Swift: The Missing Link Between Peer-to-Peer and Information-Centric Networks

Flutra Osmani
KTH Royal Institute of Technology
Forum 105, SE-164 40
Kista, Sweden
flutrao@kth.se

Victor Grishchenko*
Citrea
Ekaterinburg, Russia
victor.grishchenko@gmail.com

Raul Jimenez, Björn Knutsson
KTH Royal Institute of Technology
Forum 105, SE-164 40
Kista, Sweden
{rauljc, bkn}@kth.se

ABSTRACT

A common pitfall of many proposals on new information-centric architectures for the Internet is the imbalance of up-front costs and immediate benefits. If properly designed and deployed, information-centric architectures can accommodate the current Internet usage which is at odds with the historical design of the Internet infrastructure. To address this concern, we focus on prospects of incremental adoption of this paradigm by introducing a peer-to-peer based transport protocol for content dissemination named *Swift* that exhibits properties required in an Information-Centric Network (ICN), yet can be deployed in the existing Internet infrastructure. Our design integrates components while highly prioritizing modularity and sketches a path for piecemeal adoption which we consider a critical enabler of any progress in the field.

Keywords

Distributed systems, networking, transport protocol

1. INTRODUCTION

In this paper, we introduce *Swift* protocol. *Swift* was originally designed to be a replacement for the BitTorrent protocol and inherits some of the characteristics that have made BitTorrent successful, but was not intentionally designed according to the principles of information-centric networking (ICN) paradigm. It is, thus, until now a product of (unintentional) evolution towards ICN that we now seek to direct and accelerate, while retaining all the properties that make it work well on top of the existing network.

We find the ICN concept to be increasingly reflected in both the way Internet is being used and in how Internet-based services are being implemented today. In many cases, we find that the problems we struggle with in the current incarnation of the Internet are those that ICN design proposals seek to address.

For example, over 90% of today's Internet bandwidth [4] is effectively devoted to disseminating static multimedia content. In order to do so to an increasingly large and geographically diverse audience, various approaches are used. For example, for web-based content, Content Delivery Networks (CDNs) like Akamai use modified DNS servers that generate responses based on the topological/geographical location of the requester. These "tricks" are there to achieve on the current Internet the features that are at the core of ICN.

During the past years, a chain of new network architectures based on the information-centric paradigm (also content-centric, name-oriented, named-data) have been proposed, including CCN [17], DONA [20], NetInf [11], secure naming by Wong et al [23], and content-centric router by Arianfar et al [6] to address the limitations of IP, namely the inability to decouple data from storage, inefficient data dissemination, lack of support for middleboxes, ubiquitous availability of data, and security.

The historical conversation-centric end-to-end model, embodied in the TCP/IP stack, is based on message exchange between pairs of peers, typically *servers* and *clients*. On the other hand, ICN is a paradigm in which focus shifts away from the mechanics of moving bits between peers (end-hosts). Instead, the focus in on the information itself, and the underlying network only a conduit for the information. In a sense, named-data network breaks with the end-to-end abstraction, as there are no *ends* and the entire network is considered a *cloud*, which both stores and serves data.

Similarly, we can see in the evolution of peer-to-peer (P2P) file-sharing technologies how they have adopted ways of managing content that more and more look like ICN. While BitTorrent [10] has always used a SHA-1 hash of the content data to identify that unique content item, it used to be that you also needed a location identifier (the address of the tracker through which the peers hosting the content can be located). However, the current incarnation of BitTorrent instead uses a shared global Distributed Hash Table (DHT) to locate peers using the aforementioned content hash as the key.

The historical Usenet discussion system [5] had all the key information-centric features: logical namespace and unique message identifiers, flood message propagation and caching. The git [3] revision control system represents version history

*Work by Victor Grishchenko was carried out while at the Technical University of Delft

of a project as a directed acyclic graph of revisions, where every revision is identified with SHA-1 hash of its contents; repositories push and pull content, thus forming a network of arbitrary topology.

