CCNx 1.0 Collection Synchronization

Marc Mosko[1]*

Abstract
We describe a method to synchronize content between peers on a network using exact match names, as in the CCNx 1.0 protocols. The synchronized content could be files, service discovery, device discovery.

Keywords
Content Centric Networks – Named Data Networks

[1] *Palo Alto Research Center*
***Corresponding author**: marc.mosko@parc.com

Contents

Introduction

In a CCNx network where Content Names use equality matching, rather than prefix matching, to Content Objects, the current CCNx sync protocol will not operate correctly. The current protocol is based on responses that extend an interest name. This work proposes a mechanism to execute collection synchronization over a named network where the names use exact matching between Interest and Content object.

1. Previous CCNx Sync 0.8x

In today's synchronization protocol , a node with a Collection C under a given prefix PC, represents the collection as

(1) `ccnx:/routing/prefix/%C1.S.ra/hash{PC}`

The section `/routing/prefix` is an arbitrary length routable prefix to allow nodes communicating about the prefix to share state. The name component `%C1.S.ra` is a special marker to indicate the following name components talk about synchronization. The `hash{PC}` is a cryptographic hash of the name of the collection. Any two nodes talking about the same named collection would have the same name component.

A node with a collection C sorts all the names under that collection and computes an additive hash over that substructure, such that a topmost hash, called the "root hash" summarizes the contents of the collection. It then sends an Interest of the following form, where `<roothash>` means the value of the root hash.

(2) `ccnx:/routing/prefix/%C1.S.ra/hash{PC}/<roothash>`

If some other node has the same collection, but a different root hash, it responds with a content object named

(3) `ccnx:/routing/prefix/%C1.S.ra/hash{PC}/<roothash>/<responsehash>/<version>/<segment>`

where the `<responsehash>` is the second nodes idea of what the root hash should be. The `<version>` is an optional identifier for the version of the response and the `<segment>` is an optional appendage to version to indicates that the response is multiple messages long.

When the first node receives the response, it looks at the contents of the response and begins sending new interests to resolve the discrepancies.

If the second node is malfunctioning or malicious, its response would consume the network state about the first node's Interest. The first node would need to re-express its interest, but now exclude the unwanted response.

2. Sync with Exact Names

Sync with Exact Names is a three phase process. In the first phase, nodes advertise the hash of their catalog manifest. In the second phase, nodes retrieve unknown manifests and determine the set difference between their manifest and the remote manifest. In the third phase, nodes transfer the data to resolve the set difference.

Sync with Exact names uses a three part CCNx name. The first part is a routable prefix that identifies the collection. The second part is either `/adv` for advertisements or `/data` for

data transfers. The third part is the hash or content being advertised or transferred.

(4) `lci:/collection_prefix/adv_or_data/protocol_data`

2.1 Hash Advertisement

Sync with Exact Names uses the new URI-format CCNx names, where each name path segment may contain an explicit path segment type, such as `%F0%00` as an application-specific type, or `%00%10` as a content object chunk number.

We use one name space for hash advertisements. The 0xF000 name segment is the hash being advertised.

(5) `lci:/paravion/foo/adv/%F0%00=<hash>`

A node will advertise a hash when it first generates it, then a few times frequently afterwards. For example, when a collection changes, it may advertise the hash. If a collection changes and a new, unknown hash results, the node should advertise it once, then repeat it once or twice over the next few seconds to make sure all nodes on the segment see it. If the change results in a previously-seen hash, then the node does not need to aggressively advertise it.

If a node receives a hash it has never seen before, it should advertise its hash. It should use a gossip protocol to avoid flooding a hash that many other nodes are sending.

When a node comes on to a network segment, it may send an interest, such as `lci:/paravion/foo/adv/query`, that causes other nodes on the segment to send their root hashes. It should also send its hash.

(6) `lci:/paravion/foo/adv/query`

Nodes should use a gossip protocol to avoid flooding the segment with many repeats. For example, nodes pick a random back off. A first node sends it hash. A second node has the same hash and sends it too. A third or later system would suppress sending their hashes if they are the same.

