Evaluation of attributed names

To cite this article: Y Ni and A Goscinski 1995 Distrib. Syst. Engng. 2 87

View the article online for updates and enhancements.

Related content
- Collaborative naming in distributed systems
  S Benford and Ok-Ki Lee
- The distributed computing environment naming architecture
  N Leser
- KTK: kernel support for configurable objects and invocations
  B Mukherjee, D Silva, K Schwan et al.
Evaluation of attributed names*

Y Ni†‡ and A Goscinski§

School of Computing and Mathematics, Deakin University, Geelong, Victoria 3217, Australia

Received 17 October 1994, in final form 21 April 1995

Abstract. In this paper a new and original name evaluation method of attributed names is proposed and elaborated on. This method allows an attributed name to be presented in a non-hierarchical fashion and to be evaluated using a hierarchical approach. A three-level name model that supports attributed names for objects in a distributed system and the structure of naming contexts which bind attributed names onto objects are presented. The naming contexts are represented intuitively with a context-graph. To be able to meet the requirements generated from the name service semantics, a name evaluation model suitable for different name services, i.e. the conventional service, the selection service and the enquiry service has been developed. This model allows the name evaluation to be implemented efficiently. Moreover, to achieve the effectiveness and the efficiency in the implementation of the name evaluation, we have developed a set of algorithms, each of which is for a particular name service.

1. Introduction

In most existing operating systems (centralized and distributed), object names are organized in a hierarchical structure (Hulstena 1991). Therefore, most existing naming facilities support partitioned names which are of hierarchical structure. A partitioned name consists of a series of flat names which are delimited by a special separation token, such as '/' or '.'. There are many examples of partitioned naming schemes, e.g. the UNIX path-name system and the Internet Domain Name Service (DNS). A partitioned name denotes a path from the top level of the hierarchy to the level associated with the object identified by the partitioned name. This partitioned naming convention reflects the fact that the information about an object given by its name is the structure of directories in which the name is valid. This approach guarantees the unambiguity of names and minimizes the complexity of name evaluation. However, though names incorporate knowledge of the order of the components of the hierarchy, they hardly represent the information which describes the object itself. Partitioned names are not user-friendly. They have very strict hierarchy order and format. They are never obvious, nor natural. People have to rely on human memory as an integral component in the organization of computer-based object names.

To improve user-friendliness of object names, attributed names, also known as descriptive names, have been proposed (Bowman 1990, CCITT 1988, Goscinski 1991, Goscinski and Indulska 1992, Neufeld 1992, Peterson 1987, Peterson 1988 and Vance and Goscinski 1989). An attributed name consists of a set of attributes, each of which provides a piece of information about an object or describes a particular characteristic of it (Goscinski 1991). Such an attributed name can allow users to refer to an object by giving a description of the types and properties of the object they are looking for, instead of giving the precise path name for a name server to locate the object.

Attributed naming was initially proposed to implement name services for network users and organizations. These name services were aimed to support the e-mail system and for the purpose of network resource and server discovery (Schwartz 1990). Thus, such attributed names only apply to the users of the network or the resources and services in the public domains, e.g. White Pages and Yellow Pages services. The initially proposed attributed naming has been extended to file systems (Gifford et al 1991, Mclooneen and Sochrest 1992 and Mogul 1986). However, these proposals are based on existing centralized operating systems such as Unix or network operating systems, e.g. SUN Network File System. Extended attributed naming which covers all manageable objects (resources, services and files) of a distributed operating system has been proposed (Goscinski 1993, Goscinski and Haddock 1991, Goscinski and Indulska 1992 and Vance and Goscinski 1989). This attributed name scheme known as the RHODOS attributed name scheme, is supported by the RHODOS naming facility, which provides a service at the distributed operating system level (Gerrity et al 1991).

It has been decided that RHODOS supports a three-level naming structure, with appropriate mapping functions between them (Gerrity et al 1991). These levels are (1) well-defined attributed names (user names, denoted UNames) which are presented by a set of attributes in the

---

* This work was partly supported by Australian Research Council under Grant A492323429 and the Deakin University Research Grant 650404151.
† Current address: CSIRO, 723 Swanston Street Carlton, Victoria 3053, Australia.
‡ Email address: ying.ni@mel.dit.csiro.au
§ Email address: ang@deakin.edu.au
form of ASCII strings, for users; (2) structured and fixed-length numerical names (system names, denoted SName) which are presented by a data structure which contains certain fields to describe the object, for the operating system; (3) physical names (locations, object addresses) which are the physical addresses of the objects, for the purpose of inter-computer communication. The RHODOS naming facility finds a system name for a given user name, which is known as name evaluation. The communication sub-system (not the naming facility) maps system names onto locations, which is known as address resolution. The RHODOS names and mapping between these are shown in Figure 1. Both the mapping between UNames and SNames, and the mapping between SNames and object addresses is one-to-one.

To be able to support an attributed naming scheme, an effective and efficient evaluation mechanism is necessary. The name evaluation involves extracting components from a purported name, and matching the components with the data stored on the naming database or directories. Evaluating a partitioned name is straightforward, as the name is coupled with the naming hierarchy. Thus, to evaluate a partitioned name successfully, it is required that users enter a name which equates to a path in the naming hierarchy. The evaluation is carried out in the order set up by the naming hierarchy. This implies that users have to remember all the components that make up a name, and this order of components is used to retrieve the named object.

The attempt made by the RHODOS attributed naming scheme is to decouple attributed names from the name hierarchy and to relieve ordinary users from remembering a strictly formatted name. The main concerns placed on our development of the RHODOS name evaluation mechanism are: (1) an algorithm adopted to implement name evaluation is able to find a system name correctly using a user level name which is incomplete and/or with arbitrary ordered attributes; (2) if the information given is incorrect, the name server should recognize that fact rather than interpret it as a system name of another object; (3) the algorithms adopted should be very fast, because the performance of the name evaluation mechanism has an influence on the performance of the naming facility and, furthermore, on the overall performance of the operating system.

