Solving names within X500

Hossam Afifi, Christian Huitema

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports de Recherche

N°1633

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

Solving NAMES within X.500

Hossam AFIFI
Christian HUITEMA

février 1992

Institut National
de Recherche
en Informatique
et en Automatique

Sophia Antipolis
B.P. 109
06561 Valbonne Cedex
France
Tél.: 93 65 77 77

Solving NAMES within X.500.
Résolution de noms X.500.

Hossam AFIFI, Christian HUITEMA
INRIA. Projet Rodeo.
BP 109. 06560 Valbonne.

Abstract

An easily accessible name service is an important asset for the management of a network, and a standard service like X.500 could enhance considerably the operation of heterogeneous networks based on the OSI suite. However, the naming scheme used by X.500 is complex, and not necessarily user friendly. In this paper, after a short presentation of X.500 operations and names, we will justify a proposition to use untyped "short forms", made of sequences of text tokens, and we will develop a set of algorithms for accessing the directory with these short names, before giving comparison and measures of these algorithms. This short form of names is compatible with the naming schema used in Internet, so that we could retain the simplicity and ease of use of the Internet naming system, while being compatible with the complete X.500 standard. Our solution is particularly useful for the category of names that can be found for example when interfacing mail to X.500 and where resolution time is critical due to queues of waiting queries.

Résumé

L'implantation de serveurs de noms dans les réseaux facilite aux utilisateurs l'interconnexion et rend transparents, les détails physiques des types de liaisons. X.500 est le serveur de noms dédié aux réseaux conformes au modèle OSI. Sa structure de noms est cependant complexe et donc lourde à manipuler.

Nous présentons dans ce rapport, une forme de noms abrégée et non typée regroupant ainsi la simplicité et la conformité au standard X.500. Cela mènera à l'élaboration d'algorithmes pour la résolution correcte et rapide de cette forme de noms suivie d'une évaluation des performances de chacun des algorithmes. Les applications de ce système s'étendent de la fonction simple d'annuaire à la résolution d'adresses de messagerie X.400 où le temps et l'efficacité de résolution sont critiques.

1 Names in the X.500 Directory

The X.500 series of recommendation defines the standard "Directory service" for OSI applications. Its principal point of interest is to be used in the OSI world as a nameserver or "white page services": given the name of an object, the directory can be accessed to retrieve networking properties such as addresses, capabilities or access rights. X.500 is a specialized, distributed, database application. In order to handle more easily the distribution of objects at various locations, a strict hierarchical structure has been adopted for names, or as the standard calls them, "Distinguished Names" (DN). A DN is composed of the sequence of the names of all the segments composing the path from the root of the "Directory Information Tree" (DIT) to this node. The "name" associated to one segment is called a "Relative Distinguished Name" (RDN), and is composed of a set of one or more "Attribute Assertions". An attribute assertion is the expression that some "Distinguished Attribute", identified by its type, has a certain "Value". An example of a Relative distinguished name could be <Machine = VAX, Model = 0202>. The data itself is called "entry information" and is located by the DN. It is composed also of attributes typed values.

To use the directory, a user or a program must assert the name of the entry, i.e. specify the complete sequence of "Typed name parts" that compose the "Distinguished Name". Most existing X.500 user interfaces request that the user fills up a screen form which lists a number of possible "naming attributes", e.g. name of country, name of organization or surname. However, several user have complain that this type of interface are slow, cumbersome and error prone. This user complaints can be traced to three main causes:

- The necessity to enter not only an attribute value but also an attribute type, requires more effort and attention,
- The necessity to follow completely the hierarchy, including those pathes which have been added for administrative reasons – like facilitating management – is very unnatural and not exactly "user-friendly",
- The complex structure of the names makes it in practice impossible to enter the full name on a single line and to memorize easily the names.

The last problem is very important in our context, as we want to use the directory as a nameserver: the calls to the name service will be embedded in the networking application. In practice this precludes the "interactive" error corrections algorithms that are common in the "user oriented" interface, and justify the development of innovative "name resolution" algorithms.