This tendency towards information-centricity implies a strong demand for a generic named-data substrate that is not reflected yet in the *de jure* network architecture. Given this dissonance, some have proposed a *networking revolution* to dethrone the Internet Protocol (IP) in favor of a clean-slate redesign of the networking infrastructure.

While intellectually attractive, we do not consider such an approach realistic. Not only because it would require the expensive and disruptive wholesale replacement of the existing infrastructure, but more importantly because such a migration is unlikely to happen before the wholesale conversion of applications to information-centric analogues of the current application ecosystems, and that conversion is unlikely to happen until the required infrastructure is in place.

Having made this observation, it seems clear that evolution, not revolution, is the best way towards ICN, and by recognizing and helping this along, we can both make ICN happen sooner and ensure that the ICN approach will be one tested in both lab and real-world settings, and hence "the fittest".

We start with the necessary basic properties of any information centric architecture and determine which of them *Swift* already supports. Further, we determine which information centric primitives *Swift* does not provide and address them by leveraging existing technologies, such as DHTs to find peers and standard IP to route packets.

In this paper, we argue that our modular design addresses the gap between Internet usage and the underlying network, without requiring clean-slate redesigning of the architecture. In particular, *Swift* protocol supports most properties proposed in information-centric architectures like CCN [17], DONA [20], and NetInf [11]. First, *Swift* uses names — flat identifiers — to request content instead of end-point addresses; in addition, it segments named objects in uniquely identified chunks. Second, *Swift* employs per-packet integrity check, enabling any peer in the network to cache and relay content and verify the integrity of each piece. Third, *Swift* avoids transmitting additional metadata and is suitable for live/mutable data, by employing Merkle hashes [21].

Moreover, *Swift* is a receiver-driven chunk-level transport protocol; the receiver may send concurrent requests for chunks to multiple peers in the network in order to enhance its content retrieval rate. To efficiently exploit available bandwidth, *Swift* employs a delay-based congestion control algorithm named LEDBAT [22], and to address the issue of middleboxes *Swift* employs a NAT hole punching mechanism. For peer discovery, *Swift* can use centralized trackers or DHTs; in Section 7 we explain how Mainline DHT (MDHT) can be used to find peers offering the given data object in sub-second time periods.

The paper proceeds as follows. In Section 2 we introduce ICN related work and summarize system description. Section 3 discusses design properties and the resulting separation of transport and internetworking layers. Section 4 describes our variation of the Merkle hashing scheme and its extensions. In Section 5 we introduce a vocabulary of messages that constitutes our protocol. Section 6 describes our UDP-based implementation. Section 7 discusses the implications for peer discovery and packet routing. Section 8 concludes.

2. BACKGROUND

Most proposals on ICN architectures — evolutionary and clean-slate designs — aim to define the main building blocks of an information-centric network. In the CCN [17] design, content names have a hierarchical structure and are constructed according to the standard URI form. Content is requested using an *interest* packet which contains the name of the content. Every content router receiving the *interest* packet checks if the given packet is in its local cache and thus returns a corresponding *data* packet along the reverse path, otherwise, it forwards the *interest* to the correct interface using longest prefix matching. A similar approach for content retrieval is reflected in the PSIRP architecture [2], albeit it uses flat instead of hierarchical names to address content.

Some recent ICN projects adopt flat, self-certifying, labels to name content. Initially employed in DONA's design [20], flat names are used by the route-by-name protocol (devised on top of the IP layer) to request content. Similarly, efforts by Dannewitz et al [11] and Wong et al [23] explore secure naming schemes to ensure the data is persistent and not accessed by unauthorized users.