Two approaches to attributed name evaluation have been reported (Yeo et al 1993). One approach is that a naming facility resembles a relational database. An attributed name is evaluated by making a query to the database, e.g. the Univers Name Server (Bowman 1990). This approach allows the attributes in the purported name to be presented in any order. Another approach treats a purported attributed name like a partitioned name. The order of attributes in the name dictates where each attribute is to be evaluated according to the pre-defined attribute order, e.g. the naming scheme used by X.500. At each stage of the evaluation, an appropriate name context is established to carry out a meaningful matching (Neufeld 1992).

We propose a novel approach to attributed name evaluation which combines these two approaches. It allows a name to be presented in an unordered way, as in Univers, and to be evaluated as efficiently as if the name were a partitioned name.

In this paper, we present our study on attributed name evaluation and report on the development of the RHODOS name evaluation mechanism. A stand-alone prototype of the naming facility which achieves efficient and effective attributed name evaluation has been developed. By developing the prototype, the design concepts of name evaluation and the naming facility have been verified and the algorithms employed for name evaluation have been assessed. The implementation aspects of such an attributed name service are discussed.

The remainder of this paper is organized as follows. The RHODOS attributed naming concepts are discussed in section 2. Section 3 introduces the general issues of attributed name evaluation, and the name evaluation model which form the framework of the RHODOS attributed naming facility. Implementation of attributed name evaluation is presented in section 4. In section 5, a proposed main-memory specialized naming database implementation for the RHODOS naming service is briefly discussed. In the last section, the conclusion of the project is given.

2. RHODOS attributed naming

In the RHODOS system, the complete set of resources and services available can be treated as an accessible collection of objects. Furthermore, objects can be classified in terms of the services they provide and/or the features they possess. Objects that provide the same set of services and/or possess the same sort of features demonstrate the same property. Note, that in many cases, a user is interested in the first instance, in a service rather than in an resource that the service is built on. This was the reason to define objects based on their properties (Goscinski and Haddock 1994).

In order to define a unique user name, UNNAME, of an object it is proposed to provide a full set of attributes, denoted AttrSet, which describes the object, and define an order upon these attributes, denoted Ord. Thus, a user name of an object, UNNAME, is an ordered set of attributes of the following form

\[ \text{UNNAME} = (\text{AttrSet}, \text{Ord}) \]

An SName is a fixed-length numerical representation of an object used by the operating system to refer to the object. Formally, the notation of the relationship between an SName and a UNNAME is as follows:

\[ \text{SName} \leftrightarrow \text{UNNAME} = (\text{AttrSet}, \text{Ord}) \]

where ‘\( \leftrightarrow \)’ denotes a one-to-one relationship.
2.1. Attributed names

Each RHODOS object is known and identified by its attributes, where each attribute describes a property or a feature possessed by the object. A RHODOS attribute is a triplet:

\[ \text{attribute} = (\text{attribute.tag}, \text{operator}, \text{attribute.value}) \]

where \text{attribute.tag} is a string which specifies the type of the information given by that attribute, \text{operator} is an operator for which meaning is associated with an \text{attribute.tag}, and \text{attribute.value} is a string which gives a particular instance of the type of information indicated by the \text{attribute.tag} (Goscinski 1993).

Definition 2.1. A set of attributes \( a \), such that each attribute is known and registered by the system, is the AttrSet of an object if all attributes for that object are in \( a \) and every attribute in \( a \) is an attribute for that object.

That is, a AttrSet is a set of attributes which uniquely describe an object:

\[ \text{AttrSet} = \{ \text{attribute} \}_{1 \leq i \leq N} \]

where \( N \) is the number of attributes. Note that different objects may have a different \( N \).

2.2. Object types and attributes

In order to define an order, Ord, of attributes in the AttrSet, properties of an object are proposed to be taken into consideration. The properties of objects used to define objects are called object types.

Definition 2.2. A type of an object is defined by an attribute of the object. The types are labelled with object attributes. More formally, type, \( t \), is labelled 'attribute.tag', where 'attribute.tag' is an ASCII name allocated to a given property.

For example, all objects which are able to provide a printing service are of the printer type. In this case attribute.tag is \text{printer}, operator is =, attribute.value is any, and type is printer.

Initially, the RHODOS naming facility knows of the following basic types and instances of objects: files, software-based services, users, groups of users, ports, groups of ports, processors, printers, plotters, organizations, other types of objects. Corresponding to these basic types, there are the following \text{attribute.tag}s: file, server, user, user-group, etc (Goscinski and Haddock 1994).

Because objects having one property may have another property (feature) or provide another service, subtypes of a type can be derived. For example, the laser printer type or the dot-matrix type can be a subtype of the printer type.

Within the RHODOS environment an object is known by its attributes—its AttrSet is the complete set of its attributes (Definition 2.1). To be able to describe an object with attributes, the relationship between \text{attribute.tag}s and object types can be formally defined as follows:

\[ \text{Definition 2.3. Let } T \text{ be the set of object types which are known to the naming system, and } t = \{ \text{attribute.tag} \text{attribute.tag} \} \text{ is an element of attribute, for all attributes associated with } T \text{ be the ASCII representation of } T. \text{ Then for each object we can associate a set of its } \text{attribute.tag}s \text{ which will be a subset of } t. \text{ An object with a set of object types } T_0 \subseteq T \text{ will be labelled by an } \text{attribute.tag} \text{ set } t_0. \text{ An object type } T_1 \text{ is a subtype of an object type } T_0 \text{ if its } \text{attribute.tag} \text{ set is a superset of the } \text{attribute.tag} \text{ set } t_0 \text{ of } T_0. \]

For example, the following attributes of an object \{file = source code, language = C, purpose = searching\} show that the object is a C source file of a searching program. The attribute tag set \{file, language, purpose\} is the ASCII representation of its object type. This object type is the subset of both object types presented by the attribute tag sets \{file, language\} and \{file\}.