2 A short form of names

The main hypothesis that we made for the conception of these algorithms was that we would not require the user to "type" the names. This is a priori justified by two observations:

- Almost all name parts currently used in directory services have the same "case ignore string" syntax,
- The usage of names composed of "untyped tokens" is already very common at the user interface, e.g. in TCP-IP networks.

Note that we don't make any hypothesis on the "external representation" of the "list of name parts". In fact, when developing the algorithm, we knew that we would have to handle at least two notations:

- The "domain name" notation, used in the electronic mail applications following the RFC-822 specification [2], e.g:

  Dupond@sophia.inria.fr

- A "single line" notation, for use by the "SITA"[1] in "IATA" electronic mail service, e.g:

  fr/inria/sophia/Dupond, or *NCEYBDD*

  In the second form the first three characters represent a city, the two following characters represent a department and the two last characters are organization initials. A small treatment must be applied on this very short form to add a country name. This is done without the directory by mapping city names to corresponding countries.

It is quite easy to note that these two notations are semantically equivalent. They differ by the choice of a little endian logic in the RFC-822 case, and a big endian logic for the other case, and by the use of dots and at-sign versus slashes, but in any case they can be parsed as a sequence of text tokens, e.g. using an ASN.1 notation:

```
example-name SEQUENCE OF text-token
::= { ''fr'', ''inria'', ''sophia'', ''Dupond''}
```

A similar approach to the use of "untyped" names has been proposed by Steve Kille in [10]. He proposes to use a new syntax, based on a little-endian ordered sequence of optionally quoted text tokens separated by commas. Using his conventions, our example would be noted as:

Dupond, sophia, inria, fr

In the rest of this paper, we will develop a set of algorithms for locating in the DIT an object which is identified by a sequence of text tokens. The hypothesis, the algorithm makes on the naming scheme, is that each segment of the path must be identified by exactly one attribute assertion. The algorithms that we develop are based on the "abstract syntax" of a name presented as an ordered sequence of text-tokens , and they are not related to any external form.

[1] Société Internationale de Télécommunications Aéronautiques financing this work.

3 Comparing different search algorithms

The name resolution algorithms that we propose will be implemented by using the X.500 services through a "directory user interface" (DUA). They will use the `Read` and `Search` operations:

- The `Read` operation takes a complete X.500 name and returns the data informations attached to this name.
- The `Search` operation takes an X.500 name (called base object) and a logic filter. It returns the X.500 names of the entries locating under the base object in the DIT which match this filter. The "scope" of the operation can be parametrized to search for the base object only, or its direct children, or the whole subtree under it.

The directory service interface provides two other access operations, `Compare` and `List`. We will not use these operations:

- The `Compare` operation only aims at verifying the value of one attribute of a named entry, and is not very useful for name resolution purposes,
- The `List` operation provides a listing of the RDNs of all the entries directly attached to a base object. This operation is a compromise between `Read` and `Search` operations. It has nearly the cost of a read. The reason we can't use it is due to the fact we have observed that a number of organizations in our networking environment intend to restrict access to the `List` service: they don't want unauthorized outsiders to be able to get a full listing of their internal hierarchies. An algorithm that would rely on the `List` service would fail in these organizations.

We will compare the proposed algorithms by measuring their user friendliness – i.e. their flexibility – and their "cost". Since the DSA will be receiving instantaneously hundreds of requests, it is preferable to minimize their cost as much as possible. The cost of name resolution in X.500 depends on the name to find and on the algorithm used, i.e. on the combination of operations like read or search that the algorithm requires: a read operation is faster but less powerful than a search. The search operation, by definition, must consult the whole subtree under the base object, and its cost will vary with the size of this subtree. This job is certainly more expensive than a normal read operation. In fact, the balance between the cost of a "read" and that of a "search", for example, is heavily implementation dependent, and we can assume two extreme cases:

- The simpler directory servers (DSA) will perform a search by applying the filter sequentially to all entries in the subtree. The cost of the search will thus be N times the cost the read, where N is the size of the subtree.
- More sophisticated DSAs will manage "inversion tables" so that a filter which references "inverted attributes" can be solved by a direct access to the database, independently of the size of

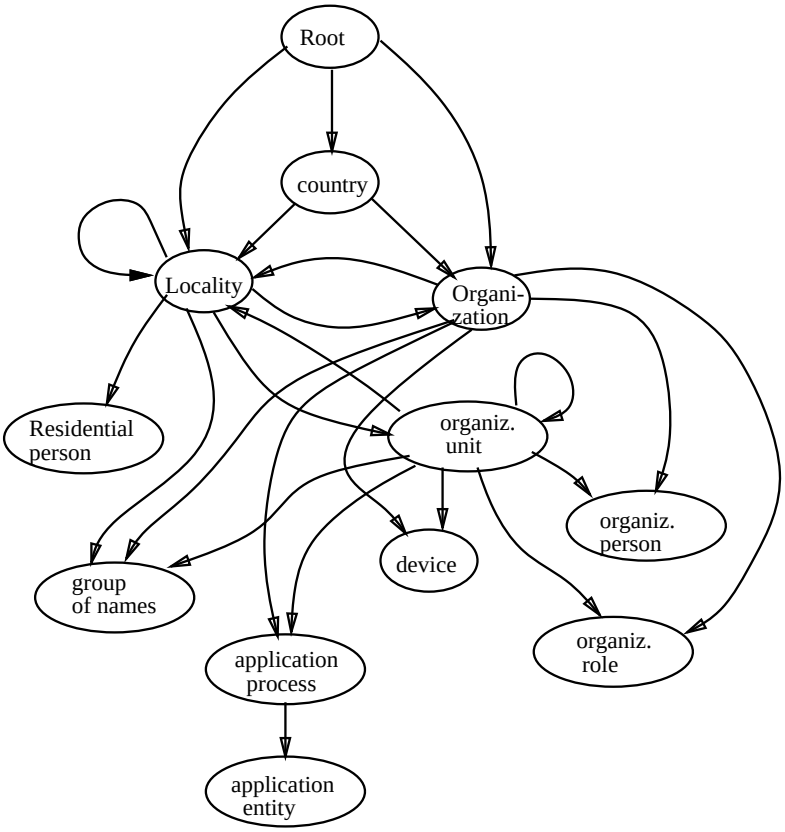

Figure 1: The recommended schema for names

the subtree. We will assume in our evaluations that the cost of such searches is approximately twice the cost of a read; this figure is backed with experimental data obtained on our DSA implementation [1].

We developed successively three algorithms for solving our textual names, which we called "direct search", "successive searches" and "read and search".

4 The "Direct Search Algorithm"

In order to perform a directory operation, we will have to "type" our untyped tokens. This "typing" requires a knowledge of the "naming schema" used in the DIT. One of the difficulty that we have to solve is that the naming schema is very flexible, and that our knowledge of it is necessarily very partial. Knowledge of the schema should be programmed in the algorithm; by default, we will assume that the shema follows the recommended name forms defined in X.500 (see figure 1).

The first algorithm that we tested tried to make as few assumptions as possible on the schema: we only assume that the most significant name part would be a "Country Name", and the least significant a "Common Name". Using the country name as "base object", we make a Search operation looking for an entry matching the "Common Name". Then, if the search returned a set of names, we will only retain those names which included all the text-tokens of the "purported name": we will compare locally the text tokens with all of the values present in the attribute values composing the relative distinguished names, filtering out all "homonyms" which would bear the same common name in different organizations or different localities.