Interests in the advertisement prefix are not expected to retrieve data. They are meant to advertise state. Therefore, they should have a short lifetime in the PIT. For example, if a node will repeat a broadcast of a hash every second the first few times to make sure it gets delivered, the PIT lifetime should be shorter than that period.

2.2 Retrieve Manifest

Sync with Exact Names uses a simple manifest that enumerates all items in the catalog. An extension could use a hash tree, such as in rsync or 0.8x sync for more efficient manifest transfer and set difference computation.

A system will transfer a manifest using interests in the `data` namespace. A system will begin requesting chunks $0, 1, \ldots$ up to the ending chunk number based on the segmentation protocol.

(7) `lci:/paravion/foo/data/%F0%01=<hash>/%00%10=chunk_number`

2.3 Resolve Set Differences

Based on the manifest entries, a system will retrieve content objects in the collection with an Interest in the `data` space.

(8) `lci:/paravion/foo/data/%F0%04=<sha256hash>`

In one variation, the `%F0%04=<sha256hash>` is the SHA-256 hash of the object in the manifest to retrieve (as reported in the manifest). The returned content object has the desired content object embedded in the reply.

In a second variation, a node retrieves the content object using its name name, such as below. This requires that every node running the sync protocol register that prefix and have appropriate routing.

(9) `lci:/whatever/name/was/in/manifest`

3. Optimization Heuristics

To avoid many nodes sending a manifest chunk or an object chunk, a node that has recently advertised a hash should take priority sending the manifest. Other nodes with the manifest should only reply on a second or later retry for the manifest chunk.

4. Security

When a device transfers a Manifest, it may verify that the party sending the manifest is authorized for the prefix using some trust model.

When a device retrieves a content object based on the manifest, it may verify that the publisher of the object is authorized for the prefix using some trust model.

In another variation, the advertising interest may be signed and contain a certificate so a device will only attempt to transfer manifests from authorized systems. Providing the certificate allows the manifest transfer to be based on a keyid.

5. Service Discovery

To use Sync for service discovery, the namespace would be divided by service type, such as printers, file servers, or music libraries. Devices of a specific type would run the Sync protocol within those namespaces.

(10) `lci:/parc/services/printers/...`

(11) `lci:/parc/services/servers/...`

(12) `lci:/parc/services/music/...`

A printer, for example, would run the Sync protocol under `lci:/parc/services/printers/` and generate advertisements of a manifest of service records. A service record describes the service, such as a printer make, model, and options, along with its CCNx name so users could directly print to it. Service records would have a TTL, such as

measured in seconds, so they would expire if not refreshed periodically by the device advertising the service.

A example series of events would be like this:

1. A printer `/parc/marvin` boots up with an empty manifest.
2. It creates service record for itself, being a Content Object named `/parc/marvin/service`. It has, for example, a JSON format and one of the fields is a serial number and another is a time to life (TTL). This creates a unique Content Object Hash for the service record.
3. It creates a manifest containing its own service record's hash, which results in some other hash to be advertised.
4. It sends a `query` advertisement so it can download other system's manifests.
5. It sends its own advertisement.
6. It downloads other manifests, resolves differences, and eventually synchronizes with all other devices.
7. A bit before the time its TTL will expire, it create a new service entry with the next serial number, replacing the previous record. this changes its advertisement hash, and the process repeats.

When a device is shutting down, it should create a new service entry with a 0 TTL and larger serial number, and advertise that.

If an Interest can contain payload, a device should always include its service record in the payload of its advertisements. Other devices should then merge in the new service record and update their hashes. This will likely bring them up to sync with the other device without needing to download the manifest or service record.

Service records replace records with lower serial numbers. Old records are not preserved in the manifest.

6. Conclusion

Sync with exact names uses two name spaces to separate hash advertisements from data transfers. It uses a three phase process of advertising hashes, retrieving manifests, and resolving set differences to retrieve missing content objects.