Filter Based Directory Replication: Algorithms and Performance

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

Abstract

Directories have become an important component of the enterprise security and identity management middleware. This paper describes a novel filter based replication model for Lightweight Directory Access Protocol (LDAP) directories. Instead of replicating entire subtrees from a Directory Information Tree (DIT), only entries matching a filter specification are replicated. Efficient algorithms for selecting such filters, keeping them synchronized with the master copy and for using them to answer directory queries have been proposed. Advantages of the filter based replication framework over existing subtree based mechanisms have been demonstrated for a real enterprise directory using real workloads.

1. Introduction

Directories are specialized databases optimized for read access. They are traditionally associated with storing information about people and accessed using address books for fast lookup of email addresses, phone numbers etc. However, directories have a very flexible information model which allows them to represent real world entities like people, network resources (e.g. printers, servers), devices, machines, services, applications and policies in a single instance. With information about all these entities and their relationships accessible from a single location, directories cease to be merely repositories of information and provide a basis for application integration and policy based network management in an enterprise. However, this also means that an overloaded directory could be a potential bottleneck in an enterprise infrastructure.

Directory replication has been particularly effective in improving scalability. Most enterprises typically deploy a centralized load balanced array of replicas to scale their directories. However, full replication at remote sites to improve performance for remote users has not been very successful, particularly for large directories. Firstly, there is a significant setup, maintenance and administration cost associated with the heavyweight replicas. Secondly, even though directories are read more often than updated, the bandwidth costs of keeping a full replica in sync with the master copy is usually prohibitive. Thirdly, while directory servers are optimized for read access, full remote replicas will typically have a high update/read ratio. Thus partial replication of directories is desirable.

Lightweight Directory Access Protocol (LDAP) [1,2], the standard way of accessing directories conforming to the X.500 information model over TCP/IP, does support partitioning of directories across multiple servers. Each server can hold one or more proper subtree of entries. However, performance of distributed operations in LDAP is severely degraded because of the distributed directory being exposed to the client. In Section 2.3, the LDAP distributed directory model is described and reasons for performance issues with the *referral* mechanism are explained.

Partial replication of directories suffers from the same performance problems. The problem with distributed directories is worsened because typical applications are minimally directory enabled (Section 3.1.1) and perform searches on the entire DIT rather than individual subtrees.

The goal of this paper is to improve upon existing replication models by providing a better hit ratio (fraction of queries answered by a replica without generating referrals) for the same replica size. A novel filter based approach for replication of directory content is proposed. The unit of replication is entries in a subtree matching an LDAP search filter (see Section 2.2 for a definition). It has been shown that an LDAP query is a much more flexible unit of replication and can be used to describe regions demonstrating locality of reference in access patterns. In addition to replicating queries which describe regions of semantic and spatial locality, recently performed user queries can be cached to take advantage of any temporal locality in the access pattern.

Earlier work in database query caching [4-8] can not be directly applied to LDAP because of the inherent differences in data models and query languages of directories and relational databases [9,10]. Relevant work in LDAP query caching has been in the following areas: (*a*) determining whether an LDAP query is contained in another query (query containment), (*b*) using caching algorithms which make efficient caching, pre-fetching, decisions to maximize the fraction of queries answered from the cache. The complexity of (*a*) is the subject of [11]. The authors of [11] consider general query containment problem for LDAP and show it to be NPcomplete in the size of the query. The authors of [12] introduce the notion of generalized queries and propose algorithms for (*b*). They use a real directory but synthetic workloads and a single type of query for performance evaluation.

This paper reduces complexity of the query containment problem by using the concept of LDAP templates (Section 3.4.2). The query generalization concept of [12] has been developed further and a simpler algorithm for selecting filters to be replicated has been used (Section 6.2). Performance of these algorithms is evaluated for multiple query types, using real workloads for an enterprise directory of a large organization.

To keep the replicated content in sync with the master copy, a new filter synchronization protocol for LDAP has been proposed. The *ReSync* protocol supports both polling and notification modes for updating. The protocol minimizes synchronization traffic and provides convergence guarantees.

The rest of this paper is organized as follows: Section 2 provides an overview of the LDAP v3 [1,2] standard. Section 3 describes and compares, existing and proposed replication models. Section 4 describes the query containment problem and the proposed solution. Section 5 discusses the filter synchronization problem and describes the proposed protocol. Section 6 describes algorithms used for filter generalization and filter selection. Section 7 presents a case study based on a real enterprise directory. Section 8 concludes the work.

2. LDAP Overview

2.1. Information and naming model

LDAP assumes the existence of one or more directory servers jointly providing access to a *Directory Information Tree* (DIT), which is composed of entries. An *entry* is defined as a set of *attribute* value pairs with the required `objectclass` attribute determining its mandatory and optional attributes. Each entry has a *distinguished name* (DN) belonging to a hierarchical namespace. The root of the DIT has a “null” DN. Figure 1 shows an example directory tree and with an `inetOrgPerson` [13] entry. Each node is named with its relative DN (RDN). The DN of an entry is constructed by prefixing its RDN to its parent’s DN.

2.2. Functional model

The functional model [1] adopted by LDAP is one of clients performing protocol operations against servers. LDAP defines three types of operations: query operations, like *search*, *compare*, update operations like *add*, *modify*, *delete*, *modify DN* (entry move) and connect/disconnect operations like *bind*, *unbind*, *abandon*. The most common operation is *search*, which provides a flexible means of accessing information from the directory. The LDAP search operation, also referred as a query consists of the following parameters which represent the semantic information associated with a query: (*i*) *base*: A DN that defines the starting point of the search in the DIT, (*ii*) *scope*: {BASE, SINGLE LEVEL, SUBTREE}, specifies how deep within the DIT to search from the base, (*iii*) *filter*: A boolean combination of *predicates* using the standard operators: AND (&), OR (|) and NOT (!), specifying the search criteria, (*iv*) *attributes*: Set of required attributes from entries matching the filter. The special value “*” corresponds to selecting all user attributes. Every entry in the directory belongs to at least one (object) class, thus the filter (`objectclass`=*) matches all entries in the directory.

