

The OpenLDAP Proxy Cache

Apurva Kumar
IBM, India Research Lab
kapurva@in.ibm.com

Abstract

This paper describes the design, implementation and usage of a query caching extension of the OpenLDAP directory server's proxy capabilities. The extension allow caching of LDAP search requests (queries). The LDAP proxy cache stores data and semantic information corresponding to recently answered queries and determines if an incoming query is semantically contained in any of the cached queries. It uses the *meta* backend to connect to one or more backend directory servers.

1. Introduction

There has been a growth of websites providing dynamic content on the web. Typically dynamic content is generated in response to a user request (*query*) evaluated against a database. Techniques used for traditional content caching are thus not useful for caching dynamic content. Directories are specialized databases which are capable of storing heterogeneous real world information in a single instance.

The Lightweight Directory Access Protocol (LDAP) provides a means for accessing and managing remote and distributed directories [1-3]. LDAP directories are being used to store address books, contact information, customer profiles, network resource information, policies etc.

Directories have assumed special significance in enterprises where a single directory instance containing employee and other organizational records is used by a wide variety of internet and intranet applications. The heterogeneous nature of directories allows them to be used by several applications. However, this also means that an overloaded directory could be a potential bottleneck in an enterprise infrastructure.

LDAP replication has widely been used for improving performance, scalability and availability of directory based web services. However, since a typical directory can contain millions of records, the search performance of replicas suffer due to disk access latency [4]. An LDAP caching solution which provides significant hit

ratio, for a small fraction of cached entries is thus desirable.

This paper discusses the proxy cache implementation for the OpenLDAP[5] directory server.

2. Notations

The LDAP search operation (also termed as an *LDAP query*) is represented as:

$q = (base, scope, filter, attrs)$
base: <dn of the base of the search >
scope: Base | one | sub <scope of the search>
filter: <search condition>
attrs: <set of required attributes>

These parameters are collectively called the *meta-data* corresponding to an LDAP query. LDAP filters are represented using the string representation described in [7].

3. Architecture

A typical directory server architecture consists of a protocol *front end* which receives a client request, uses the *backend* to perform the operation corresponding to the request and sends results back to the client. The backend abstracts the database from the front-end by performing the read and write operations corresponding to various LDAP operations.

Figure 1 shows the architecture of an LDAP proxy cache. It extends the *meta* backend [6] by including a cache manager which handles search requests. The cache manager maintains cache state and implements the *query containment*, *cache replacement* and *consistency control* (CC/CR in Figure 1) algorithms. The extended meta backend is associated with one or more database backends which are combined using the glue backend. These backends store the cached entries. The *cache backend* provides a query level interface to the cache manager for performing read and write operations on the database backends. It supports adding and removing a query (i.e. corresponding entries) or searching the cache for an answerable query by converting these operations

into the add, modify, delete and search operations which are supported by the database backends. The cache manager uses the meta backend functionalities to connect to one or more backend directories using the LDAP client API. Using meta backend allows the proxy cache to work as a meta directory cache.

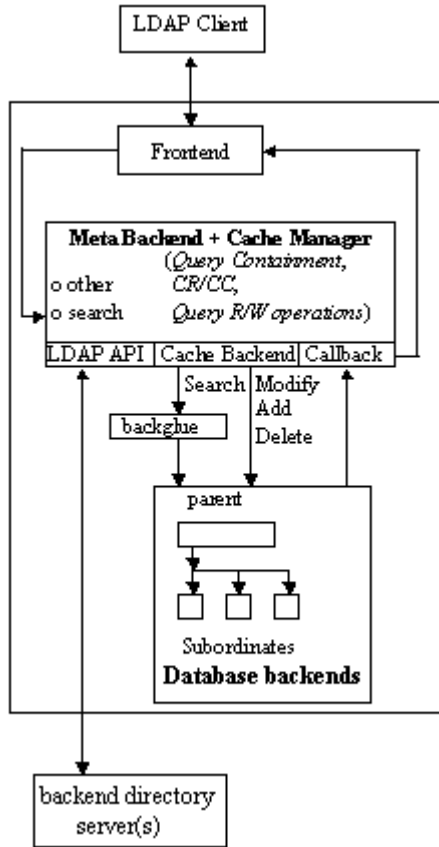


Figure 1: LDAP proxy cache architecture

4. Caching Algorithms

The following algorithms are implemented in the cache manager:

1. Query containment.
2. Cache replacement.
3. Consistency control.

4.1. Query containment

General query containment

A query Q , is said to be *contained* in another query Q' , if all of the following are true:

- (i) The *base* of Q' should be the same or a descendant of the *base* of Q . The *scope* of Q' must be *sub* except for

the case when the *base* of Q' is the parent of *base* of Q and the *scope* of Q is *Base*. In this case, the *scope* of Q' can be either *sub* or *one*.

- (ii) *attrs* in Q is a subset of *attrs* in Q' .

- (iii) The *filter* of Q is more restrictive than (or contained in) the *filter* of Q' , i.e. any entry satisfying the condition specified by the filter of Q also satisfies the condition specified by Q' .