Self-certifying (flat) names have been criticized for their lack of scalability (cannot be aggregated), flexibility, and lack of security during the translation of flat names to human-readable names. However, recent work by Ghodsi et al [13] argues that self-certifying names exhibit better security properties than human-readable names because 1) they can handle better denial-of-service attacks — the network knows the binding between the name and the key thus it can verify that a given object is associated with a given name, and 2) may scale better through *explicit aggregation* — using concatenations of the form $A.B.C$, where each letter is a name itself.

Despite differences in the naming scheme, the necessary mainstay of any name-oriented network architecture is to employ either cryptographic hashes or signatures in order to enable indiscriminate caching of data in the network and the possibility of its retrieval from any available peer. In *Swift*, we employ hashes — Merkle hashes — and argue that they are *sufficient* to perform any transport function. More specifically, Merkle hash trees [21] allow to identify and verify data, thus enabling any peer in the network to request, relay and store data. Furthermore, our variant of hash trees needs no supplementary transfer metadata, rendering transport into a thinner *layer* than usual.

Swift must be implemented by all inter-operating network peers, and its functioning involves cross-layer relaying of the data, known as *internetworking* (see Figure 1). Hence, the required functionality needs to be as simple and formalized as possible. It follows naturally that any rich semantic data names or transfer metadata is unnecessary and should therefore not be part of transport.

We define a natural separation of *Swift* from the upper *naming* part which deals with problems inherently semantic and the lower *internetworking* layer. Leveraging existing deployed routing infrastructure and a simple hash-based naming mechanism, *Swift* retrieves pieces of content requested by the receiver from peers in the network — functionality that researchers propose to incorporate in any transport protocol

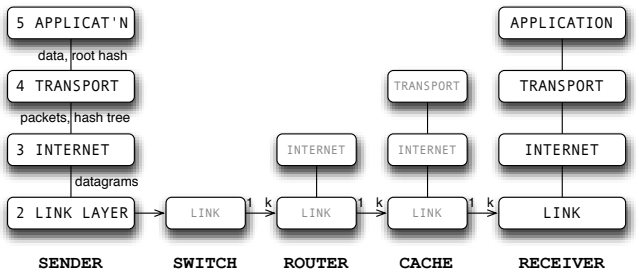

Figure 1: Swift protocol stack: application-dependent queries are translated into requests to the transport layer for particular data — identified with hashes. Transport deals with data streams, their verification and storage.

for information-centric networks [9, 7]. The naming layer is out of scope, thus we make no specific assumptions.

3. DESIGN OVERVIEW

In our design, content is identified by a single cryptographic hash that is the *root hash* in a Merkle hash tree, calculated recursively from the content (see details in the Merkle hash extension document [8]). The ability to verify data against its name allows for storage in the network and retrieval of data from an arbitrary location. Second, as a (packet) network may need to check data integrity piece by piece, possible options boil down to either per-packet signatures, as in CCN, or Merkle hash trees. Differently from signatures, Merkle hash trees provide strict permanent identifiers of static data pieces, so we chose them as the foundation, later extending the approach to dynamic data (see Section 4).

The hashing scheme enables the entire information-centric stack, illustrated in Figure 1, in two ways. First, it allows for a perfect application-to-transport handover. Semantically-rich and application-dependent queries are eventually converted into requests to the transport layer for particular data pieces, precisely identified with hashes. Second, hashing enables information-centric internetworking, i.e. identification and relay of data pieces, data verification, and storage in the network.

As depicted in Figure 1, *Swift* embeds a layer separation scheme very much reminiscent of TCP/IP. Namely, there is a relay internetworking layer that only deals with separate datagrams. On top of it, there is a somewhat more intelligent transport layer that deals with entire data streams, performing verification, caching, and storage.

Any peer running *Swift* may cache content. Technically, there is no difference between a *peer* and a *cache* — they run the same protocol; the conceptual difference lies in the intention: a cache "stores" content to further disseminate it but is not particularly interested in the given content. The caches may be regular peers or peers put in place by ISPs — who are interested in replicating "popular" content within their administrative domains and thus avoid transit traffic and costs to external domains. If operated by ISPs, such caches (interchangeably, peers) may manage the content they offer according to some basic rules, such as LRU or demand.