Note that this object type model does not imply the class inheritance hierarchy commonly used in object-oriented models. Attribute tags specify the types of properties possessed by objects. They do not define any inherited relationship between an object type and its subtypes.

The above shows that the basic object types can be further divided into subtypes, such as the text file and the binary file, to allow object types to be expressed precisely. Furthermore, the RHODOS naming facility allows new object types to be created by creating new attributes. This means that the set of types is not restricted, and a new object type can be created when needed.

2.3. Contexts—imposing an order on attributes

It has been proposed in Goscinski and Haddock (1994) and Goscinski and Indulska (1992) that in order both to define an order, Ord, of attributes in an AttrSet, and to bind a UName to an object, the concept of a naming context needs to be considered. According to the RHODOS naming model, a UName in RHODOS stands for an object, and each RHODOS object is referred to by its SName. For the naming facility it is enough to bind a UName to an SName. The binding of UNames onto SNames is conducted in a relevant naming context, which is the context defined by a particular UName.

Saltzer (1977) defined a naming context as a particular set of bindings of names to objects. A naming context in RHODOS is slightly different. A RHODOS naming context is a set of objects which have the same \text{attribute.tag}. The following are the formal definitions of the RHODOS naming context and its structure, which are the framework to construct the RHODOS naming facility.

\[ \text{Definition 2.4. Let } S = \{ \text{all SNames} \} \text{ and } S' \subseteq S. \text{ Let } C \text{ be the object set corresponding to } S'. \text{ The number of objects in } C, \text{ denoted by } |C|, \text{ is called the cardinality of } C. \text{ Let } A = \{ \text{all attributes } (\text{attribute.tag}, \text{operator}, \text{attribute.value}) \}. \text{ If there exists one attribute } a \in A, \text{ such that } a \text{ is called a context and the } \text{attribute.tag} \text{ is its label.} \]
Definition 2.5. Let $C_i$, $C_j$ and $C_k$ be distinct contexts. If $C_j \subset C_i$, then $C_i$ is called a parent context and $C_j$ is called a child context (or sub-context). If there exists no $C_h$ such that $C_j \subset C_h \subset C_i$, then $C_i$ is called the direct parent of $C_j$, and $C_j$ is called a direct child of $C_i$. If $C_j$ is the direct parent of both $C_j$ and $C_k$, and $C_j \cap C_k = \emptyset$, then $C_j$ and $C_k$ are siblings. Obviously, $|C_i| > |C_j|$.

As we said earlier, in order to have name evaluation carried out efficiently an order on attributes in the AttrSet must be imposed. This attribute order is derived from an order of those contexts which are labelled by the attributes of an AttrSet.

Definition 2.6. Let $C_i$ and $C_j$ be contexts. If $C_i$ is a parent of $C_j$, then a shrinking order of contexts $C_i > C_j$ is determined. Let $a_{i_0}$ be the label of $C_i$ and $a_{j_0}$ be the label of $C_j$. If $a_{i_0}$ and $a_{j_0}$ are distinct attributes in a UName with attribute-iag labelled $a_{i_0}$ and $a_{j_0}$ respectively, then these attributes can be ordered as $a_{i_0} > a_{j_0}$, where $a_{i_0}$ is called the predecessor of $a_{j_0}$, and $a_{j_0}$ is called the successor of $a_{i_0}$.

The following theorem describes the relationship between a context and an object type.

Lemma. A context only has one direct parent and the label of the direct parent is the attribute-iag preceding the attribute-iag which is the label of that context.

Proof. Let $C_i$ be a context labelled by $a_{i_0}$, $C_{i-1}$ be the direct parent of $C_i$ and $a_{i-1}$ be the label of $C_{i-1}$. Suppose there exists another direct parent of $C_i$, $C_{i-1}$ and $C_{i-1} \neq C_{i-1}$. Let $a_{i_{0_1}}$ be the label of $C_{i-1}^1$ and $a_{i-1}^1 \neq a_{i-1}$.

Because $C_i \subset C_{i-1}$ and $C_i \subset C_{i-1}^1$, then $C_i \subset C_{i-1} \cap C_{i-1}^1$.

Let $o$ be an object, $o \in C_i$. Then $o$ has two UNames:

$a_1 = (a_{i_1}, \ldots, a_{i-1}, a_i, \ldots, a_N)$, where $a_i$ is an attribute whose attribute-iag is $a_{i_1}$; and

$a_2 = (a_{i_1}, \ldots, a_{i-1}^1, a_i, \ldots, a_N)$, where $a_{i-1}^1$ is an attribute whose attribute-iag is $a_{i-1}$, and $a_i \neq a_2$, which contradicts definition 2.1.

Therefore, a context only has one direct parent and the label of the direct parent is the attribute-iag of $a_{i-1}$.

Theorem. A context contains all objects which form the object type defined by the labels of that context and its parents.

Proof.

(i) Let $o$ be any object which has a UName, $a = \{a_i\}_{i \leq N}$. $a_i > a_j$, $j > i$, and $a_i = \{a_{i_1}\}_{i \leq N}$, be the attribute-iag set of $a$. Then $o$ has the object type defined by the $0 = \{a_{i_1}\}_{i \leq S}$.

Let $C_i$ be the context labelled by $a_{i_1}$, the labels of $C_i$ and its parents be $t' = (a_{i_1}^1, a_{i_1}^2, \ldots, a_{i_1}^k, a_i)$. If $C_{i-1}$ is the direct parent of $C_i$, then $a_{i-1}^1$ is the label of $C_{i-1}$. Because $a_{i-1}^1$ has the attribute-iag $a_{i-1}$, and according to the lemma, $a_{i-1} = a_{i-1}^1$, therefore $o \in C_i$.