LDAP filters are represented using the parentheses prefix notation of RFC 2254 [3], e.g.: `(&(sn=Doe)(givenName=John))`. Filters without any NOT operators are called positive filters. Predicates of the form (*name operator value*) where operator ∈ {=, ≥, ≤ } are considered. *name* is an attribute name and *value* is termed as *assertion value*. Examples of predicates are: `(sn=Doe)`, `(age≥30)`, `(sn=smith*)` where “`Doe`”, “`30`” and “`smith*`” are assertion values representing equality, range and substring assertions, respectively.

LDAP *controls* can be attached to operations to alter their behavior. An example is the control for requesting, server side sorting of search results [14].

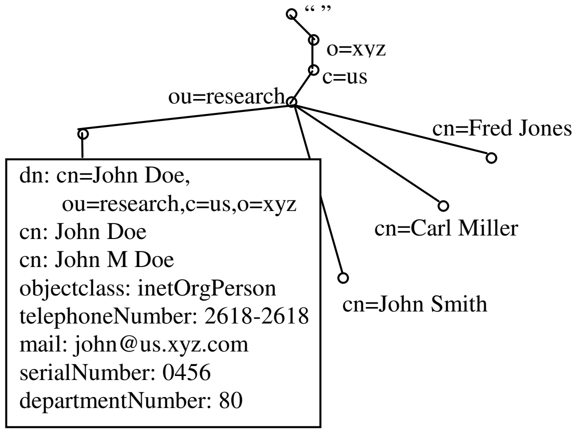

Figure 1: Example DIT and entry.

2.3. Distributed directory model

LDAP supports partitioning a directory into multiple servers with each server holding one or more *naming contexts*. A naming context is a subtree of the DIT rooted at an entry, known as its *suffix* and terminated by leaf entries or special *referral* objects. Referral objects point to other subordinate naming contexts. A naming context is defined as an n+1 tuple, where n is the number of referral objects. Formally $C = (S, R_1, R_2 \ldots R_n)$ where S denotes the suffix DN and $R_1, .. R_n$ denote DNs of referral objects.

Distributed operation processing in LDAP and the problems associated with the referral mechanism are illustrated by an example. Figure 2 shows three servers collectively serving the `o=xyz` namespace. hostA contains a single naming context with suffix as `o=xyz` and referrals for hostB and hostC which hold subordinate naming contexts. The client requests a subtree search with base as `o=xyz` from hostB.

First, a distributed name resolution is performed for the target `o=xyz`. Since hostB does not contain the target, it refers the client to hostA (default referral). The client contacts hostA which contains the target object. hostA returns three matching entries and referrals for hostB and hostC for naming contexts contained by them. Finally the client sends searches (with modified bases) to hostB and hostC which return the remaining entries. It requires four round trips between client and the servers to evaluate one request. This example illustrates why the referrals based LDAP operation completion mechanism is extremely slow.

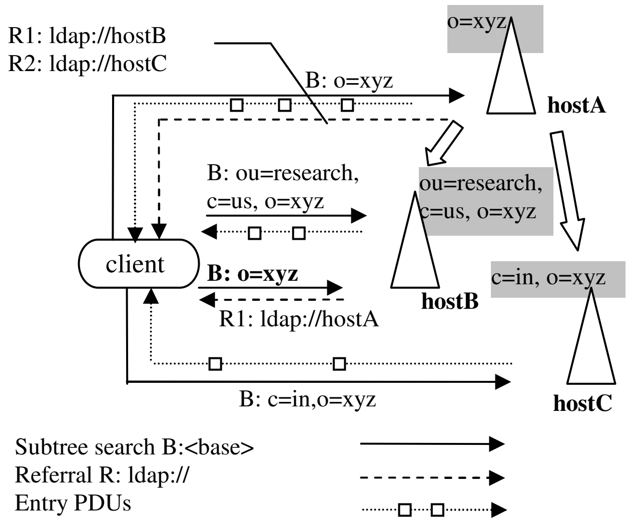

Figure 2: Distributed Operation Processing

3. Directory Replication Models

LDAP itself does not standardize replication, allowing different vendors to support their proprietary protocols and mechanisms. However, there are certain features common to all models. Each entry in the DIT must have at least one master server. Replication is typically based on an implicit or explicit agreement between the *consumer* (replica) and the *supplier* (master). The agreement identifies what is to be replicated (*replication unit*) and when the updates should be sent (mode of update). E.g. consumer could be notified immediately after an update takes place (on change) or the master could wait for the next update window in a periodic update scheme. In either case the consumer might be allowed to poll the master for changes since it was last updated. The agreement might be explicitly negotiated using a proprietary protocol, or implicitly reached between the administrators of the two systems. In this paper the following two replication models are discussed and compared. They differ in how the replication unit is defined.

Subtree Based Replication

Since directories are partitioned by naming contexts (Section 2.3), which are essentially subtrees of entries, it seems natural to define the unit of replication in a similar way. A subtree based replica contains one or more subtrees of entries. Most directory vendors implement a subtree based replication model. For each replication unit, the replica stores meta information which includes the context suffix and referrals (if any). The meta information is used to determine whether a query can be answered, or a referral needs to be generated.

