Peer-to-Peer Sharing with CCNx 1.0

Marc Mosko[1]*

Abstract
We describe a distributed peer-to-peer sharing service using CCNx 1.0 and Nameless objects. The system is very efficient, as a client only needs to create and sign one object to serve content. it does not need to modify or hash any other objects. A consumer can detect correct operation at every step. It can verify the original manifest embedded in a re-published manifest immediately. It can detect correct operation for every fetch request because all subsequent objects have a hash chain verification from the original publisher's embedded manifest. We describe a service similar to BitTorrent. Our goal is not to re-invent BitTorrent, but to show how CCNx 1.0 and Nameless Objects enable such a distributed service.

Keywords
Content Centric Networks – Named Data Networks

[1] *Palo Alto Research Center*
*__Corresponding author__: marc.mosko@parc.com

Contents

Introduction

In CCNx, a file typically is chunked in to individual Content Objects that have a specific name, version stamp, and chunk number. The name is used both for routing and to identify the Content Object. Because of this binding, creating a distributed sharing service requires that entire content distribution trees be renamed, re-hashed, and a new root manifest signed. A consumer cannot detect error until the entire object is downloaded to verify the overall file hash.

We describe a distributed peer-to-peer sharing service using CCNx 1.0 and Nameless objects ("System For Distributing Nameless Objects Using Self-certifying Names" (20140480US01)). The system is very efficient, as a client only needs to create and sign one object to serve content. it does not need to modify or hash any other objects. A consumer can detect correct operation at every step. It can verify the original manifest embedded in a re-published manifest immediately. It can detect correct operation for every fetch request because all subsequent objects have a hash chain verification from the original publisher's embedded manifest.

We describe a service similar to BitTorrent. Our goal is not to re-invent BitTorrent, but to show how CCNx 1.0 and Nameless Objects enable such a distributed service. One significant difference to BitTorrent is that the Nameless Object Manifest is a multi-object entity. A client does not need to download the entire manifest before beginning to download chunks. In fact, part of the novelty that makes this system work is that the root manifest, which carries the signatures, is a small object easily replaced at the top of an unsigned and unnamed Manifest tree. Various optimizations and features that apply to BitTorrent also apply to this service, such as swarms, choking, endgame mode, and so forth.

In outline, the protocol works as follows. One or more trackers keep a loosely synchronized database of content names

The tracker service may be public or may require an authentication step. It could use, for example, a system like "System And Method For A Reliable Content Exchange Of A Ccn Pipeline Stream" (20140361US01) to transfer data over an authenticated and possibly encrypted channel.

1. Background

Nameless Objects are a Content Object without a Name. They can only be addressed by the ContentObjectHash self-certified name. An Interest would still have a Name, which is used for routing, and a ContentObjectHash, which is used for matching. On the reverse path, however, if the Content Object's name is missing, it is a "Nameless Object" and only matches against the ContentObjectHash. This means that a requester can fetch the data from anywhere it lives.

Because the downloader trusted the initial Manifest response, which is a named object with proper cryptographic signature, it will trust the Manifest and the enumerated Content Object hashes. Nameless Objects are truly a placeless object: they have no name, so no implied routing.

Having no name means that they will never match an entry in the PIT that requested something only by Name, or perhaps Name and KeyId. That is the desired action, because a Nameless Object does not obey routing, so it could be a so-called "off-path attack". For example, if a Content Object could be retrieved by only Content Object Hash, but also carried a name, an attacker, Eve, could issue an Interest with the name, for example, of `/hacker/attack` with a Content

Object hash, and have it return a content object with the name `/parc/csl/slides.pdf`, even though the Content Object is not those slides and the Interest would never have been routed to PARC. This could cause a timing attack against valid requests.

To publish a Nameless Content Object, one would first create a signed Manifest with an authoritative name in it. The Manifest would need to enumerate the possible content distribution names and the Nameless object's Content Object hashes. When the storage replicas change, the manifest must be changed, which could be expensive for a large manifest.

As an alternative, one could publish both the Manifest and the Content Objects as nameless objects. Then, one would publish a single small Manifest that only links to the Nameless Manifest via the available replicas. This extra step of indirection could by more flexibility in publishing.