Discovering peers or caches may be done centrally through trackers or ISP-based trackers, or in a decentralized fashion through PEX or DHTs. We explain how discovering new peers can be done with Peer Exchange in Section 6 or with Mainline DHT in Section 7.

In *Swift*, data storage and data verification are highly interdependent and important in terms of security. For example, if a caching peer does not verify data integrity, it makes *cache poisoning* possible. While a final recipient does not accept (drops) incorrect data, an erroneous cache may form a *clot* in the network, preventing the correct data from passing through. Similarly, data verification requires storage in peers to some degree, as Merkle hash trees need accompanying uncle hash chains to be available in order to verify data pieces.

In a sense, hashes replace IP addresses as end-point identifiers. A receiver uses a root hash to "open" the connection to the network and retrieve the data. The receiver requests specific pieces of data using a novel method called *bin numbers* (see details in the RFC document [14]) which allows the addressing of a binary interval of data using a single integer. This numbering mechanism reduces the amount of state that needs to be stored in each peer and minimizes the space required to denote intervals on the wire. Because the receiver directly addresses the data instead of a single end-point at a particular IP location, it has no control over which peer (replica) will respond; the receiver controls the reception of pieces based on local parameters.

4. THE HASHING SCHEME

We modified Merkle hash trees and focused on smooth operation of both vertical (application to transport) and horizontal (internetworking) handovers to ensure that no peer requires third parties to verify bindings between keys and names (as in CCN [17]) or to retrieve additional metadata to perform their function. We ensure their operation is as simple and formalized, as possible. In Sec. 4.3, we extend our basic technique to the cases of live data streams and versioned data.

4.1 64-bit Merkle Trees

We developed a variant of the Merkle hash tree scheme [21] to satisfy three key requirements: (a) per-packet data integrity checks, (b) no additional metadata and (c) suitability for live/mutable data. The general concept is to start with the root hash only, then incrementally acquire data and hashes, while verifying every single step.

First, content is divided into 1KiB chunks named *packets*, except for the tail packet, which may have less than 1KiB of data. A cryptographic hash, such as SHA1, is then calculated on every packet. Second, a hash tree is defined over the complete $[0, 2^{63})$ byte range, which we consider to be a good approximation of infinity in relation to content size. The tree consists of aligned binary intervals called *bins*, i.e. $[i2^k, (i+1)2^k)$. Bins are nested, forming a strict binary tree (see Figure 2). Each tree contains 2^{64} bins of different sizes, including one void and one root bin; the base — the lowest level — of the tree is composed of 2^{10} byte long bins.

The base of the tree (the leaves) accommodates all the data chunks, starting from the left-most leaf. Normally, the base of the tree is wider than the number of chunks, thus the remaining empty leaves in the tree are assigned hash values of zero. In higher levels of the tree (above base), bins contain hashes which are calculated as a SHA1 hash of a concatenation of two — left and right — child (lower-level) hashes. This hashing process iterates until a hash value for

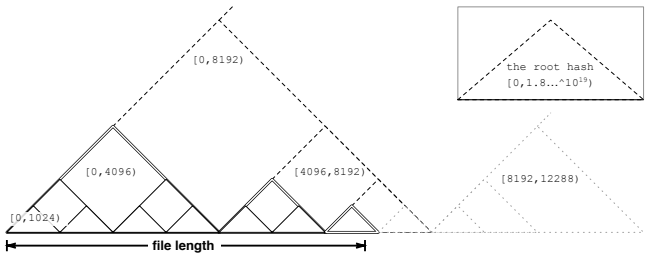

Figure 2: Merkle hash tree constructed for a file of size less than 8KiB. Bins for peak hashes are marked with double lines and they cover the ranges $[0, 4096)$**,** $[4096, 6144)$ **and** $[6144, 7162)$**. Filled bins are marked with solid lines, incomplete bins with dashed lines, and empty bins with dotted lines.**

the root bin is calculated, known as the *root hash*.