Filter Based Replication

This is the replication model proposed in this paper. The replication unit is semantically equivalent to an LDAP query (Section 2.2). A replica stores entries satisfying one or more LDAP queries. For each replicated query, meta information comprising of the corresponding search specification is stored. The meta information is used to determine if an incoming query is semantically contained in any stored query. Otherwise a referral is generated.

Note that a query specification can be reduced to a subtree specification with base as the root of the subtree, scope as SUBTREE, and filter as `(objectclass=*)`.

It should be noted that these models are for replicating directory content and not for partitioning. In both the models, directory is partitioned between a set of master servers, each containing one or more naming contexts (essentially subtrees) as described in Section 2.3.

Sections 3.1-3.4 provide a comparison of subtree based and filter based models.

3.1. Hit-ratio comparison

Hit-ratio for a replica is defined as the fraction of client requests which can be completely answered (without generating referrals) by the replica. The following factors are responsible for a lower hit-ratio in subtree based replicas:

3.1.1. Minimally directory enabled applications. Many directory enabled applications work with a simplified model of the directory. The simplified model typically consists of relational database like tables corresponding to important object classes in the directory. Without knowledge of the hierarchical namespace most applications generate queries with base as the root of the DIT. These queries can not possibly be answered by subtree based partial replicas. This results in a lower hit-ratio for such replicas. Filter based partial replicas can replicate null based queries.

3.1.2. Semantic versus spatial locality. Any partial replication model will be useful only if there is some semantic locality of reference in the access pattern. Semantic locality in directories is not restricted to spatial locality. Consider two semantically close filters generated from: `(&(objectclass=inetOrgPerson)(departmentNumber=<value>))` with `value`s `2406` and `2407`. If the departments are located in different country subtrees, the result sets satisfying the two filters are not spatially close to each other and might not be held by the same replica. A filter based replica, however is not restricted to hold queries from a single subtree and could contain the filter `(&(objectclass=inetOrgPerson)(departmentNumber=240*))` which answers both these queries.

3.1.3. Partially answered queries. A subtree based replication context is not necessarily a complete subtree. A replica might point to servers holding subordinate contexts (e.g., see Figure 2). Thus even if the base of a query is inside a subtree of the replica, it is possible that the query does not contribute to hit-ratio since it generates referrals for subordinate servers. In filter based replication, once the base lies under the base of the replicated query, the query can be answered without generating referrals.

3.2. Update traffic

The replica must be updated to be consistent with the master copy. The mode of update might be through notifications, periodic batch updates or polling. All of these require modified entries in the replication context to be sent to the replica. Subtree based models store all entries from a subtree, generating larger update traffic volumes compared to the proposed filter based model which stores only selected entries from a subtree. For the same number of replicated entries the filter based model can thus span larger subtrees, thereby reducing the referrals generated from the replica.

Filter based replication also allows the flexibility of adapting to the access pattern. Since replication units can be very small in size, it is feasible to dynamically fetch new filters and discard old ones. This can be used to provide trade-off between hit-ratio and update traffic.

Finally, a filter based replica allows the flexibility of specifying different consistency levels for different types of objects. A subtree of entries could contain a variety of objects. A subtree based replication model will associate the most stringent requirement amongst these with the complete subtree, thus further increasing the update traffic.

3.3. Partial replication of flat namespaces

Some directories could have a very flat DN namespace. E.g. carrier directories used by large telcos can have all their subscribers (millions of entries) under a single container entry. Since subtree based replicas can not partially replicate the container's children, large replicas need to be deployed. Filter based replication can be used to selectively replicate entries from a flat namespace.

3.4. Query processing

The increased hit-ratio for filter based replication comes at the cost of increased query processing. Sections 3.4.1 and 3.4.2 describe complexity of the cache answerability problem for the two replication models.

3.4.1 Subtree based replication. To know if a query can be answered (fully or partially) a subtree based replica simply checks if the base object (b) of the incoming query lies inside any of the replication contexts held in it. Let a replica hold n replication contexts with the ith context defined as (see Section 2.3): $C_i = (S_i, R^i_1, R^i_2 \ldots. R^i_{Ci})$. The algorithm for determining whether the query can be partially or fully answered by the replica is as follows:

```
Algorithm isContained (b, C ) {
  /* C = {C_1, C_2 .... C_n } */
  for each Ci in C
    if (S_i = b)
      return TRUE
    if (!isSuffix (S_i, b) )
      continue
    for Rj in { R^i_1, R^i_2.... R^i_Ci }
      if (isSuffix (Rj, b))
        return FALSE
    return TRUE
  return FALSE
}
```

where *isSuffix* (a, b) returns TRUE if the DN a is an ancestor of DN b and FALSE otherwise.

3.4.2. Filter based Replication. A filter based replica needs to check for containment of an incoming query in all replicated queries. To reduce the complexity of the problem, we introduce the notion of LDAP *templates*.

Typical directory applications have a finite number of prototypes (templates) for generating query filters. In this paper templates are represented using the LDAP filter representation of [3] except that the assertion value is replaced by the "_" character. Examples are `(&(cn=_)(ou=research))`, `(uid=_)`, `(&(sn=_)(givenName=_))`, `(sn=_*)`. The last template is used to generate substring queries for the `surname` attribute.

