] F. R. Dogar, P. Steenkiste, K. Papagiannaki. Catnap: Exploiting high bandwidth wireless interfaces to save energy for mobile devices. In *ACM MobiSys*, pp. 107–122. 2010.

[21] F. R. Dogar, et al. Ditto: a system for opportunistic caching in multi-hop wireless networks. In *ACM MobiCom*. 2008.

[22] J. Eriksson, H. Balakrishnan, S. Madden. Cabernet: vehicular content delivery using wifi. In *MobiCom '08*, pp. 199–210. 2008.

[23] K. Fall. A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03*, pp. 27–34. 2003.

[24] B. Ford, J. Iyengar. Breaking up the transport logjam. In *ACM Hotnets*. 2008.

[25] S. Guha, P. Francis. An end-middle-end approach to connection establishment. In *SIGCOMM*. 2007.

[26] D. Han, et al. XIA: Efficient support for evolvable internetworking. In *Proc. 9th USENIX NSDI*. Apr. 2012.

[27] V. Jacobson, et al. Networking named content. In *CoNEXT '09*, pp. 1–12. 2009.

[28] S. Kandula, et al. FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput. In *NSDI*. April 2008.

[29] T. Koponen, et al. A data-oriented (and beyond) network architecture. In *SIGCOMM '07*, pp. 181–192. 2007.

[30] M. Li, et al. Block-switched networks: a new paradigm for wireless transport. In *NSDI'09*, pp. 423–436. 2009.

[31] X. Liu, X. Yang, Y. Xia. Netfence: preventing internet denial of service from inside out. In *ACM SIGCOMM 2010*.

[32] E. Nordstrom, et al. Serval: An end-host stack for service-centric networking. In *Proc. 9th USENIX NSDI*. April 2012.

[33] L. Popa, A. Ghodsi, I. Stoica. HTTP as the narrow waist of the future Internet. In *Hotnets 2010*.

[34] S. Roy, et al. Application level hand-off support for mobile media transcoding sessions. In *NOSSDAV '02*, pp. 95–104. 2002.

[35] J. H. Saltzer, D. P. Reed, D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 1984.

[36] I. Stoica, et al. Internet indirection infrastructure. *SIGCOMM Comput. Commun. Rev.*, 32(4):73–86, 2002.

[37] N. Tolia, et al. An architecture for internet data transfer. In *NSDI '06*.

[38] M. Walfish, et al. Middleboxes no longer considered harmful. *OSDI*, pp. 215–230, 2004.