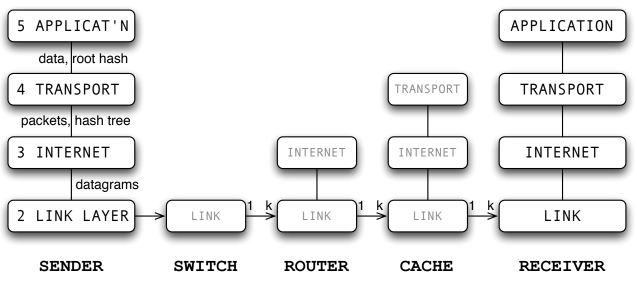

Figure 1: Swift protocol stack: application-dependent queries are translated into requests to the transport layer for particular data — identified with hashes. Transport deals with data streams, their verification and storage.

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