2. Protocol

A first system establishes itself as a Tracker, preferably on any anycast name `/tracker`. Other trusted systems may join as additional trackers on the same anycast address. The trackers need to run their own consistency protocol to keep the tracker databases eventually synchronized. We assume that all messages carry a serial number (or timestamp) so all protocol messages are well-ordered and duplicates are avoided.

An original publisher creates a Manifest tree that points to a set of Placeless Content Objects. The original publisher signs the root of the Nameless Object Manifest tree with its key pair identified by its KeyId. The root Nameless Object Manifest also carries the original publisher's CCNx name for the content. One method of naming the manifest is Eq. 1, where `/publisher` is the routable prefix to the publisher, `/OBJSTORE` is a place to store objects, and `filehash` is the SHA-256 hash of the entire content object once reconstructed from all the constituent objects (not to be confused with a CCNx 0.x or NDN implicit hash name). Protocol correctness does not depend on the exact name, so long as it is unique on the publisher.

```
/publisher/OBJSTORE/filehash                    (1)
```

The Nameless Object Manifest, among other things, creates a security binding between the original publisher, who signed it with a specific KeyId, the original publisher's name, and the ContentObjectHash of the root of the Nameless Object Manifest tree. We specifically split the root manifest object from the rest of the manifest tree. The goal is to keep the root manifest small with a minimum number of Manifest entries. The remaining Manifest objects are all Nameless objects and do not have a CCNx Name embedded in them. Only the root Nameless Object Manifest has a CCNx Name.

An original publisher contacts the Tracker (via any cast routing), possibly using an authenticated exchange protocol such as "System And Method For A Reliable Content Exchange Of A Ccn Pipeline Stream". It uploads its original

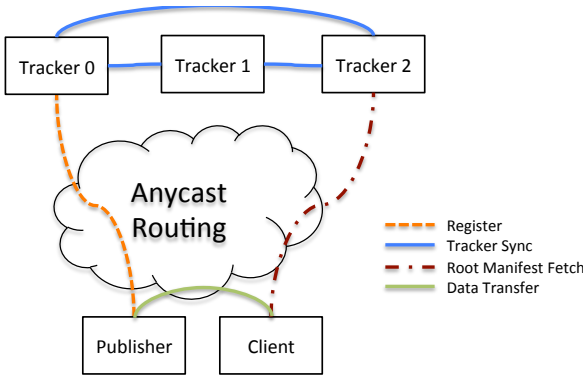

Figure 1. Manifest Upload and Transfer

Nameless Object root Manifest. This should be a single Content Object, as it only carries the Name, Signature, and minimal Manifest entries to the rest of the Nameless Object manifest tree. The root manifest should also embed the public key to authenticate the object (whose hash should equal the KeyId). Once the tracker verifies the root manifest signature, it may then begin servicing requests for the original name.

A downloader issues a request to the Tracker for the original name. As the tracker only has one entry, it returns a Link to the single name of the original manifest. The downloader may now download the Nameless Object Manifest and contents from the publisher, who happens to be the only source listed at this point. Downloading proceeds as follows.

The downloader chooses a peer and sends a Interest message with the name `/peer/index/publisherhash/next`. This message will query the peer about which pieces of `publisherhash` it has. The `publisherhash` is the ContentObjectHash of the original publisher's nameless object manifest – the piece that is being embedded in each seeders upload to the tracker. The `/next` is the last known peer's "next" response, or 0 if unknown. When a peer receives an `index` request, it will respond with its current index of the specified manifest. The response includes the "next" number to use in the request, i.e. the next sequence number the peer would publish an index under if asked. No matter what the requested "next" number, it always responds with the current index and the current "next" value. This technique allows downloaders to quickly learn the current contents without knowing any state *a priori* and to keep asking for the next state if the peer does not have all the content. Note that the response will be framed as a Manifest object. Once the downloader knows the set of objects for a given `publisherhash`, it can begin downloading them in any order. Given responses from different peers, it can begin striping requests among multiple peers. To download a content object, it constructs an Interest with the peer's routable name for the content, as in Eq. 2, and specifies the exact ContentObjectHash desired. The ContentObjectHash is what selects a specific chunk from the overall object.