Figure 2 illustrates an example where the file size is less than 8KiB long. Its $[8192, 12288)$ empty bin has zero hash by definition, as do the rest of empty bins outside the $[0, 8192)$ range. The root hash covers the entire $[0, 2^{63})$ range; this approach gives us a fixed point of reference when growing the hash tree down from the root.

4.2 Peak Hashes

The concept of peak hashes enables two cornerstone features: file size proving and unified processing of static data and live streams. In addition, they help avoid the usage of additional transmission metadata. Formally, *peak hashes* are hashes defined over filled bins, whose parent hashes are defined over incomplete (not filled) bins. A filled bin is a bin which does not extend past the end of the file, or, more precisely, contains no empty packets.

Practically, we use peaks to cover the data range with a logarithmic number of hashes, so each hash is defined over a "round" aligned 2^k interval. As an example, suppose a file is $l = 7162$ bytes long (see Figure 2). That fits into seven packets ($\frac{7162}{1024} < 7$), the tail packet being 1018 bytes long. For this particular file we will have three peaks, covering $[0, 4096)$, $[4096, 6144)$ and $[6144, 7162)$ ranges (triangles depicted with double lines). The last range might also be written as $[6144, 8192)$ because we round-up to 1KiB packet size.

The number of peak hashes can not exceed $\lceil \log_2 \frac{l}{1024} \rceil$. Practically, peak hashes provide us with more convenient "reference roots", as compared to the root hash which is 53 levels higher than the packets. More importantly, peak hashes allow a sender to quickly *prove* the file size to a recipient who only knows the root hash; otherwise, file size would have to be supplied as a separate metadata piece and thus separately verified, showing up in the protocol and in the interfaces.

4.3 Live Data Streams

In the case of live data streams, the root hash is undefined or, more precisely, transient, as long as new data keeps coming, filling new packets to the right. Hence a transfer has to be identified with a public key instead of a root hash. Keys are more difficult to deal with than hashes, as they have more degrees of freedom. For example, once a key is compromised, any party may rewrite a pre-existing stream. Also, while a hash might be derived directly from the data, a signature can only be verified once known.

Because of such issues, we try to minimize key/signature usage by using the same peak hashes scheme as in the case of static data. Indeed, once a peak hash is defined, it never changes. Thus, we only need a logarithmic number of signatures to sign peak hashes. After that, we may deal with the same Merkle hash tree as before.

Signing the peak hashes only requires the sender to issue the newly formed peak hashes with their signatures attached. On the receiver side, the recipient will only have to check the signature of a new peak hash and whether it matches its child hashes. Such a calculation is incremental and local. Otherwise, if the root hash were to be signed instead, this would require constant re-verification of all the encompassing peak hashes.

Until this point, we assumed that the sender emits data in "round" 1KiB long packets. What if smaller portions of data need to be committed to the network? We do not equal, but we strongly associate our "packets" with link-layer "frames". Thus, once data is worth sending, before it fills a packet, then it also needs a hash and a signature.

5. MESSAGE VOCABULARY

In this section we describe the set of messages that constitute *Swift* protocol. In this section, we refer to it as a *vocabulary* which is instantiated as a transport protocol (as in Figure 1). No particular serialization, encapsulation schemes, or message exchange patterns are specified, beyond the very basic requirements.

A $DATA$ message simply carries pieces of data. A $DATA$ message must carry a bin of data. Specifically, each $DATA$ message contains the bin number of the piece and the piece itself. This way, uniform pieces or multiples of pieces can be processed, making it easier to check the data hash tree at once.