In template based containment, queries belonging to only a specified set of templates are replicated and answered. Templates simplify the query containment problem in several ways. Firstly, number of query comparisons is reduced by eliminating containment checks against templates which can not potentially answer the query. E.g. a query of template `(&(sn=_)(ou=_))` can not answer a query of template `(sn=_)`. Secondly, for all the remaining cross template comparisons, conditions for containment can be computed apriori (Proposition 2 in Section 4.1). E.g. query `(age=X)` can be answered by query `(age ≤ Y)` if `(Y ≥ X)`. Thirdly, answerability against queries of the same template requires simply comparing the corresponding assertion values. This is formalized in Proposition 3 of Section 4.1. Section 4 discusses LDAP query and filter containment in detail.

4. LDAP Query Containment

A query Q is termed as semantically contained in another query Q_s if, (*i*) the region defined by the base and scope of Q falls completely inside the corresponding region for Q_s, (*ii*) the attributes in Q are a subset of attributes in Q_s, and (*iii*) the filter in Q is more restrictive than the filter in Q_s. This can be stated formally as:

```
Algorithm QC ($Q$, $Qs$) {
 /* $Q = (b, s, f, A)$, $Qs = (b_S, s_S, f_S A_S)$ */
  if ($b_S = b$ & $s_S \geq s$)
    goto NEXT
  else if (!issuffix ($b_S$, $b$))
    return FALSE
  if ($s_S$ = SUBTREE)
    goto NEXT
  else if (($s_S > s$) & isparent ($b_S$, $b$))
    goto NEXT
  return FALSE
NEXT:
  if ($A \subseteq As$ & $f \prec fs$)
    return TRUE
  return FALSE
}
```

where $f \prec f_s$ denotes that the filter f is semantically contained in fs and the scope values are assumed to be integers with BASE=0, SINGLE LEVEL=1, SUBTREE=2. *isparent* (a, b) returns TRUE if the DN a is the parent of DN b.

4.1. LDAP filter containment

An LDAP filter F1 is contained in F2 if it is impossible for an entry to satisfy F1 but not F2. This condition is formalized by Proposition 1.

Proposition 1: *(General LDAP query containment)*
An LDAP query filter F_1 is semantically contained in another query filter F_2 if and only if the expression $F_1 \wedge \neg F_2$ is inconsistent. ┐

For the expression $F_1 \wedge \neg F_2$ to be inconsistent:
$\nexists\, x_1, x_2..x_n$ such that $F_1 \wedge \neg F_2$ is satisfied.
where attribute set $\{x_1, x_2..x_n\}$ is the union of attribute sets appearing in the filters F_1 and F_2.
If $F_1 \wedge \neg F_2 = B_1 \vee B_2 \ldots \vee B_k$ where each B_i is a conjunction of simple predicates, then each B_i should be inconsistent, i.e. the following boolean expression should evaluate to TRUE.

$$(\nexists x_1,x_2..x_n(B_1)) \wedge (\nexists x_1,x_2..x_n(B_2))..\wedge(\nexists x_1,x_2..x_n(B_k)) \qquad (1)$$

where $\nexists x_1,x_2..x_n\,(B_i)$ represents the condition that B_i is not satisfiable for any values of attributes $x_1,x_2..x_n$ in their valid ranges. Let the set A_{XY} represent the union of sets of assertion values in LDAP filters X and Y

Proposition 2: *(Cross template containment)*
For positive LDAP filters F_1 and F_2 containing equality and range predicates, the condition for F_1 to be contained in F_2 can be expressed as a boolean expression in conjunctive normal form (CNF) with each simple predicate of the form:
$(a \geqslant b)$ where $a,b \in A_{F1F2}$. □

Sketch of proof: The expression in *(1)* is a conjunction. The condition for each B_i being inconsistent requires that the predicates in B_i should impose an empty range for at least one of the attributes appearing in it. Thus the condition of each B_i being inconsistent is disjunctive and *(1)* can be written in CNF. It is easy to show that a possibly empty range for an attribute x_j imposed by the predicates of B_i is $(a_{xj}, b_{xj}]$, or $[a_{xj}, b_{xj})$ where $a_{xj}, b_{xj} \in A_{F1F2}$. For this range to be empty: $a_{xj} \geqslant b_{xj}$.

Example:
Let F_1 be $(a \leqslant p) \wedge (b \geqslant q)$ and F_2 be $(a = x) \vee (b \geqslant y)$
The condition for F_1 to be contained in F_2 is easily seen to be $(q \geqslant y)$. The example helps in illustrating proposition 2. Here, $A_{F1F2} = \{p, q, x, y\}$.
F_1 is contained in F_2 if the following expression is inconsistent.
$F_1 \wedge \neg F_2 = ((a \leqslant p) \wedge (b \geqslant q) \wedge (a > x) \wedge (b < y)) \vee ((a \leqslant p) \wedge (b \geqslant q) \wedge (a < x) \wedge (b < y))$

$B_1 = ((a \leqslant p) \wedge (b \geqslant q) \wedge (a > x) \wedge (b < y))$.
$B_2 = ((a \leqslant p) \wedge (b \geqslant q) \wedge (a < x) \wedge (b < y))$.
For B_1 to be inconsistent: $(x \geqslant p) \vee (q \geqslant y)$
For B_2 to be inconsistent: $(q \geqslant y)$
Thus F_1 is contained in F_2 if:

$((x \geqslant p) \vee (q \geqslant y)) \wedge (q \geqslant y) \Rightarrow (q \geqslant y)$ □

In the worst case all *m* predicates in F_1 might have to be compared with all *n* predicates in F_2. Thus checking containment of an incoming filter with a cached filter requires *O (mn)* such comparisons.