(ii) Let $C$ be a context, $a_i$ be the label of $C$. If $o$ is an object and $0 \in C$, then $o$ has the UName $a = \{a_i\}_{i \leq N}$, where $a_i > a_j$, $j > i$, and the attribute-iag set of $a$ is $a_i = (a_{i_1}, \ldots, a_{i-1}, a_i, \ldots, a_N)$, which defines an object type for $o$, $T_o$. If $C_1, C_2, \ldots, C_{i-1}$ are parents of $C$, then their labels $a_1 = (a_{i_1}, a_{i_1}^2, \ldots, a_{i_1}^k, a_i)$ defines an object type $T'$. According to the lemma, $a_{i_1} = a_{i_1}^k$, $1 \leq k \leq i$, then $a_{i_1}^k \leq a_i$ and $T'$ is a supertype of $T$.

According to the theorem, if a context contains all objects of basic object types, then it is in the top level of a context hierarchy. Consequently, this context has the biggest cardinality (definition 2.5). We call this context the basic context. The child context of a root context can be created if the basic object type has subtypes.

The proposed hierarchy of the Contexts, which follows a shrinking order of contexts, is shown in figure 2(a). The basic context is the biggest context in the hierarchy. The smallest context is an object context. The object context comprises instances which are of the same types. In other words, the object context contains identical objects in terms of the services which they can provide, or the resources which are managed by one server. An attributed name may be interpreted, according to the shrinking order of contexts, in a context which is a composition of sub-contexts, and each context/sub-context is labelled with the attributes of the attributed name.

Definition 2.7. A UName of an object is a pair (AttrSet, $\triangleright$), i.e., a sequence of attributes, where AttrSet is a set of attributes describing properties of the object and $\triangleright$ is an order of shrinking contexts.

The shrinking order of contexts—the order of attributes—show a composition of sub-contexts in which a UName is unambiguous. It allows attributed naming to be effectively implemented. However, users do not have to know or remember the order of attributes when they refer to objects, in other words, this order is transparent to users. Users are still able to refer to an object with a sequence of attributes in an arbitrary order. A possible arrangement of a context hierarchy is shown in figure 2(b).
3. Attributed name evaluation

The RHODOS naming facility accepts an attributed name provided by a user, evaluates it against the information stored in the naming database, and then returns the result of evaluation in the form of a SName set to the user. An effective and efficient name evaluation mechanism is a key part in the development of an attributed naming facility.

3.1. Name evaluation

The RHODOS naming evaluation is based on the attributed name provided by a user. The attributed name which is to be evaluated is called a candidate name, denoted CName. From the set theory point of view, the RHODOS naming evaluation is mapping of a CName to an SName, defined as follows:

Definition 3.1. Let $A$ be the attributes set and $S$ be the SName set. Then the name evaluation is a mapping $f : 2^A \rightarrow S$.

Needless to say, the mapping $f$ is partial† and many-to-one‡.

Note that in the above definition, the RHODOS name evaluation mechanism allows a user to enter a CName in which the order of the attributes may be arbitrary, and not all the attributes of an object need to be given. From the users point of view, it means that users specify attributes of the desired object known to them. The name server then matches these attributes to find the objects with these attributes.

A number of factors need to be taken into account in the development of a name evaluation mechanism. Firstly, the ‘quality’ of a CName, which discriminates the accuracy of the information represented by a CName has a substantial effect on the result of the evaluation process. For example, a CName might contain enough attributes to match a single UName, such that a single object can be identified. In the other case, a CName only has attributes to distinguish a number of UNames, therefore the result is a group of objects. It is also possible that there is no attribute in a CName matching any one in UNames. The CName in the first case is an unambiguous CName. The CName in the second case is ambiguous. To make it unambiguous, attributes must be added.

Secondly, different clients of a name server would prefer different services to be provided with. Some clients would tolerate an ambiguous name evaluation results, while some would not. The term clients applies to both programs and human users. For example, a file server supplies a CName to the name server for the purpose of opening a file and expects at most one SName to be returned. In this case, the client, a file server, cannot accept an ambiguous result, i.e., more than one object. In contrast, a human user, for example, wants to learn the printer names in the system. The user often supplies several pieces of information about the objects he/she wants, such as the speed of the printer, the location of the printer and the resolution of the printer. This information may result in a number of printer names being returned. However, the user is willing to tolerate this ambiguous result as long as it contains the desired printer.

Thirdly, user demands may have an influence on an evaluation result. For example, a user wants a printer with the following attributes: \{printer type = laser, speed = 20, resolution = 200, cost < 90\}. The name server evaluates this CName and finds a printer which matches the first four attributes, except the cost, which is 100. Then the user may either accept the result, because he/she desperately needs it, or not accept the result, because he/she cannot afford the cost. In the former case, the user thinks that the first three attributes are the basic attributes which the required printer has to have. The last attribute is an optional attribute which the required printer is better to have. In the latter case, the four attributes are basic attributes. If the name evaluation mechanism allows a user to enter a CName as well as the conditions associated with the CName, and checks these conditions in the evaluation process, the efficiency of the attributed name evaluation can be improved. From the example above, if the user can specify the attribute cost < 90 as ‘basic’ or ‘optional’ for different demands, then the name evaluation can directly return the desired result to the user.

Furthermore, the result of name evaluation relies on the accuracy of information stored in the naming database. The naming database is constrained by the conditions under which it is implemented (Bowman et al. 1993). For example, if a particular attribute is not entered in the database, the evaluation for this attribute will fail. As another example, a naming database that is implemented in a distributed environment may lose information because independent components fail, or information may become out-of-date because of communication delays between the components of the system. As we are focusing on the effectiveness and the efficiency of the name evaluation mechanism at this stage, we assume that all the information stored in the naming database is accurate. This means no incomplete and out-of-date information is in the database.