If LDAP query filters $F1$ and $F2$ consisting of a boolean combination (AND, OR, NOT) of atomic filters are such that $(\&(F1)(!F2))$ is trivially inconsistent, then the filter $F1$ is semantically contained in filter $F2$.

Let the attribute set $\{x1, x2, \dots, xn\}$ be the union of attributes sets appearing in the filters $F1$ and $F2$. To be trivially inconsistent there should not exist any set of values $\{v1, v2, \dots, vn\}$ for attributes $\{x1, x2, \dots, xn\}$ in their valid ranges such that the condition specified by $(\&(F1)(!F2))$ is satisfied.

If $(\&(F1)(!F2)) = B1 + B2 \dots Bk$ where Bi is a conjunct of atomic filters, then the condition can be written in terms of Bi as follows: The query $F1$ is contained in $F2$ if $\forall i=1..k, Bi$ is trivially inconsistent.

A contained query is *answerable* (from the cache) if *attrs* in the conditions above is interpreted as the union of required attributes and the attributes appearing in the search filter of the query.

Template based query containment

To reduce the complexity of the problem, we introduce the concept of *LDAP templates*. Most queries generated by applications are different from other similar queries only in their assertion values. An LDAP template is similar to a query but consists of a *prototype filter* instead of a filter. The prototype filter is a filter without any assertion values. Different filters can be generated from the same prototype by supplying different assertion values. The string representation of prototype filters is similar to LDAP filters in [7], except that the assertion values are missing. An example of a prototype filter is:

$(\&(givenName=)(sn=))$

If the cache supports only a few specified templates, query containment between queries of two given templates simply requires substituting their assertion values in a pre-determined expression. In the current proxy cache implementation a simplified version of template based query containment is implemented, in which a query is only answered from queries of its own template.

The implemented algorithm supports answering of positive queries (no NOT operator) having equality, ordering or substring assertions. The query containment algorithm ensures that the correct syntaxes and matching rules are applied when comparing two assertion values.

4.2. Cache replacement

Cache replacement uses a simple Least Recently Used (LRU) policy. The metadata of the least recently used query and entries belonging only to that query are removed. A query is considered to be used when it answers an incoming query. Cache replacement is invoked when the cache size crosses a threshold (*hi-thresh*) and continues till it is greater than a *lo-thresh*.

4.3. Consistency Control

A time to live (TTL) value is associated with queries. The value is same for all queries of the same template. After the time to live expires the query (and entries belonging to only that query) are removed.

5. Caching Operations

The cache manager invokes the cache backend to add or remove entries corresponding to a query or to search the database backends for an answerable queries.

For adding or removing queries the *callback mechanism* of OpenLDAP has been used. The mechanism allows the routines for sending *search entry* and *search result* to be overridden by those supplied by the caller. If the search interface for a backend is invoked using this mechanism, the supplied routine for sending an entry is called once for every entry returned by the search and the routine for sending result is called at the end of the search. The callback mechanism has been used to add/remove queries to/from the *database backends*.

5.1. Adding a query

Adding a query requires all its entries to be added in the cache. However, since some of these entries might be partially or completely cached. For each entry, it is determined using an internal search request, whether the entry is already cached. If the entry is already cached, an internal modify is performed in the callback for sending the entry, if not, the entry is added in the callback for sending the search result.

While adding/modifying an entry, a value for a cache specific operational attribute *q_uuid*, is added to the entry which represents the unique identifier of the query being cached. An entry which belongs to multiple queries has multiple values for the *q_uuid* attribute.

The cache backend informs the cache manager after adding a query, so that its metadata can be added.

5.2. Removing a query

To remove a query with a given identifier, say *ID*, an internal search: filter: (*q_uuid=ID*) is performed. In the callback for sending the matching entry, the entry is *deleted* if it has a single value for the *q_uuid* attribute. Otherwise the entry is modified to have the value, *ID*

removed from its *q_uuid* attribute. The cache backend informs the cache manager after removing a query, so that its metadata can be removed.

6. Configuration

At configuration time, a list of cacheable templates is specified. Queries belonging to only these templates are cached. An incoming query is checked for containment in the cached queries belonging to templates with the same template string. If an incoming query can not be answered from the cache, the results are obtained from the backend directory server and sent to the client. If the number of entries returned is within a cacheable limit, the entries are added to the cache backend and the meta data for the query is added.

6.1. Cache specific configuration directives

Three cache specific directives have been added to back-meta.

1) *cacheparams*

Used to set various cache parameters. The syntax is:

```
cacheparams <lo_thresh> <hi_thresh> <numattrsets>  
                <max_entries> <consistency_cycle_time>
```

The first two parameters are used for cache replacement. Cache replacement starts when the cache size crosses the *hi_thresh* bytes and continues till the cache size is greater than *lo_thresh* bytes. Total number of attributes sets (as specified by the attrset directive) is given by *numattrsets*. Queries are cached only if they correspond to a cacheable template (specified by the addtemplate directive) and the number of entries returned is less than *max_entries*.