A $HASH$ message carries the necessary hashes that the receiver needs in order to verify the integrity of a piece. We employ the principle of *atomic* datagrams, which means that every piece of data must be verified once received and accepted, otherwise dropped. At this point, the sender must make sure the receiver has every hash needed to verify the incoming data immediately. Finally, it is possible to supply the recipient with parts of the hash tree incrementally, to allow for an amortized and local verification of data.

As we allow for the possibility of data retrieval from multiple peers in parallel, the vocabulary employs $HINT$ and $HAVE$ messages. A $HINT$ (request) message indicates which pieces of data a receiver wants to retrieve, while a $HAVE$ message conveys what pieces of data a sender has available. On incoming data, a receiver uses ACK messages to acknowledge the received pieces; acknowledgements follow the logic of hash trees, which means that data must be acknowledged in bins as well.

We define *channels* as a means to identify ongoing transfers, where each transfer is identified by either a hash or a public key. Channel identifiers are conveyed through the datagram headers.

6. THE UDP IMPLEMENTATION

As previously stated, *Swift* [1] protocol is implemented over UDP; the detailed design is described in an IETF draft [14] and an overview is outlined in a technical report [15]. The

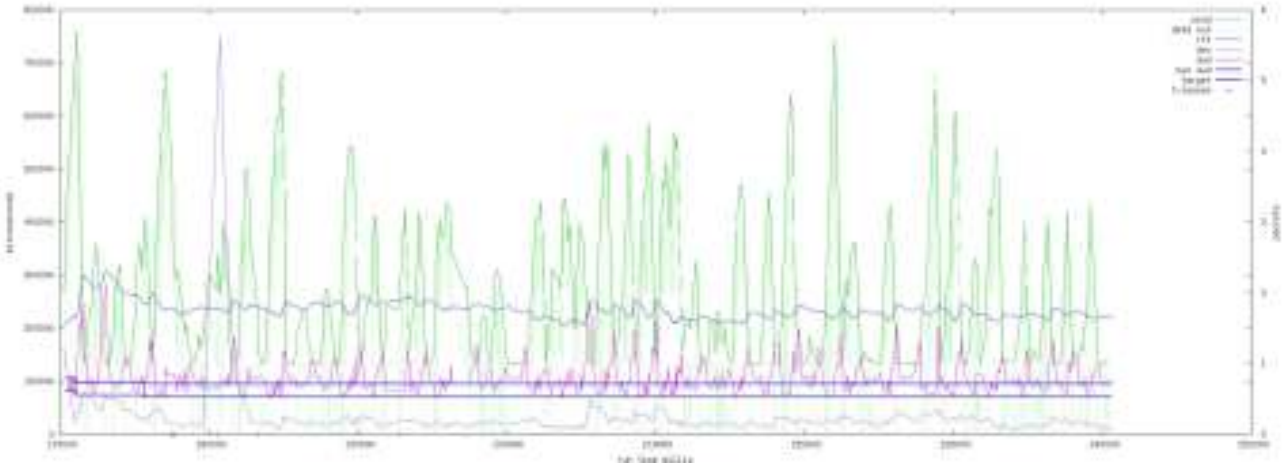

Figure 3: LEDBAT congestion control adjusts the congestion window *cwnd* in order to avoid data losses, by comparing estimated queuing delay against a fixed target delay. The figure depicts experimental results when two nodes exchange pieces of content.

protocol is a direct implementation of the vocabulary (see Section 5), with some additions and extensions.

Messages are serialized as fixed-width fields starting with a single-byte message type field, followed by fixed-width payload fields, such as bin numbers, data, hashes and such. Messages are packed into UDP datagrams. Datagram processing is event-driven, fully implementing the atomic datagram concept. This means that every datagram is either immediately committed to storage or immediately dropped; there are no buffer-re-assembly mechanics.

The UDP implementation employs LEDBAT [22] congestion control algorithm, which allows streams to run virtually lossless under normal conditions. LEDBAT is a delay-based congestion control algorithm which increases/decreases the congestion window based on the estimated queuing delay. It uses an increased queuing delay as indicator of congestion and thus immediately reacts by backing off (decreasing the rate).