3.2. A graph representation of the RHODOS naming context

RHODOS name evaluation algorithms are designed using the concepts of the naming context. The naming context is implemented as a hierarchy which can be represented by graph for purposes of describing the naming context and the name evaluation model.

As we demonstrated earlier, the contexts both define an order of attributes and bind UNames to SNames. The attributes which make up UNames are associated with contexts (section 2.3). For the purpose of representing RHODOS naming contexts, a graph is interpreted in a particular way. To represent the basic context hierarchy, a rooted tree is chosen. This rooted tree then is extended to a directed graph, to represent the name evaluation models. These constructs are illustrated in figure 3 and figure 4, respectively, and described as follows.

† Since not all elements in $2^A$ need to be evaluable.
‡ Since an object can be identified by more than one attributed name.
Figure 3. An example of RHODOS context tree.

(i) A rooted tree representation:
- A vertex is labelled by an attribute type. For example, in figure 3, vertex 2 is known as attribute tag 'text'.
- An arc which links two vertices in different levels represents the order of two attributes. For example, the arc that goes from vertex 1 to vertex 2, shown by an arrow, represents 'file -> text'.
- A non-leaf vertex represents a naming context which contains those attributes that have the same attribute tag. For example, 'text = ASCII' and 'text = postscript' are in vertex 2.
- A leaf vertex represents a set of SNames. These are the SNames of object instances which have the same type. For example, vertex 6 represents the SNames of all 'ps' files.
- A path from the root to a leaf of the tree, called a UName path, represents a UName. Each component of the UName is the attribute represented by the vertex that makes up the path and ordered by the arcs on the path. For example, in figure 3, the path described by the vertex list (1, 3, 10) determines a UName \{file = source code; language = C; purpose = searching\}. The object—in this example is a C source file 'searching'—to which the UName refers is identified by the SName at the leaf vertex at which the path terminates. The name evaluation which maps CNames onto SNames can therefore be performed by finding paths which determine UNames.

Here we assume that the attributes in the graph only have a single-value. For example, if a C source file can perform both searching and sorting functions, then a new attribute 'purpose = Searching&Sorting' has to be used to identify the file. Dealing with multi-valued attributes will be the subject of our further research.

(ii) A graph representation:
To represent sibling relations, the rooted tree representation needs to be extended to a graph, as shown in figure 4:
- Each arc† linking two vertices at the same level represents a sibling relation of two contexts.
- The shortest path comprising the paths from the root vertex to a leaf vertex in the context graph represents a UName. Each component of the UName is the attribute represented by the vertex that makes up the path and ordered by the vertical arcs on the path.

† We use a single arc with two arrows to express symmetric arcs in a directed graph for convenience.

Figure 4. An example of a RHODOS context graph.

The naming context graph provides a means of constructing objects into a hierarchical structure based on the attributes of a UName of the objects. In the case that the hierarchy relationship does not explicitly exist in a collection of attribute tags, additional attribute tags can be used to form the hierarchy. For example, in figure 4, the 'lib' can be an attribute tag of both a C source code and an executable object file. To be able to construct the naming graph, 'lib' has to appear more than once in the hierarchy.

3.3. Service-oriented name evaluation model

In this section, we propose a name evaluation model using the context graph described in section 3.2. Because SName are bound in naming contexts, and structures of naming contexts are presented with context graph, UNames can be seen as paths in the context graph. Thus, the name evaluation model is naturally described based on the context graph. Therefore, evaluation of a CName is performed in a way that component attributes of the CName are compared to the context graph in order to find UName paths determined by the CName.

Name evaluation involves extracting attributes of a CName, identifying the contexts to which each extracted attribute belongs and matching these attributes against the attributes associated with the vertices (contexts) in the context graph. After parsing a CName, the name server has to know how to start matching the attribute, i.e., which context the evaluation should start with.

The matching can start from either: the context indicated in the CName, in the form of a specified attribute or defined by default; or the root context. In the case that the user knows the context to which the required object belongs when supplying the CName, he/she can specify that context in the CName in such a way that the evaluation can be performed only on the sub-context-graph indicated. There is no need to search the entire context graph. On the other hand, if the user does not know the context that the required object is in, which is common for an ordinary user, then the name server can start the search from the root context. In this case, the entire context graph will be searched, which give the user the maximum possibility to find the required object.

The matching is carried out in the top-down fashion. From the starting vertex to the lowest vertex, each attribute is compared to the attributes in the corresponding vertex, respectively. The comparison results in two possible results for an evaluated CName:
• **evaluable:** Either the entire or a portion of the CName determines UName paths, where 'a portion of a CName' refers to an attribute list comprising a proper subset of the CName. The number of paths found can be one or more than one.

• **failed:** No UName path determined by the CName can be found.

These results are specified in a general way. They only take into account the 'quality' of a CName and do not take into consideration any other facts such as the ambiguity of a name evaluation result. These requirements usually are expressed in the form of name services semantics which are implied by the name service operations.

3.3.1. Naming services It has been proposed in Goscinski and Indulska (1992) that the RHODOS naming facility should provide the following naming services:

• **conventional service**—mapping UNames onto SNames. This service is similar to the service provided by a Unix file system, that is, a UName is provided and the corresponding SName is returned. This service requires an unambiguous result.

• **selection service**—This is a process-oriented service, which evaluates a CName provided by a client process and returns a proper SName without interacting with the client process. The SName can be an object or a server which is able to provide the required service, or which manages the requested object. If the CName contains enough attributes to identify an object, the object’s SName is returned. If the CName only contains attributes which can identify a service (e.g. printing) rather than an object (e.g. a printer name), the SName of the server which is able to provide this particular service is returned. This service allows dynamic binding between object names and objects. This means that the client process does not have to know the exact object name. The name server can select an object based on the attributes provided by the client, even if the object may be created after the client.