The following observation about positive filters belonging to the same template can be made:

Proposition 3: *(Filters of same template)*
Let F1 and F2 be two positive LDAP query filters belonging to the same template. F1 is contained in F2 if each predicate in F1 is contained in the corresponding predicate of F2. ˥

Containment problem for filters of the same template having *n* predicates using Proposition 3 requires *O*(*n*) comparisons of assertion values. The algorithms described in the section can be extended for substring assertions by interpreting substrings as range assertions. An implementation of LDAP query containment algorithm was contributed to OpenLDAP [17], the open-source directory server, as proxy cache engine [16]. Proxy caching is included in the OpenLDAP 2.2 distribution.

5. Replica Consistency

LDAP does not standardize a replication protocol for keeping a replica in sync with the master copy, allowing individual vendors to use their own proprietary protocols. Section 5.1 describes the filter synchronization problem in LDAP while Section 5.2 discusses existing and proposed filter synchronization protocols for LDAP.

5.1. Filter synchronization

To support consistency in filter based replicas, a means of synchronizing *content* corresponding to a search request is required. Let the set of DNs corresponding to entries satisfying an LDAP search request, *S*, at instant *t* be $C_S(t)$. The set of entries at $t' > t$ is then given by:

$$C_S(t') = C_S(t) + \underbrace{E_S^{01}(t,t')}_{add} - \underbrace{E_S^{10}(t,t')}_{delete} \qquad (2)$$

where $E_s^{01}(t,t')$ is the set of DNs of entries moving into the content and $E_s^{10}(t,t')$ is the set of DNs of entries moving out of the content in the interval (t,t'). To obtain the content at time *t'* entries corresponding to these two terms must be respectively added and deleted from the content at time *t*. Additionally entries inside the content at *t* which are changed during this interval but remain inside the content (represented by $E_s^{11}(t,t')$) should be modified.

The last term in (2) requires the master server to reliably compute the set of entries which are deleted from the content of *S*. This requires history information to be maintained either in the form of change logs, tombstones (empty entries representing deleted entries) or a per-session history of entries leaving the content.

In the absence of complete history information, it is still possible to synchronize the content without a full reload being required. This can be achieved by the server returning DNs of all unchanged entries in the content and entries changed since *t* which match the search criteria at *t'*. Mathematically,

$$C_S(t') = E_S^{un}(t,t') + E_S^{01}(t,t') + E_S^{11}(t,t') \qquad (3)$$

where $E_s^{un}(t,t')$ represents DNs of the set of entries in the initial content which remained unchanged in the interval (t,t').

5.2. The ReSync protocol

The *persistent search* control proposed in [15] is a means of extending the LDAP search operation such that the operation does not end after all the matching entries have been sent. Instead the connection between the client and server remains open over which subsequent changes to the content are sent. While persistent search can provide strong consistency for filter based replicas, it requires a TCP connection per replicated filter which might not scale for large replicas. Polling is a better mode of update for information typically stored in directories.

The proposed mechanism is an extension of persistent search to support polling mode for synchronization. The client can specify the mode of update as polling or notifications and optionally specify a cookie. The following *resync* control is attached to a normal search request:

reSyncControl = (mode, cookie)

The server (master) handles a resync request from the client (replica in this case) as follows: (*i*) if cookie is null, it is the initial request in an update session and the entire content is sent, (*ii*) otherwise, the cookie is used to identify the resync session and content updates accumulated since the last request (stored as session history) are sent, (*iii*) further if the *mode* is "`persist`", the connection with the client is maintained on which any further change notifications can be sent, (*iv*) else if the mode is "`poll`", a cookie to resume the session is also sent.

Each notification/update PDU contains an entry along with a control specifying the *action* to be taken by the client. If the action is `add` or `modify`, the complete entry is sent, otherwise if the action is `delete`, only the DN of the entry is sent. Note that an `add` action representing an entry moving into the content could happen due to an add, modify or modify DN operation at the master. Similarly the `delete` action could take place due to an entry being deleted, modified or renamed.

When the server has incomplete history information, unchanged entries ($E_s^{un}(t,t')$ in (3)) are conveyed using the `retain` action. A session can be ended by the client sending a request with mode as "`sync_end`" or abandoning a persistent search. The server can time out sessions which have been inactive for more than an admin time limit.

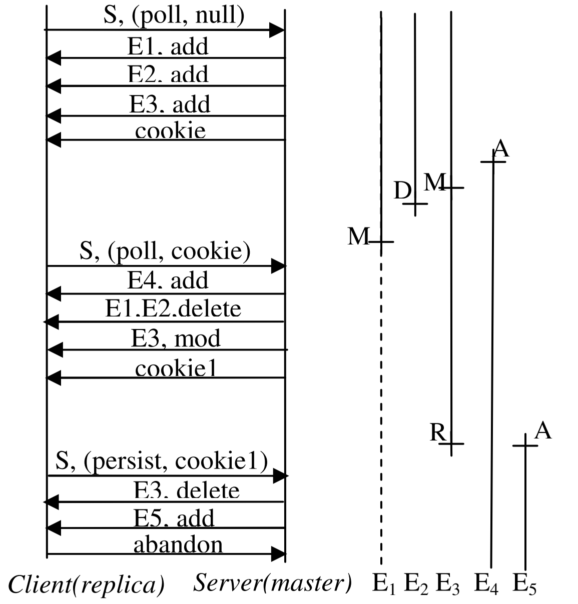

Figure 3: An example ReSync session

Figure 3 shows the message sequence chart for an example session. Vertical lines representing lifetimes of entries E_1, E_2…E_5 are also shown. The length of a line represents the life span of an entry. The duration that an entry is in the content of a search request *S*, is shown by a solid segment, while the duration which it spends outside the content is shown by a dashed segment. The symbols *A*, *M*, *D*, *R* correspond to the four update operations: add, modify, delete and rename (modify DN). Note that update corresponding to a modify DN which does not move an "in content" entry "out" is a delete action for the old DN (E_3) followed by an add action for the new DN (E_5).

