NDNREPO: Transparent Permanent Storage for NDN-Ready Content Objects

Zhe Wen
UCLA Computer Science, Los Angeles, CA 90095

1. INTRODUCTION

Content object repository (REPO) is a very important application that facilitates NDN wire-format data storage and fetching. On the one hand, the REPO listens to interests under certain set of NDN prefixes, and replies the requested content object if available; on the other hand, the REPO accepts command interests signed by certified requesters and performs insertion/deletion operations as requested.

The previous REPO implementation, i.e. ccnr, leveraged its low level storage on raw file system and lacked support for access control and deletion. Its successor migrated to SQLite database and enabled deletion operation. In both versions, however, processing and matching the interest prefix and selectors to search for content object had been the major, if not the only, headache that compromised the simplicity and elegance of REPO design and implementation.

Compared with both ccnr and its SQLite based successor, our new REPO prototype exploits graph database which provides intuitive and natural support for NDN hierarchical naming search and data storage. Specifically, we base our implementation on Neo4J, a popular graph database that comes with its own graph storage model (in contrast to either the relational model or the key-value pair model) and query language (Cypher) that features graph pattern (node, relation, path, etc) search that natively fits into the NDN naming philosophy.

The only possible drawback of the prototype is that we used the TCP based RESTful Neo4J API set for Python, which might become the bottleneck of query performance. However, it is still good enough to reveal the pros and cons of exploiting a graph database for NDN REPO and can be later improved via migrating to another Neo4J driver that accesses the low level storage directly.

Our prototype is constituted by:

- REPO driver. Library that provides API set for insertion/deletion/extraction operations on the REPO.
  - insertion. wraps given content as an object and insert it into REPO under specified name; insert given content object into REPO under specified name.
  - deletion.[1]
  - extraction. searches REPO for content object as requested by the interest (prefix + selectors).
- REPO server. listens to incoming interests under specified prefixes and replies the requested content object if available.

For the first prototype release, our REPO prototype is supposed to provide the following functionality:

- REPO API set. support for REPO insertion/deletion/extraction operations via REPO driver API set.
- data extraction. listens to and replies interests of content object fetching.[2]
- data insertion. data insertion protocol [3]
- data deletion. data deletion protocol [4]

2. DESIGN

2.1 Overview

NDNREPO is based on graph database, which makes the design simple and elegant. The tree-style NDN hierarchical name naturally fits the graph representation. Therefore it is intuitive and effective to split a given NDN name into a sequence of components as nodes, and connecting them with simple relations that leads to the next component. The actual content object can be attached to the leaf component with another relation.

The root component of the NDN names, can be exploited to optimize searching the graph database. The graph database searches for requested patterns (node, relation, path) through traversing the graph, which could be done much more efficiently if the start node is known ahead. The root component of a NDN name (for now it is "/ndn", so we have only *one* name tree in the graph database) serves as

[1] to be decided
[2] command interests not supported yet
[3] not available now
[4] not available now

this start node perfectly and it is therefore natural to trace the requested content object following the name components in order.

2.2 Nodes and Relations

Nodes and relations, both can come with a type and multiple schema free properties, constitute the property graph database. We use a node to represent a name component and a relation to connect 2 name components.

For example, given 6-component name "/ndn/ucla.edu/melnitz/1451/power/seg0", we use accordingly 6 "Component" type nodes to represent each component with the "component" property that stores the component content. Specifically, node (root:Component {component:"ndn"}) is store in the graph database for the first (root) component. Similarly, node (comp:Component {component:"ucla.edu"}) is for the second component. To store the actual content object under this name, we exploit another "Segment" type node. This node is attached to the leaf "Component" node and has a "file" property whose content is Base64-encoded wired format object data. For instance, node (data:Segment {file:"¡Base64-endcoding-data¿"}) stores the actual content object in wire format.

2 types of relations are used to connect graph nodes. The "CONTAINS_COMPONENT" relation connects one name component to the next name component. One example is the relation connecting component *root* and component *comp* in above example. The "CONTAINS_SEGMENT" relation connects the leaf name component to the actual content object data node. In above example component *leaf* (leaf:Component {component:"seg0"}) and segment *data*, which does not corresponds to any component in the name, are connected with this relation.

Differentiating nodes of components and nodes of content object data facilitates identifying the content of a node. The name/selectors based search is limited in the isolated name tree consisting of mere "Component" nodes connected by "CONTAINS_COMPONENT" relations. Once we find a node that fulfills all selector requirements under the specified name prefix, its "CONTAINS_SEGMENT" relation directly leads to the content object data requested. In this design, each node has *at most* one outgoing "CONTAINS_SEGMENT" relation.