Queuing delay in LEDBAT is known as the *one-way delay* (label *owd* in Figure 3) and it is calculated as the difference of timestamped packets between the sender and the receiver. The receiver also maintains a minimum over all one-way delays — *base delay* — which indicates the amount of delay due to queuing. LEDBAT compares this estimated queuing delay against a fixed target delay value (line *target* in Figure 3); the difference determines if the congestion window should be increased or decreased.

Figure 3 depicts how LEDBAT predicts congestion and avoids data losses for a given exchange between two peers (the figure only illustrates a preliminary test performed in a controlled environment with several peers spread across continents). In the testing scenario, one peer acts as a content provider — has the whole content — and other peers (requesters) are interested in the given content. The requesters retrieve the pieces, initially from the only content provider, and later on, they continue retrieving pieces from the participating requesters — who already obtained some pieces of the desired content.

Furthermore, *Swift* implements a unified mechanism of PEX and NAT hole punching functionality [12]. It uses two types of PEX messages — PEX_REQ and PEX_ADD — to retrieve/exchange addresses among the peers, in a gossip fashion. However, PEX messages are transmitted in such a way that they facilitate the communication between peers that are located behind middleboxes: once a peer A introduces peer B to C, it should — within a period of 2 seconds — introduce peer C to B. This mechanism makes *Swift* agnostic to middleboxes.

To guarantee that a receiver can verify every packet, the sender has to prepend it with the missing hashes. In a network with no data loss, the receiver builds the hash tree incrementally, thus every packet of data needs one hash on average. More precisely, every *even* packet needs a hash for its sibling, every fourth also needs a hash for its uncle, every eighth also needs a hash for its parent's uncle, and so forth, thus the average is 1. In practice, some packets are lost, so a prudent sender over-provisions hashes to compensate for possible loss. Thus, the actual traffic overhead of hashes is somewhat above the perfect value of 2% (assuming 20 byte hashes for a 1024 byte packet).

The protocol needs to keep more state on the transfer progress, as data might arrive out of sequence — mostly because data is delivered from different peers in parallel. The *state* must also be communicated over the wire, using unreliable datagrams. We adopted a generic compressed-bitmap data structure named *binmaps* [16], a hybrid of bitmap and a binary tree, which allows to track data at an arbitrary scale, starting from a single packet. Data is requested and acknowledged in bins; this provides the necessary compression and redundancy as continuous data pieces are acknowledged with a logarithmic number of messages.

7. ROUTING AND TRACKING

In order to retrieve data associated to a *root hash*, *Swift* needs to discover peers. This *peer discovery* process is performed by requesting peers from a *tracker*.

Trackers are used in peer-to-peer systems to keep track of peers sharing a given piece of content. The tracker's interface is simple. Peers can request a list of peers for a given content identifier (a *root hash* in *Swift*, an *info hash*

in BitTorrent). Peers also register itself in the tracker to be discovered by others.

Swift can use any tracking mechanism regardless of its particular implementation. Tracker mechanisms used in BitTorrent are prime candidates to be used due to their proven merits on large-scale deployments, but other implementations offering equivalent functionality may be used [24].

BitTorrent's trackers can be centralized or DHT-based. In the first case, the URI of the tracker tracking a given piece of content is necessary. The DHT-based option, on the other hand, forms a global tracking system where all content is tracked, thus no tracker URI is needed.

We favor the DHT-based tracker mechanism due to the scalability of the DHT and the minimization of metadata for *Swift* (no tracker URI is needed, just a *root hash*) to retrieve the data. Scalability is well illustrated by Mainline DHT, the BitTorrent's largest DHT-based tracker on the Internet. Mainline DHT is supported by most of the popular BitTorrent clients, forming a DHT overlay of between 6 and 11 million nodes[19][1]. It is difficult to estimate how many pieces of content Mainline DHT tracks at a given time, but given the size of the BitTorrent ecosystem, even conservative estimations would yield six-digit numbers.