ReSync protocol maintains session history of entries leaving the content. The alternatives (described below) either do not provide convergence or require unreasonably large history information and/or synchronization traffic. Some servers keep track of a deleted entry using a *tombstone* - a hidden entry that keeps track of the state, but not the data, of an entry that has been deleted. Similarly *change logs* [18] use the directory itself to store information about update operations.

A tombstone does not contain the original attributes of the entry, and therefore it is impossible for the server to determine if a deleted entry moved out of the content, thus requiring transmission of all deleted entry DNs since the last update. Change logs only contain information about the changed attributes. If an entry is first modified out of the content and then deleted, change logs are not sufficient to determine whether the entry moved out of the content. The ReSync protocol is lightweight and designed to reduce synchronization traffic while providing convergence guarantees.

6. Replica content determination

In subtree replication, a set of subtrees to be replicated is identified based on long term spatial locality of access patterns and configured to be replicated. Typically there is no provision for allowing dynamic changes to the set of replicated subtrees.

Similarly, for filter based replication, it is possible to find a set of generalized filters (Section 6.1), which capture the semantic and spatial locality of the access pattern and statically configure them to be replicated. However, since the replication unit is much smaller compared to subtrees, it is feasible to dynamically update the set of filters stored in the replica to improve hit-ratio (Section 6.2).

6.1. Generalizing filters

User queries typically return very few entries for them to be efficient units of replication. The meta-data size for queries like `(telephoneNumber=_)` will be comparable to the data size. Moreover, such queries will not take advantage of any spatial/semantic locality in the access pattern. However, generalized form of user queries can be used to represent frequently accessed regions.

The following guidelines for generalizing filters based on those in [12] have been used: (*i*) generalization based on attribute components, (*ii*) generalization based on natural hierarchy of filters. E.g. of generalized queries are `(telephoneNumber=261-758*)`, `(&(div=X)(dept=_))`.

6.2. Filter selection

In filter based replication, it is possible to dynamically adapt to changes in access patterns. Authors of [12] describe an efficient algorithm for improving hit-ratio from stored filters. It works by maintaining statistics for two lists of filters: the list of filters which is actually stored and a list of candidate filters. For each user query, the benefits of filters in both the lists are updated which might result in filters moving in and out of the lists. Updating of the actual list, when a new query arrives, is termed as *evolution*. If the benefit of the 'candidates' becomes larger than the 'actuals' by a specified amount, a *revolution* is initiated in which the two lists are combined and filters with the best benefits are chosen.

Using *evolutions* as described above requires frequent updates to the stored filter list and is thus not suitable for a replication scenario. The approach used in the case study of Section 7 is to maintain 'hit' statistics for candidate filters which is then used to perform a periodic update to the list of stored filters by selecting filters with the best *benefit* to *size* ratios. The benefit is defined as the number of hits for a candidate since the last update, while size is the estimated number of entries matching the filter. The interval between updates depends on the type of query. This is a simple means of approximating the expensive *revolutions* of [12].

7. Replica Performance

Effectiveness of partial replication in improving performance of directory enabled applications when accessed from remote locations is considered. Subtree and filter based partial replicas are compared on the basis of hit-ratio, update traffic and processing overheads.

7.1. Directory and workloads characteristics

The IBM enterprise directory containing more than half a million employee and organizational records has been used to evaluate performance of replication models. Each employee entry is approximately 6KB in size. The enterprise directory is used by hundreds of applications accessed from over 150 countries and different geographies. The problem considered was to use partial replication to improve performance for a geography containing nearly 30% employees.

Most directory queries are accesses to the following entities represented in the directory: people, departments and locations. The employees are organized in the directory on a country basis with all employees of a country appearing as children of the country entry. This is an example of a relatively flat namespace (Section 3.3). Similarly all department entries belonging to a particular division are placed under the division entry.

Table 1: Workload distribution

Type of query	*Approx % contribution*
`(serialNumber=_)`	58
`(mail=_)`	24
`(&(dept=_)(div=_))`	16
`(location=_)`	2

Real workloads for two days of accesses were considered. The distribution of query-types in the workload is given in Table 1.

7.2. Hit ratio comparison

(a) Serial number query: Figure 4 shows that the filter based model provides a hit-ratio of 0.5 with a replica size which is less than 10% of the total person entries in the directory. A subtree based replica can not selectively replicate employee entries from a country. The entries in a country are not accessed uniformly and semantic locality can be captured in filters of the form `(serialnumber=_*_)`. This is the reason for filter based replication performing better.

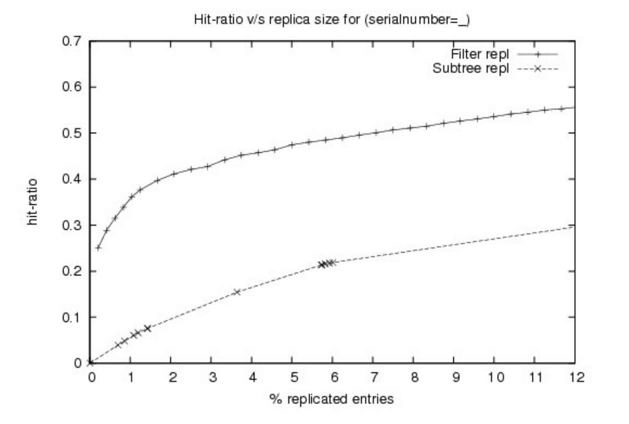

Figure 4: Hit ratio v/s replica size

(b) Department query: Not all departments in a division are accessed uniformly. While a filter based replica stores only the more beneficial departments, a subtree based replica can either store all or none of the department entries under a division. Since the replicated generalized queries of this type are smaller in size, the dynamic filter selection described in Section 6.2 can be used. Figure 5 shows the effect of reducing the revolution interval from 10000 to 6000. queries for filter based replication.