This solution has one big advantage, its extreme simplicity: only one operation is requested to the directory service. It has however two disadvantages:

- It is extremely unrealistic to assume that one could apply a search on all entries within a country. The directory service is designed to be distributed, and one should assume that each organization will provide its own directory. Searches performed at the "country" level will have to be "broadcasted" to all organizations within the country, and "inversion tables" cannot be used at the country level. This would result in a very excessive cost, and the directory service is very likely to refuse to perform such operations.
- The algorithm only compares the text tokens with the "distinguished" attribute values. It will thus report an error if an alternative name, or an alias, had been purported.

We clearly have to refine this simplistic algorithm, in order to make it less costly and more effective.

5 Using "Successive searches"

We saw that resolution with one single search was incorrect. A possibility to solve the problem is to use a sequence of searches, solving one name part at a time. This algorithm is essentially similar to

the resolution algorithm proposed by Steve Kille in [10], with however one major difference: if there are ambiguities, we will not allow ourselves to "query the user". The software that we develop has to be used in mail gateways, and the operation should be completely automatized; refusal of the message will be preferred to misdelivery.

The algorithm aims at transforming the set of untyped text tokens into a fully qualified DN. It will use as many stages as the number of tokens in the purported name; at any stage, the set of possible solutions will contain one or several "base objects names" resulting from the resolution of the "most significant" tokens, and a set of "unmatched tokens". At each stage, we will perform the following:

- Build up a "search filter" from the most significant unmatched token. As we only assume a very partial knowledge of the "naming schema", the search filter will contain as many alternatives as possible attribute types. For example, if we are trying to solve:

  Dupond@sophia.inria.fr

  and have already resolved *inria.fr* into

  C=FR; O=INRIA;

  the filter for the next stage will be:

  ```
  O == "sophia" || OU == "sophia" || Locality == "sophia"
  ```

  i.e allowing the presence of attributes of type "organization", "organizational unit" or "locality". If the token had been the least significant within the purported name, we would also have allowed the types "common name" and "surname". The initial query, at the root level, should only look for the the attribute types "country" and "organization".

- Apply this filter successively to all the base objects resulting from the resolution of the previous phase. The set of names resulting for each of these searches will constitute the base object for the next phase. If this set is empty, the query has failed.

The set of names resulting from the last stage is the solution of the name resolution. As we need to converge on exactly one name in our gateway operations, we will reject the message if the set contains more than one name.

There are a number of possible variants in the implementation of this algorithm. For example, if a given stage results in an empty set of solution, it is possible to "loosen the filter", and to repeat the operation with an "approximate match" or a "substring match" condition rather than the simple equality test mentioned here. We did not want to implement it in our "automatized" context for two reasons:

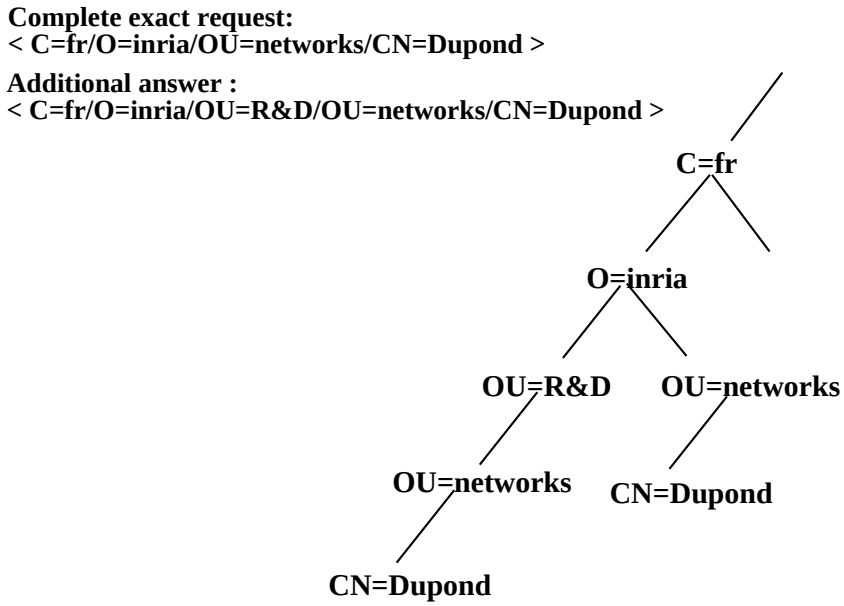

Figure 2: A case where successive searches return a wrong answer

- Loose matches are much more expensive to perform than equality tests: it is almost impossible to implement "inversion tables" for the substring test, and all entries must be tried. Approximate matches can be handled by inversion tables, but they tend to result in large lists of possible solutions, which will all have to be tested in the next stage of the algorithm.
- Loose matches are ... loose, which results in an increased probability of selecting "the wrong entry". This may be acceptable for a user driven interface, but could not be allowed in our context.

Another variant concerns the scope of the searches applied at each stage. If we limit the scope of the searches to the direct children of each base object, we will oblige the users to give exactly one "text token" for each "RDN", i.e. for each level of the naming tree. This is certainly not user friendly; as the users do not necessarily have a complete understanding of the naming hierarchies within remote organizations, we have to allow for "partial names", where some intermediate components may be left unspecified. This is very easily implemented by setting, at each stage, the search scope to the "whole subtree", but in some cases, a user may be sure of his untyped query and he would obtain unneeded results. For an automated name resolver expecting a single result, this could block the application process calling the service (figure2); one could probably argue that example like this one mostly result from poor naming practices.

6 The "Read and Search Algorithm"

The most important problem of the "successive search" algorithm is its cost: Suppose a typical four token name such as:

Dupond@sophia.inria.fr

Its resolution will require four successive searches. We could assume that inversion tables would be available all over the place, and that the set of solutions will only include one component at any stage: taking "one read" as the unit cost R, and assuming that inversion tables are available and that the cost of a search is only $2.R$, then the cost of the algorithm would be $8.R$. However, one will immediately observe that the first search is performed at the root level: this is a very undesirable feature, as it would put an extra burden on the root DSAs. Similarly, one can observe that the second search is performed at the "country" level, which may well imply a broadcasting of the query towards all organizations DSAs within a country; the figure of $8.R$ is thus extremely optimistic. The "read and search" algorithm takes advantage of the partial knowledge of the schema that we possess in order to generally speed up the resolution.

The first step of the algorithm consists in building a complete name by making an hypothesis on the attribute types derived from "dominant schema". For example, as the dominant name form in R&D networks consists of:

- a country name,
- an organization name,
- a variable number of organizational units,
- a common name

We can "a priori" make the hypothesis that the "most probable" solution for our example name is something like:

```
< C=FR; O=INRIA; OU=SOPHIA; CN=Dupond; >
```

Indeed, the local agent can be made reasonably intelligent. For example, it could imagine that when the most significant token cannot be recognized in the table of countries defined in the ISO, then one should use the second most dominant name form, which consist of an organization name followed by a variable number of organizational units and a common name. It could even incorporate extra knowledge on the dominant forms in various parts of the world, e.g. that one should expect the name of a "state" to follow the country code "US".

In any case this first step should only be considered as a "best guess". We will verify it by using it as the key to a `Read` operation. Should this operation succeed, the query would be resolved and

we will be pleased. It could however fail, and return an error coded "NameError", for a number of reasons, e.g. because we guessed the wrong attribute types or because the attribute value purported by the user is not the "distinguished" value. However, the error report includes parameters [4]:

```
NameError ::= ABSTRACT-ERROR
       PARAMETER SET {
              problem [0] NameProblem,
              matched [1] Name}
```

The "matched" parameter contains the best match that was obtained when trying to resolve the key of the `Read` operation. The main purpose of the operation was in fact to obtain this parameter, so that it could be used as the "base object" for a search: we intend to perform, from that point, a "Direct search", using essentially the algorithm described in section 4. Suppose for example that after the read request, the "matched" parameter contains:

```
< C=FR; O=INRIA; >
```