• **enquiry service**—This is a user-oriented service which allows a user to identify objects using imprecise descriptions (CNames); this service also allows a user to look up information about an object, such as the UName of an object, or the type of an object. It can be implemented with an interactive user-oriented interface to allow the 'interactive' error correction during the evaluation process.

In summary, a user will normally request an enquiry service from a name server. A process, on the other hand, can use the selection service to find an object of interest, or the conventional service if the UName of the required object is known by the process.

From the name evaluation point of view, the enquiry service can allow ambiguous names to be returned from the name server; the selection service can accept an imprecise CName as long as the answer gained from the name server is unambiguous; the conventional service only accepts a precise CName and the unambiguous answer. Therefore, the conventional server can be seen as a subset of the selection service, and the selection service can be seen as a subset of the enquiry service. The result of the CName evaluation—**evaluable or failed**—which specifies an ambiguous result, covers the semantics of the enquiry service.

A name evaluation algorithm needs to traverse the entire graph/sub-graph from the starting context, find all the possible paths, check these paths with the semantics of the required service, then return the result. If the service cannot tolerate an ambiguous result, the algorithm must check the result with the semantics defined by the service, filtering out all paths which might result in ambiguity. In order to achieve efficiency in the name evaluation process, we propose a service-oriented name evaluation model which supports a set of independent algorithms for each naming service.

3.3.2. Name evaluation model and naming services In order to build independent algorithms for different naming services, we have to re-specify the basic name evaluation model taking into account the semantics of each service:

• For the conventional service, an evaluable CName means that the entire CName determines only one UName path. If the comparison to the context graph leads to more then one UName path or there is an attribute which cannot be matched, the name evaluation process terminates and returns a failure code.

• For the selection service: If the comparison of a CName to the context graph leads to more than one disjoint UName path, or there are some attributes which cannot be matched, the name evaluation process should terminate and return failed. Therefore, an evaluable CName means that the entire CName determines either one UName path or a path ending at a non-leaf vertex. The case of a path ending at a non-leaf vertex implies the fact that the CName identifies a group of objects which have the type defined by the labels of the vertices making up the path. If these objects are managed by the same server, then the SName of the server is returned. For example, if the type of object identified by the CName is a printer, then the name server can map the CName to the SName of a printing server.

There will be a set of rules which specify the conditions for the name facility to decide whether those paths found are acceptable, which SName can be returned upon those matching paths, or which system server can be referred to the caller. These rules are used for checking of the matching paths by the algorithm. Setting the rules, which depend on the overall system configuration, the structure of the operating system and the way of objects being managed, is beyond the scope of the name evaluation.

In a large distributed system, the name space for objects having the same type can be very large. In this case, a single name server is not able to accommodate all the naming information. There is a need to have a trading service which consists of a number of cooperating traders to manage this naming information (Goscinski and Ni 1994 and Ni and Goscinski 1994).
For the enquiry service, an evaluable CName means that either the entire CName determines one or more UName paths, or the entire CName does not determine any UName path, but a portion of the CName determines a UName path(s). In the case of a CName portion determining UName paths, the remaining attributes in the CName cannot be matched. No UName path is determined by the entire CName. The remaining attributes cannot be evaluated. Such a CName is called a partially-evaluated CName. For example, the CName (file = text, language = C, C code = lib) cannot determine a UName path (see Figure 4), but the portion {language = C, C code = lib} can determine the UName path (file = source code, language = C, C code = lib). A user will usually have supplied a partially-evaluated CName to the enquiry service for one of the following reasons (White 1984):

(i) to guess an object UName on the basis of information about the object that he/she naturally possesses, which may result in some attributes being incorrectly guessed;
(ii) to ensure that the naming system has all the information it needs to identify the intended object by supplying extra attributes to the name server; or
(iii) to ensure that the naming facility does not identify an object other than the one intended by supplying incorrect attributes together with right attributes.

In the first two cases the user would be happy to accept the UName found. In the third case, the user would think that the evaluation of this CName failed and would reject the result. By reporting the UNames found to the user, the enquiry service allows the user to decide the correct course of action according to the UNames returned.

Another instance for the partially-evaluated CName is that a user will have supplied a CName with conditions associated with the attributes in the CName, such as some of the attributes indicated as 'basic' attributes. Those 'basic' attributes must be evaluated, while evaluation of the other attributes could help a user.

To evaluate a CName for the enquiry service, the sub-graph whose root is the starting context needs to be searched thoroughly. Name evaluation does not terminate until all the paths are found or the entire sub-graph has been searched.

Moreover, the functionality of the enquiry service can be intensified by introducing the concept of 'possible matching'. The possible matching means that an attribute in a CName is not exactly equal to, but 'compatible to'\(^*\) an attribute in the context. For

\(^*\) Where 'compatible to' means that two attributes may be distinct however the meaning represented by one attribute is a logical subset of that by another, or the conditions specified by one are stronger than those by another. For example, if \(a\): speed = 250 and \(b\): speed > 100, then \(a\) is 'compatible to' \(b\), and if \(c\): printing type = postscript and \(d\): printing type = ASCII, then \(d\) is 'compatible to' \(c\). Obviously, if two attributes are identical, then these two attributes are 'compatible to' each other. Here, a basic issue is how to define the semantics of 'compatible', in other words, how to decide that an attribute is compatible to another in performing a matching. This issue will be addressed in our future research.

\[\text{Figure 5. The structure of linked contexts.}\]
The T-Tree retains the binary search nature of the AVL Tree and the storage advantage of the B-Tree. A node is very fast since the binary search is intrinsic to the tree structure. An AVL Tree has a binary search nature, which has good storage characteristics (the pointer to data ratio is low). The T-Tree is a balanced binary tree (AVL Tree) with many elements in an ordered set (no arithmetic calculations are needed). A B-Tree has good storage characteristics (the pointer to data ratio is small). The T-Tree retains the binary search nature of the AVL Tree and the storage advantage of the B-Tree. A node of the T-Tree called a T-Node, which is a combination of an AVL Tree node and a B-Tree node, is shown in figure 6.