Support for weak consistency is provided by associating a time to live (*TTL*) with each query. The *TTL* is specified while adding a template using the addtemplate directive. The consistency checks are performed every *consistency_cycle_time* seconds. Stale queries and entries belonging to *only* those queries are removed.

Example:

```
cacheparams 100000 120000 3 5 1000
```

2) *attrset*

Used to associate a set of attributes to an index.

```
attrset <index> <attr1> <attr2> <attr3> ...
```

Each attribute set is associated with an index number from 0 to *numattrsets-1*. These indices are used by the *addtemplate* directive to define cacheable templates.

Example:

```
attrset 0 cn sn mail
```

3) *addtemplate*

Adds a cacheable template with a given TTL.

```
addtemplate <template string> <attr set index> <ttl>
```

template string representation is similar to [7] with any values in simple/substring filters omitted. Some examples of template strings are: *(sn=)*, *(age>=)*, *(&(sn=)(c=*))*. First template could have queries with equality and substring assertions. Second template represents queries with a simple *>=* assertion and the *(c=*)* in the third template represents a presence filter. The *attr set index* is used to associate one of the attribute sets defined by the *attrset* directive with a template. *TTL* (in seconds) is used to associate a time to live with queries of the template.

Example:

```
addtemplate (&(sn=)(givenname=)) 0 3600
```

6.2. Example *slapd.conf*

In this section an example configuration file for *slapd* when caching is enabled. To enable caching in the backmeta backend, the following cache specific directives should be specified.

1) *General Directives*

The schema file could be the same or a relaxed version of the backend server. Since partial entries are also cached, schema checks are relaxed while adding/modifying entries.

```
include <schema_file>
<other general directives>
```

2) *Database Backend directives*

```
#subordinate
database ldbm

suffix "ou=people,dc=example,dc=com,cn=cache"
directory <database_dir>
cachesize 1000
# other LDBM directives

# parent
database ldbm
suffix "dc=example,dc=com,cn=cache"
directory <parent_database_dir>
```

```
cachesize 1000
# other LDBM directives
```

3) *Meta backend directives*

```
database meta
rewriteEngine on

# rule for rewriting DNS of entries to be
cached.
rewriteContext cacheResult
rewriteRule "(.*)dc=example,dc=com"
"%ldc=example,dc=com,cn=cache" ":"

# rule for rewriting the base for searching the
cache for answerable queries
rewriteContext cacheBase
rewriteRule "(.*)dc=example,dc=com"
"%ldc=example,dc=com,cn=cache" ":"

# rule for rewriting DNS of cached entries
before sending the result to the client.

rewriteContext cacheReturn
rewriteRule "(.*)dc=example,dc=com,cn=cache"
"%ldc=example,dc=com" ":"

#pointer(s) to backend directory server(s)
suffix "dc=example,dc=com"
uri ldap://<backend_hostport>/dc=example,dc=com
#setting cache parameters
cacheparams 10000 15000 4 50 1000

#attribute sets
attrset 0 cn title
attrset 1 homephone pager telephonenumber mail
attrset 2 postaladdress
attrset 3 member

#cacheable templates
addtemplate (uid=) 0 3600
addtemplate (sn=) 1 3600
addtemplate (sn=) 2 3600
addtemplate (&(objectclass=)(cn=)) 3 100000
```

The Relative Distinguished Name (RDN), *cn=cache* is appended to the Distinguished Name (DN) of entries added to the database backends. Thus DN rewriting, as described by the rewriting rules above, is required while adding entries to the cache and while sending entries to the client.

7. Future Work

The proxy cache feature of OpenLDAP can be used to improve client latency and scalability of directory based services. Work on schema for representing queries and templates in an LDAP directory and for supporting query containment is underway. *SLAPI* based implementation of the caching extension is under consideration.

8. References

- [1] Hodges, J, Morgan, R, "Lightweight Directory Access Protocol (v3): Technical Specification", RFC 3377 (<http://www.rfc-editor.org/rfc/rfc3377.txt>) September 2002

- [2] Wahl M, Howes T, Kille S, "Lightweight Directory Access Protocol (v3)", RFC 2251 (<http://www.rfc-editor.org/rfc/rfc2251.txt>), December 1997
- [3] Wahl M, Coulbeck A, Howes T, Kille S Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions, RFC 2252 (<http://www.rfc-editor.org/rfc/rfc2252.txt>), December 1997
- [4] Xin Wang, Henning Schulzrind, Dilip Kandlur, Dinesh Verma, "Measurement and Analysis of LDAP Performance", International Conference on Measurement and Modeling of Computer Systems, ACM Sigmetrics, 2000
- [5] OpenLDAP Project, Web page (<http://www.openldap.org>)
- [6] OpenLDAP Project, back-meta backend manual page, slapd-meta(5) (<http://www.openldap.org/software/man.cgi?query=slapd-meta>)
- [7] Howes T, "The String Representation of LDAP Search Filters", RFC 2254 (<http://www.rfc-editor.org/rfc/rfc2254.txt>), December 1997