We observe that this name matches the two most significant text tokens, "fr" and "inria" (we will address in section 6.1 the problem that could arise if the matching resulted from "aliasing"). We only have now to match the remaining least significant tokens, "sophia" and "Dupond". We could indeed at this stage engage in successive searches – using one search per remaining level of hierarchy – but we prefer to minimize the total number of searches by looking first for the last token, supposedly the most discriminant. We can build up a filter based on this last token:

```
CN == "Dupond" || Surname == "Dupond" ||
O == "Dupond" || OU == "Dupond" || Locality == "Dupond"
```

and, using the "matched" parameter as the base object, perform a search operation. That search will return a set of names, e.g:

```
< C=FR; O=INRIA; Locality=Rocquencourt; CN=Francis Dupond; >
< C=FR; O=INRIA; Locality=Sophia-Antipolis; CN=Michel Dupond; >
```

As explained in section 4, we have now to filter out homonyms by checking that all text tokens present in the original name have been matched. We can perform this operation:

- either implicitly: by definition, all the entries which matched the search filter matched the least significant token "Dupond",
- or by looking at the name parts (this is not useful in our example),
- or by doing supplementary operations, as for the token "sophia", which does not match any of the RDNs.

The purpose of the "supplementary operations" is to find the missing tokens, which can be located in any of the entries in the hierarchy. For each of these missing tokens, we could for example do a search for the filter:

```
O == "sophia" || OU == "sophia" || Locality == "sophia"
```

in the various levels of hierarchies:

```
< C=FR; O=INRIA; Locality=Rocquencourt; CN=Francis Dupond; >
< C=FR; O=INRIA; Locality=Rocquencourt; >
< C=FR; O=INRIA; >
```

for the first element in the set, which will have to be discarded, and then for the next element:

```
< C=FR; O=INRIA; Locality=Sophia-Antipolis; CN=Michel Dupond; >
< C=FR; O=INRIA; Locality=Sophia-Antipolis; >
```

which will be accepted, as the alternative attribute value `Locality=Sophia` was listed in the entry. Each of this searches will have a scope limited to exactly one entry, and a cost equivalent to that of a single read operation; in fact, we will use a read operation and perform the comparisons within the DUA. Moreover, as we know that we may have to perform these comparisons locally, we will also request that the search operations returns, in complement to the names of the matching entries, the values of the attributes of type *Locality, Organization* and *Organizational unit* present in these entries.

The cost of the name resolution, following this algorithm, can be decomposed as follow:

1. An initial `Read` operation,
2. When this operation result in a `NameError`, a following `Search` operation,
3. For each of the tokens which remain to be matched, and which are not also matched by the attributes of type *Locality, Organization* and Organizational unit of the result, up to one `Read` per result and per intermediate level of hierarchy.

The following table shows the cost of our query under the two hypothesis for DSA software – with or without inversion, and a comparison with the "successive searches" variant:

Operations for N entries	Cost (inversion)	Cost (linear)
Read and direct search		
Initial read	R	R
Search for last token	$2R$	NR
3 intermediate reads	$3R$	$3R$
Total	$6R$	$(N+4)R$
Read and successive searches		
Initial read	R	R
Search for "sophia"	$2R$	NR
Search for "Dupond"	$2R$	$\frac{N}{2}R$
Total	$5R$	$(1.5N+1)R$

The algorithm minimizes the number of searches – at most one – and is significantly faster than the "successive searches" variant when the DSA does not optimize searches by use of "inversion tables". The costs are approximately equivalent when the "inversion" technique is used. This is not very surprising as the the cost of the first and second step of the algorithm are noted here as equal: searching for "sophia" in the whole database is equivalent to a search for "Dupond" – we are in fact comparing here an arbitrary number of `Read` operation with a single `Search`. On the other hand, we should note that the direct search is much less "variable" than the successive searches for many reasons:

- Whenever the last token is fully qualified, e.g. set to *"Michel Dupond"* instead of just *"Dupond"*, the set of answer to the direct search will contain just one response, reducing the cost of the "discrimination" pass,
- When the naming practices of the organization are "flat", e.g. when the "Organizational Unit" component is not present in the hierarchy but when its name is instead stored as one attribute of the entry, the first of the successive searches is likely to result in a very large number of matches which will all have to be "explored" in the next step,
- The hypothesis that an optimized `Search` costs no more than twice a `Read` is really a "best case". We cannot predict the characteristics of remote DSAs, and should not rely on remote optimizations.