![Figure 6. The T-Tree structure.](image)

4.2. Index and its data structure

The attribute.tag of an attribute carries important information for name evaluation. It identifies the type of an object. It also decides the context to which the object belongs. Moreover, it relates to the shrinking order of contexts (definition 2.6) on which the context hierarchy is constructed. The attribute.tags are the most used data in name evaluation. An attribute.tag is passed in the form of an ASCII string by a user. Such an ASCII string format is suitable for users, but is not suitable for representing the data which are frequently used by a system server. It is necessary to have an internal representation for attribute.tags. This internal format should indicate the type of the object, the context to which the object belongs and the position of the context in the shrinking order of contexts. It is also necessary to have a method which maps attribute.tags to their internal format.

The internal format of an attribute.tag is a duplet (attr.num, context.level), called a context identifier. The attr.num is an integer assigned to an attribute.tag by the name server, in order to guarantee that it uniquely identifies the ASCII string used for the attribute.tag. The context.level indicates the position of a context in the context graph. It is determined by the order of an attribute in a UName. With the context identifier, a context can be easily located in a context graph.

In order to achieve high efficiency of name evaluation, a data structure called an index which maps attribute.tags onto context-ids is used. The index is a special data structure called a 'T-Tree', evolved from AVL Trees and B-Trees (Lahman and Carey 1986). The T-Tree is a balanced binary tree (AVL Tree) with many elements in a node. An AVL Tree has a binary search nature, which is very fast since the binary search is intrinsic to the tree structure (no arithmetic calculations are needed). A B-Tree has good storage characteristics (the pointer to data ratio is small). The T-Tree retains the binary search nature of the AVL Tree and the storage advantage of the B-Tree. A node of the T-Tree called a T-Node, which is a combination of an AVL Tree node and a B-Tree node, is shown in figure 6.

Like an AVL Tree, rebalance of a T-Tree may occur when inserting or deleting a T-Node. The rebalancing is done using rotations similar to those of an AVL Tree (Wirth 1976), but it is done much less often than in an AVL Tree. Because a T-Node contains many elements, data movement which is required for insertion and deletion is usually needed only within a single node. Inserting a new element does not always result in a new node being inserted, unless the leaf nodes are full. Deleting an element usually results in an element being removed from a leaf node. Deleting a leaf node happens only if the leaf node is empty. The rebalancing cost results from updating a T-Tree can be reduced significantly in the way that the T-Tree is formed (Lahman and Carey 1986).

The data elements in T-Nodes are the attribute.tag structure which is a table to map an attribute.tag to a context identifier.

4.3. Name evaluation implementation

The evaluation of a CName is carried out in three steps.

(i) Mapping attribute types onto context-ids. This is to extract each component attribute from a CName, and convert each attribute from its ASCII string format to its internal structure. The internal structure of an attribute is a triplet: (context.id, operator, attribute.value), which is denoted by (c, o, v). The mapping of an attribute.tag onto a context.id is accomplished by searching the index with the ASCII string of the attribute.tag as the key (see section 4.2).

(ii) Sorting attributes. The context graph is organized in the shrinking order of contexts, which defines the order of attributes to be evaluated; the attributes in a CName need to be sorted into the same order as that in the context graph before calling a name evaluation routine. The order of attributes is defined by the context.level part of a context.id in (c, o, v). Therefore, all (c, o, v)s are sorted in a monotonically increasing order of context.levels.

(iii) Searching a context graph. The search of a context graph is used to find a matched UName. For the efficiency, a search is carried out in various ways which depend on the semantics of the three naming services as stated in section 3.3.2. We have developed three independent algorithms with one for each naming service, namely, the conventional service, the selection service and the enquiry service. These algorithms take a sorted (c, o, v) list as the parameter and return the evaluation result—evaluable or failed. If a CName is evaluable, the UName(s) which match the CName will also be returned.

The basic operation in the searching of the context graph is to compare the attributes in a CName against those attributes stored in the context graph. Those attributes are contiguous in terms of the attribute order. As both attributes in a CName and in the context graph are ordered, a sequential search which compares the elements in an ordered set against those in another ordered set is sufficient to conduct the searching of the context graph. However, there can be 'gap's in the CName. A 'gap' refers to a point at which one or more attributes are
missing, such that the attributes between that point are not contiguous. The sequential search can be carried out up to the point where a 'gap' in the CName is found. An exhaustive search needs to be committed to 'fill' the 'gap' (See algorithm 1 in the appendix). This algorithm starts at the context in which the last matched attribute was found. It exhaustively searches the sub-context-graph below that context until the next matching context is found, or all the contexts at the level indicated by the context_level in the attribute next to the 'gap' in the sub-graph have been searched. If the next matching context is found, the sequential search is continually carried out until the next 'gap' is found or all attributes in the CName have been checked. In the case of the name service being distributed across the network, the algorithm can be performed in parallel. The searching name server can contact all other sibling name servers to find the next matching context (Goscinski 1991).

The fundamental searching technique used here is the sequential search which is complemented by exhaustive searching. This technique is used in all name evaluation algorithms presented in the following sections.

4.4. Starting a name evaluation

Before name evaluation is conducted, a starting context needs to be located. The starting context can be the root context, the current context or the context indicated by the first attribute in the CName.