Furthermore, recent measurements [18] have shown that Mainline DHT's response time is consistently low, which makes it suitable for latency-sensitive applications such as on-demand video streaming.

8. CONCLUSION

In this paper, we presented a peer-to-peer based transport protocol for content dissemination named *Swift* and argued that the protocol exhibits ICN properties that help close the gap between the way Internet applications are used today and the underlying infrastructure supporting such applications. Further, we explored ways *Swift* may embed additional ICN properties in its behavior by leveraging existing technologies and infrastructure, such as decentralized peer discovery mechanisms and standard IP routing.

9. ACKNOWLEDGMENTS

We would like to thank Arno Bakker and Pehr Söderman for providing us with valuable feedback. The research leading to these results has received funding from the Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 216217 (P2P-Next).

10. REFERENCES

[1] libswift Homepage. `http://libswift.org`.

[2] PSIRP Project. `http://www.psirp.org`.

[3] The git source code management system. `http://git-scm.com`.

[4] Cisco Visual Networking Index: Usage Study. `http://cisco.com`, October 2010.

[5] R. Allbery and C. Lindsey. RFC 5537: Netnews Architecture and Protocols.

[6] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. In *ReARCH'10*.

[7] S. Arianfar, J. Ott, L. Eggert, P. Nikander, and W. Wong. A Transport Protocol for Content-Centric Networks. In *ICNP'10. Poster Session*.

[8] A. Bakker. Merkle hash torrent extension, BEP 30. `http://bittorrent.org/beps/bep_0030.html`.

[9] G. Carofiglio, M. Gallo, and L. Muscariello. Bandwidth and storage sharing performance in information centric networking. In *SIGCOMM ICN'11*.

[10] B. Cohen. Incentives Build Robustness in BitTorrent, 2003.

[11] C. Dannewitz, J. Golic, B. Ohlman, and B. Ahlgren. Secure Naming for a Network of Information. In *INFOCOM'10*.

[12] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-Peer Communication Across Network Address Translators. `http://www.brynosaurus.com/pub/net/p2pnat/`.

[13] A. Ghodsi, T. Koponen, J. Rajahalme, P. Sarolahti, and S. Shenker. Naming in Content-Oriented Architectures. In *SIGCOMM ICN'11*.

[14] V. Grishchenko. The Generic Multiparty Transport Protocol (swift). `draft-grishchenko-ppsp-swift-03.txt`.

[15] V. Grishchenko, F. Osmani, R. Jimenez, J. Pouwelse, and H. Sips. On the Design of a Practical Information-Centric Transport. PDS Technical Report PDS-2011-006.

[16] V. Grishchenko and J. Pouwelse. Binmaps: hybridizing bitmaps and binary trees. PDS Technical Report PDS-2011-001.

[17] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *CoNEXT '09*.

[18] R. Jimenez, F. Osmani, and B. Knutsson. Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay. In *11th International Conference on Peer-to-Peer Computing 2011*, Kyoto, Japan, 8 2011.

[19] K. Junemann, P. Andelfinger, J. Dinger, and H. Hartenstein. BitMON: A Tool for Automated Monitoring of the BitTorrent DHT. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1–2. IEEE, 2010.

[20] 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 *SIGCOMM '07*.

[21] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO*, 1987.

[22] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). `draft-ietf-ledbat-congestion-04.txt`.

[23] W. Wong and P. Nikander. Secure naming in information-centric networks. In *ReARCH '10*.

[24] L. Xiao, D. A. Bryan, Y. Gu, and X. Tai. A PPSP Tracker Usage for Reload. `draft-xiao-ppsp-reload-distributed-tracker-03.txt`.

[1] Real-time estimation available at `http://dsn.tm.uni-karlsruhe.de/english/2936.php`