In short, each of these reasons would be sufficient to tilt the balance towards the "Direct Search" approach; the combination of all of them is overwhelming.

6.1 The effects of aliasing

One of the key elements of the "Read and Search" algorithm is the usage of an initial "Read" operation to obtain a "base object" for later searches. In order to properly disambiguate homonyms, we must be able to determine which tokens were matched by the "base object", or by levels of hierarchy above this "base object".

A particular problem occurs when the "matched" parameter of the "NameError" is an "alias". For example, our attempt to read the name:

```
< C=FR; O=INRIA; OU=SOPHIA; CN=Dupond; >
```

could have resulted in a "matched" parameter set to:

```
< C=FR; Locality=Valbonne; O=INRIA, Sophia-Antipolis; >
```

If we look for matched name parts, we only find one – the country name. In fact, even that is not necessary. For example, an attempt to read the name

```
< C=FR; O=Vraiment Grosse Multinationale; OU=Support; CN=Gaston Lagaffe; >
```

could have resulted in a "matched" parameter set to:

```
< C=US; O=Mega Big International; OU=French Division; >
```

where the relative distinguished names bear absolutely no relation with the initial purported name. Both situations are characterized by the fact that the "matched" parameter contains name parts which were not present in the original query.

One simple way to solve this problem is to use some complementary reads, trying successively:

```
< C=FR; O=Vraiment Grosse Multinationale; OU=Support; >
< C=FR; O=Vraiment Grosse Multinationale; >
```

until one the `Read` operation succeeds. At that point, we know that the tokens which composed the key of the `Read` have been matched, and we can use the `Distinguished name` returned by this operation as base object for the next stage of the algorithm.

7 Applications

The use of algorithms for untyped names forms resolution is mainly proposed to replace the actual *finger* [9] TCP/IP system call. We see clearly that this operation does not offer enough reliability for users. Moreover, its use is restricted to TCP/IP world and does not offer any flexibility in naming or partial name resolution. No hierarchy is permitted under user names and searches can only be done on name parts. Another accessible nameserver "the DDN Network Information Center" is also

	name with missing parts	complete name	name with parts in entries	name with aliases
One normal read op.	—	0.9 sec	—	—
Successive list op.	—	5.3 sec	—	—
General solution	3 sec	3 sec	4.4 sec	7.3 sec
without inversion	30 sec	30 sec	32 sec	1 min 30 sec

Figure 3: Comparison of four resolution algorithms

very heavy to use and so rarely updated. Since users and hosts informations must be copied locally in the nameserver administration, query results are often obsolete. Solving names for a finger-like application does not necessitate all the complicated algorithms developed, a direct search would be enough. The read-search algorithm has been however developed according to the needs of the SITA to cover most probable cases of figures that could face a nameserver when interfaced to administrations running with different naming policies.

8 Some measures

We gave without any comment the resolution times for different methods. We shall now compare them and give the time response obtained. The results hereafter were obtained using "PIZARRO", an X.500 Directory Server developed at INRIA (Sophia Antipolis). The name resolution and the DSA were on two separate computers on a LAN. The DSA had 2000 entries. We did not use chaining otherwise results would have depended too much on transient network conditions. The results in the table are .5 sec more than the calculated reslut due network overhead and print screen functions calls.