Fig. 1 illustrates the original upload process. The pub-

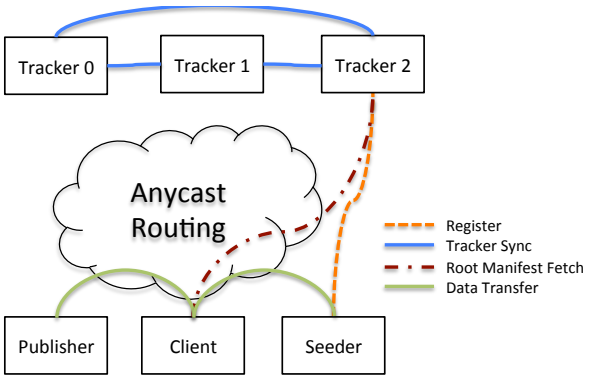

Figure 2. Seeder

lisher sends a register request to a tracker via any cast. Once accepted and processed, the tracker synchronizes its databased between all trackers on the any cast address. This may be a lazy process and it is not required for all trackers to be in exact synchronization. A downloader may then request a root manifest from the trackers via any cast. Once it has verified the manifest, it can begin transferring the data peer-to-peer from the registered seeder.

Once a downloader has downloaded the content, it can create a new root manifest. The downloader root manifest will have a CCNx name like 2, where `/client` is the downloader's routable prefix. As with Eq. 1, protocol correctness does not depend on the exact name, as long as it is unique on the downloader. The payload of the root manifest is the embedding of the original Manifest. The downloader then signs the root object. This downloader manifest promises that the downloader can serve the content named in the embedded manifest using the exact same ContentObjectHashes listed in the original manifest, but fetched from the client's routable prefix, given by the client's Manifest name.

```
/client/OBJSTORE/filehash                    (2)
```

The client may then upload the client Manifest to the tracker, becoming a Seeder. Again, this could be an authenticated, reliable upload. Once the tracker verifies both the client signature and the embedded original manifest signature, it may add the seeder's routable prefix to the list of possible routable prefixes for the content. Note that the seeder only had to create one Content Object and perform one hash and signature. It did not need to re-name the entire object tree.

Fig. 2 shows transfer from a seeder. Once the seeder registers content, the tracker will report it as a possible routing prefix for the content. The client may then begin data transfer from one or both of the registered providers using their individual routable prefixes.

A seeder or publisher may withdraw an entry by sending a signed withdraw request and the content object hash of the manifest to withdraw. If the KeyId is the same and the withdraw signature verifies, then the Tracker will stop advertising the client's routable prefix for the object. A seeder can also issue a complete withdraw request. In this case, the tracker should not respond with the client's routable prefix for any previous registration.

3. Extensions

The original manifest may also include keywords or title or other metadata that a client could use to search by.

The original publisher and client root manifests could give an additional CCNx name to use as the routable prefix to fetch the content rather than using a continuation of the manifest's name.

A seeder could upload its nameless object manifest before it has downloaded any data from a peer. It could, in fact, upload its manifest immediately after downloading one from a tracker.

The tracker could be extended to support both seeders and leechers, so one could download from systems with the entire content (seeders) or from partially downloads (leechers).

When a peer responds to an Index request, it may use "Difference Based Content Networking" (20140078US01) to encode the response as a set of diffs.

Such a service could operate without Nameless objects, but it would require each client to rename every content object, re-compute the manifest tree, and sign the new root manifest. It would still need to embed the original manifest from the original publisher so the client knew what the final cumulative hash should be, but it would not be able to detect corruption until the entire object was downloaded (or some sort of incremental hash tree was used).

4. Conclusion

We have presented a distributed peer-to-peer sharing service using CCNx 1.0 and Nameless objects. The system is very efficient, as a client only needs to create and sign one object to serve content. it does not need to modify or hash any other objects. A consumer can detect correct operation at every step. It can verify the original manifest embedded in a re-published manifest immediately. It can detect correct operation for every fetch request because all subsequent objects have a hash chain verification from the original publisher's embedded manifest.