2.3 Content Object Data

The content object data is stored as the "data" property of the "Segment" nodes. Ideally the wire format content object is supposed to be stored as binary data in the node. However, we need to use Base64-encoded data for storage due to limit of the Python binding exploited in the prototype.

3. IMPLEMENTATION

In this section, we discuss the implementation details of the neo4j based REPO prototype.

3.1 Interact with Neo4j

Neo4j is a world-leading graph database with full ACID support. Neo4j comes natively in Java and is built on top of native graph storage. Therefore the logic structure of neo4j nodes and relations is reflected in the underlying storage exactly the same way.

We chose py2neo, a Python binding for neo4j database, to implement our REPO prototype. py2neo interacts with neo4j server with the Cypher query language, which to neo4j is as SQL to relational database, via neo4j server's RESTful API.

Cypher query language provides readable and flexible interaction with neo4j. We introduce the 4 most frequently involved Cypher clauses in our prototype as follows:

- START. Every query describes a pattern, and in that pattern one can have multiple starting points. A starting point is a relationship or a node where a pattern is anchored. We use the START statement to describe the starting point of the pattern we search for such that the unnecessary patterns (with the incorrect starting point can be avoided). *START n=node({ids})* identifies the starting point by the internal node id of neo4j.
- CREATE UNIQUE. CREATE UNIQUE is in the middle of MATCH and CREATE: it will match what it can, and create what is missing. CREATE UNIQUE will always make the least change possible to the graph: if it can use parts of the existing graph, it will. *CREATE UNIQUE* {*pattern*} creates the unique pattern described by {pattern}.
- MATCH. MATCH is the primary way of getting data from the database into the current set of bindings. The MATCH clause allows you to specify the patterns Cypher will search for in the database. *MATCH* {*pattern*} searches the graph database for the pattern described by {pattern}.
- DELETE. Deleting graph elements, i.e. nodes and relationships, is done with DELETE. We first use the MATCH clause to search the graph database, and then use the DELETE clause to delete the matching patterns. Note that if any end node of a relation is removed, this relation should also be removed. *MATCH p={pattern} DELETE p* deletes all patterns described by {pattern}.

In the following subsections, we describe in detail how we implement content object lookup, insertion, and deletion with above clauses.

3.2 Content Object Lookup

We use a 2-step method to look up a content object specified by the interest, i.e. a name prefix plus a number of selectors.

We first search for the designated pattern specified by the name prefix. Following the way the graph database

is constructed, the prefix is split into components and therefore path by connecting them with the "CONTAINS_COMPONENT" relation. This search starts from the root component "ndn" with the START clause, which guarantees we do not take part of another branch not from the root in the name tree by mistake. If there is no match found, lookup finished without finding a match. Otherwise, the selectors are applied if any before we have the final result. By the hierarchical naming scheme, the matching node is identical should there be any.

Selectors include Exclude, ChildSelector, Min/MaxSuffixComponents, PublisherPublicKeyLocator, and MustBeFresh. Due to the difficulty in defining equality of PublisherPublicKeyLocator and meaning of MustBeFresh in the REPO, we only consider Exclude, ChildSelector, and Min/MaxSuffixComponents.

The selectors are applied sequentially as follows:

- The order of applying selectors is: Exclude/ChildSelector $\rightarrow$ Min/MaxSuffixComponents
- Exclude/ChildSelector. both apply to the immediate continuation component of the given prefix in the interest. Should either of them presence, handle them first. The result of applying these 2 selectors is that from the given prefix, branches are pruned.
- Min/MaxSuffixComponents. if there is no Exclude/ChildeSelector, simply return all content objects that fall into the requested range; otherwise, return all content objects that fall into the requested range and are on the branches NOT PRUNED. If there if on suffix component selector, simply assume MinSuffixComponents = 0 and MaxSuffixComponents = 64 (virtually infinite).
- If multiple content objects qualify (after applying all available selectors), select the minimum one by canonical ordering.

3.3 Content Object Insertion

The REPO prototype provides support for inserting both raw content and content object in wire format. For raw content, it will first be wrapped as a content object (signed) before encoded into wire format. This is done by PyNDN2 library.

To insert a wire format content object into neo4j, it is first encoded into a string with base64 encoding, given py2neo and neo4j does not have very good support for binary data operation. Once base64 encoded, the wire format content object can be treated as a property of a SEGMENT node and therefore inserted into neo4j.

When looked up, base64 decoding is needed before having the wire format content object available.

3.4 Content Object Deletion

Deletion is done in a very similar way to lookup. We delete content objects by an interest: all content objects that match an interest, i.e. all content objects left in the lookup procedure, will be deleted. This can be easily done with the DELETE clause.

One thing to note is that both insertion and deletion should be done via command interests. The current REPO prototype only provides support for related neo4j operations (insert/delete). To fully support command interests, parsing command interests is needed.