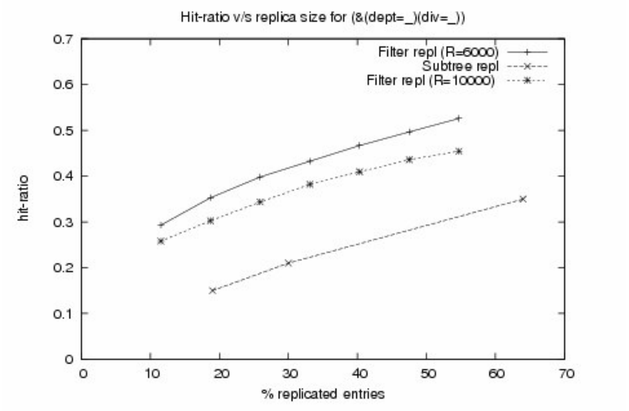

Figure 5: Hit-ratio v/s replica size.

(c) Other queries: Since the field <user> in <user>@ <cc>.xyz.com is not organized (unlike the fields in `serialnumber` attribute), filter based caching can not describe the access patterns efficiently for this case.

The access rate of location entries was seen to be high compared to the relatively small number of location entries. Thus the entire location tree can be replicated ensuring a hit ratio of 1 for this type of query while using a very small fraction of the total replica size.

7.3. Update traffic

For filter based replica there are two components of the update traffic, (*i*) the resync traffic corresponding to currently stored filters, (*ii*) traffic associated with bringing new filters to adapt to the access pattern. (*ii*) is attributed to the revolutions described in 6.2. This component is not present for the `(serialnumber=_)` query because generalized filters in this case could have thousands of entries, hence dynamic selection of filters is not performed.

(a) Serial number query: Figure 6 compares update traffic (in number of entries) for subtree and filter based replication for a given hit-ratio. The resync protocol is used by a filter based replica to reliably determine the minimal set of updates to be sent. Thus the higher update traffic for subtree based replicas is a direct consequence of the large number of entries stored for the same hit-ratio.

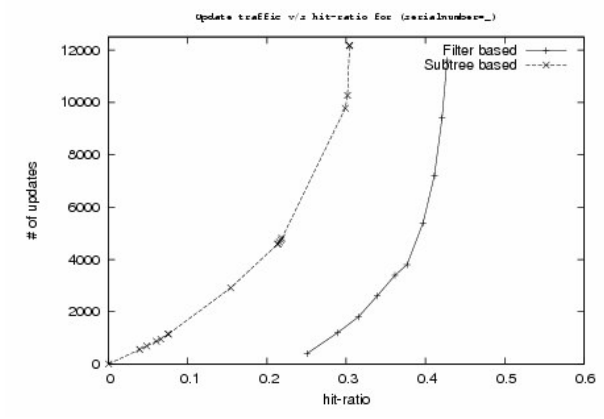

Figure 6: Update traffic v/s hit ratio

(b) Department query: Department entries in the directory have a very low update rate, thus the update traffic for subtree based replication is negligible. However updates for filter based replica are not negligible due to the second component of update traffic mentioned above. This component can be controlled by having larger intervals between revolutions as shown by the lower curve (R=10000).

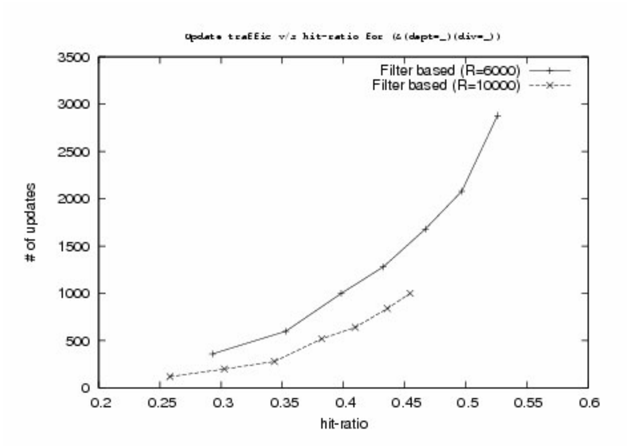

Figure 7: Update traffic v/s hit ratio

7.4 Query processing overheads

Additional query processing overhead incurred in filter based replication is directly proportional to the number of stored filters. Along with replicating generalized filters (templates) it is also advantageous to store recently performed user queries. However these queries are simply cached for a short time window and not updated. The three curves in Figures 8 and 9 correspond to storing only user queries, storing only generalized filters and storing both. Storing a window of last 50 queries gives a 20% hit-ratio. Since query hits are due to temporal locality in the access pattern the hit-ratio curve saturates after 100 cached queries.

From Figure 8, it can be seen that storing both generalized filters and user queries provides a hit ratio of 0.5 with just 200 stored filters for the `(serialnumber=_)` query. Since query containment in this case is a simple substring match, the processing cost is minor.