The results appear in figure 3. The successive list operations result is only for comparison since we do not recommend the use of this algorithm. The general solution corresponds to the read and search algorithm. We conducted the trials for four different names:

- A name with "missing parts", corresponding to a reasonable "user perception": jean/NCE/AAZ. This name can only be solved by the "general" algorithm.
- A complete name, where all tokens are present. Once solved and fully typed, this name is: C=fr/O=AAZ/OU=NCE/Locality=NYC/CN=jean.

- A name where non distinguished attributes have been added; these name parts must be compared to values stored in one of the entries in the path,
- A name where the path includes an alias.

9 Name resolution extensions

The search algorithm could be extended to cover not only "untyped" tokens, but also descriptive names [5], i.e. a set of RDNames that describe a name in which RDNs are unordered and may be uncomplete. Fortunately types are here to guide the resolution; proceeding as previously, we search for the country attribute type, if there we must find the organization or region attribute else we insert default values. So we arrange RDNs in a hierarchical manner following a schema like (figure 1). We make a search operation with the last RDN in hierarchy and the comparisons with remaining tokens. One could possibly ask about the solution if we find two attributes of the same class in the hierarchy. The answer is simple. Semantically it is wrong to give in a set two same types unorderly. There could be in this case two or more solutions that are exact. This is not permissible for a name that must be unique. Since from the beginning of this paper we try to overcome lacks or unvoluntary abstractions in names, we can make exhaustive trials with all possibilities i.e. we permute orders of ambiguous type and iterate the operations.

10 Conclusion

The figures speak for themselves: provided an efficient search facility, we can solve sensible names in about three times the duration of a "read" operation, and we can also use non distinguished attributes to identify the entry itself, or some components of the hierarchy.

We wish to introduce by this article the concept of untyped hierarchical naming in X.500. The reason is that most networks use it and that it is the most logical aspect a normal user can adopt.

As we see, there is no optimal unique solution to the problem. It is not good to put all options at the service of an application that will use the majority of time the same naming schema.

In the most general case the given name will be exact but untyped. It is not necessary hence to include complicated techniques. They will be aside procedures called when needed. We note also that the user friendliness is obtained at the expense of server load – but isn't that the general justification for more processing power?

References

[1] Hossam AFIFI. PIZARRO, an X.500 Directory Service. *Proceedings of the Second Annual AEC'91 Conference.*

[2] David H. Crocker, Standard for the Format of ARPA-Internet Text Messages. Dept. of Electrical Engineering, University of Delaware, Newark, DE 1971.

[3] Olle Jarnefors, Jacob Palme. Expert Contribution, Proposal for a concise format for MOTIS/X.400 address ISO/IEC JTC 1/SC 18/WG 4.

[4] CCITT RECOMMENDATIONS X.500-X.521. Data communication Networks: DIRECTORY SERVICE, CCITT Blue BOOK, Vol. VIII - Fasc. VIII.8 Melbourne, 1988.

[5] Gerald W. Neufeld. Descriptive Names in X.500. *Proceedings of the SIGCOMM'89 symposium, Communications, Architectures and protocols.* Austin, Texas Sept 19-22, 1989.

[6] CCITT RECOMMENDATIONS X.400-X.430. Data communication Networks: Message Handling Systems CCITT Red Book, Vol. VIII - Fasc. VIII.7 Malaga-Torremolino, 1984.

[7] Christian Huitema and Anne-Marie Bustos. Using a standard directory as a name server within the thorn project. *IFIP, Message handling systems and distributed applications* Costa Mesa, California .

[8] Douglas Brian Terry.(CSL-85.1) *Distributed Name Servers: Naming and Caching in Large, Distributed Computing Environments.* Xerox corporation Palo Alto Research Center. 3333 Coyote Hill Road,Palo Alto California 94304.

[9] Harrenstien, K., NAME/FINGER Protocol, 1977 December 30, RFC 742.

[10] S.E. KILLE, Using the OSI Directory to achieve User Friendly Naming. *Internet-draft, University College of London-March 1991.*