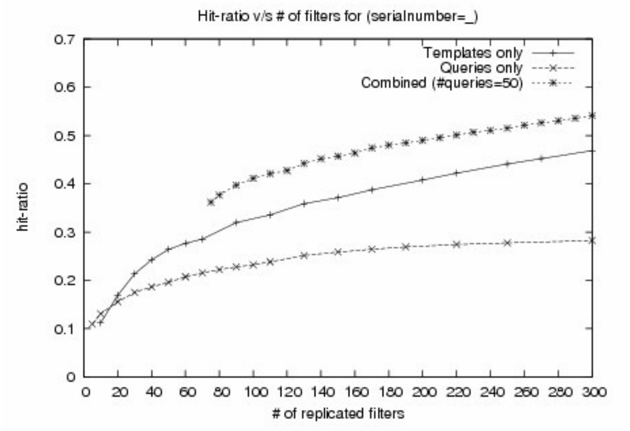

Figure 8: Hit ratio v/s # of filters

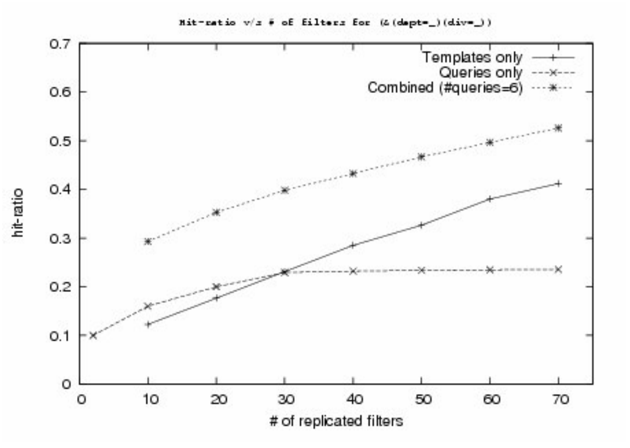

Figure 9: Hit ratio v/s # of filters

8. Conclusions

A new directory replication model based on filters has been proposed and compared with existing subtree based replication models. The model consists of a directory server replica storing entries and meta data corresponding to one or more LDAP filters. Replicated filters are generalized user queries which correspond to semantic regions demonstrating locality of reference. The concept of LDAP templates (query prototypes) has been introduced to reduce the complexity of the query containment problem. A filter synchronization protocol which uses standard means of extending LDAP is proposed. The protocol supports both polling and notification modes of synchronization. The session history based protocol reduces synchronization traffic and size of historical data to be maintained compared to existing techniques like changelogs and tombstones. Performance of the proposed model is evaluated and compared with subtree based models for a real enterprise directory using real workloads. A hit ratio (percentage of queries completely answered by the replica) of 0.5 is reported for the proposed model while replicating less than 10% of the employee directory for a typical query which requests an employee entry matching a unique ID. The update traffic is also considerably smaller than subtree based replicas. Filter based partial replication can be used to significantly improve performance of directory based applications.

9. References

[1] M Wahl, T Howes, S Kille, "RFC 2251: Lightweight Directory Access Protocol (v3)", http://www.ietf.org/rfc/rfc2251.txt.

[2] M Wahl, A Coulbeck, T Howes, S Kille, "RFC 2252: Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions", http://www.ietf.org/rfc/rfc2252.txt.

[3] T Howes, "RFC 2254: The string representation of LDAP search filters", http://www.ietf.org/rfc/rfc2254.txt.

[4] Q Luo, S Krishnamurthy, C Mohan, H Pirahesh, H Woo, B Lindsay, J Naughton, "Middle-Tier Database Caching for e-Business", ACM SIGMOD, 2002.

[5] Q Luo, J F Naughton, R Krishnamurthy, P Cao, and Y Li, "Active Query Caching for Database Web Servers", ACM SIGMOD Workshop on the Web and Databases, WebDB 2000.

[6] S Dar, M Franklin, B Jonsson, D Srivastava, M Tan., "Semantic Data Caching and Replacement", Proceedings of the 22nd VLDB Conference, 1996.

[7] P Deshpande, K Ramasamy, A Shukla, J Naughton, "Caching Multidimentional Queries using Chunks", ACM SIGMOD 1998.

[8] Q Luo, J F Naughton, "Form-Based Proxy Caching for Database-Backed Web Sites" VLDB Conference, Rome 2001.

[9] H V Jagadish, LVS Lakshmanan, D Srivastava, "Revisiting the Hierarchical Data Model", IEICE Transactions on Information and Systems, Vol. E00-A, No. 1, January 1999.

[10] H V Jagadish, LVS Lakshmanan, T Milo, D Srivastava, D Vista, "Querying Network Directories", ACM SIGMOD Conference, Philadelphia, PA, June 1999.

[11] S Cluet, O Kapitskaia, D Srivastava, "Using LDAP Directory Caches", Proc. ACM Principles of Database Systems,1999.

[12] O Kapitskaia, R T Ng and D Srivastava, "Evolution and revolutions in LDAP directory caches", Proceedings of the International Conference on Extending Database Technology (EDBT), 202-216, 2000.

[13] M Smith, "RFC 2798: Definition of the inetOrgPerson LDAP object class". http://www.ietf.org/rfc/rfc2798.txt.

[14] T Howes, M Wahl, A Anantha, "RFC 2891: LDAP Control Extension for Server Side Sorting of Search Results". http://www.ietf.org/rfc/rfc2891.txt

[15] M Smith et al "Persistent Search: A Simple LDAP Change Notification Mechanism", draft-ietf-ldapext-psearch-xx.txt, a work in progress.

[16] Apurva Kumar, "The OpenLDAP Proxy Cache" http://www.openldap.org/pub/kapurva/proxycaching.pdf

[17] OpenLDAP Project, web page: http://www.openldap.org

[18] G Good, L Poitou, "Definition of an Object Class to Hold LDAP Change Records", draft-good-ldap-changelog-xx.txt, a work in progress.