The algorithm for locating a starting context (see algorithm 2 in the appendix) takes the first component in an attribute-list as the starting context. If the starting context is neither the root context nor the current context, the algorithm then searches the context graph from the root vertex in order to find the corresponding context entry. If such a context entry exists, the contexts in the entry are compared in order to find the matching context. It returns a context-list, each element of which is a (context_id, context) pair. The context-list contains all contexts matching the parameter. The reason for the algorithm to return a context list rather than a context is that it can happen that the same attribute can appear in different UNames.

4.5. The evaluation algorithm for the conventional service

The algorithm performing the name evaluation for the conventional service (see algorithm 3 in the appendix) accepts an attribute-list (CName) and finds all matching paths in the context graph by comparing the component attributes of the CName against the graph. After the comparison has finished, the matching paths are checked in order to allow an unambiguous result to be returned to the caller of the algorithm. The checking is to be carried out based on the name evaluation model for the selection service (section 3.3.2). To be able to store the multiple paths which can be matched in the context graph, in addition to the Clist, a data structure matching-list (Mlist) is used by the algorithm. The Mlist links a number of Clists, each of which stores a matching path (figure 7).

When the algorithm starts, it calls the starting context function. In the case that the starting context is either the root context or the current context, the function returns only one context. In the case that the starting context is specified in the CName, the function can return more than one contexts. Starting from these contexts, different matching paths may be found.

After finishing the matching procedure on each Clist in the Mlist, the algorithm returns the result upon the conditions discussed earlier in this section.

4.6. The evaluation algorithm for the selection service

The algorithm performing name evaluation for the selection service (see algorithm 4 in the appendix) accepts an ordered attribute-list (CName) and finds all matching paths in the context graph by comparing the component attributes of the CName against the graph. After the comparison has finished, the matching paths are checked in order to allow an unambiguous result to be returned to the caller of the algorithm. The checking is to be carried out based on the name evaluation model for the selection service (section 3.3.2). To be able to store the multiple paths which can be matched in the context graph, in addition to the Clist, a data structure matching-list (Mlist) is used by the algorithm. The Mlist links a number of Clists, each of which stores a matching path (figure 7).

When the algorithm starts, it calls the starting context function. In the case that the starting context is either the root context or the current context, the function returns only one context. In the case that the starting context is specified in the CName, the function can return more than one contexts. Starting from these contexts, different matching paths may be found. The algorithm takes all the starting contexts, initializes Clists with these contexts and starts the matching procedures at these contexts one by one.

After finishing the matching procedure on each Clist in the Mlist, the algorithm returns the result upon the conditions discussed earlier in this section.

4.7. The evaluation algorithm for the enquiry service

An algorithm performing name evaluation for the enquiry service is required to be able to find UName paths determined by either the entire CName or a portion of the CName in a context graph. Furthermore, because the enquiry service is a user-oriented service and is to be implemented with an interactive and flexible user-server interface, it is desirable that an enquiry operation can accept a logical expression of attributes. For example, a user may
issue a name operation to find printers with the following attributes:

\[
\text{printer.type = postscript; speed = 10; resolution = 200; cost = 20; location = 1st floor},
\]

These attributes can be expressed in the form of a logical expression as follows:

\[
\text{((printer = ps \land speed = 10 \land resolution = 200) \land (cost = 20 \lor location = 1st floor))}.
\]

These attributes may be evaluated in an interactive manner. The algorithm first evaluates the attributes \text{printer}, \text{speed}, \text{resolution} and \text{cost}. If a printer matching these attributes has been found, then the algorithm returns the result and asks the user whether he/she wants the evaluation to be continued.

The algorithm designed for the enquiry service (see algorithm 6 in the appendix) takes a logical expression of attributes as the parameter. It parses the expression into CNames and carries out the evaluation procedure on each of them. This algorithm is non-interactive, but it can be implemented in an interactive manner.

In our current prototype implementation, checking whether two attributes are compatible is done by checking the operator parts of a context and of an attribute in order to see whether they are in valid range, e.g. '≠' is compatible to '≥'.

For each CName, the algorithm matches the attributes in the CName against those in the context graph. The matching is to find context_ids and the contexts which are compatible to the attributes in the CName. The matching is carried out in the following way:

- if the current context_id is equal to the current attribute.sag, then it goes into the context entry to find a compatible context;
- if neither the current context_id nor the contexts within that context entry are compatible to the current attribute, then it goes to look at the siblings of the current context entry. Because sibling contexts of a context contain those objects which have identical supertype but different subtypes (theorem), the looking up the sibling context can result in the objects with a compatible object type being found.

The fundamental data structure used by this algorithm is the Mlist, which is similar to the one in the selection service.

4.8. Related work

In this section, we compare our work with related work.

Neufeld has proposed a descriptive name resolution method for an X.500 directory service (Neufeld 1992), (Neufeld et al 1992). A descriptive name is an unordered set of attributes which unambiguously denote an object. Such a descriptive name is independent of the logical name hierarchy defined in the X.500 directory. To resolve descriptive names efficiently, Neufeld proposed a concept of registered names. A registered name is a set of attributes uniquely defining an object. It is guaranteed that no two distinct objects have the same registered names. The resolution of a candidate descriptive name results in one of three cases: the name was found, not found or ambiguous. In fact, the cases of not found and ambiguous imply the failure of the resolution as far as the name resolution algorithm is concerned. Only if the candidate name matches one object, can the name resolution algorithm return the object found. This is semantically equivalent to our conventional name service. However, the selection and the enquiry services cannot be implemented with this approach.

Univers is another attempt made on descriptive names (Bowman 1990), (Bowman et al 1993). Univers supports attributed names and allows these names to be presented with any combination of attributes. It has been proposed that Univers has been designed to not only provide a single name service, such as name-address mapping, but also provide a mechanism upon which a variety of naming services can be implemented. It is based on a model called preference hierarchy, with which clients may specify the semantics about the name resolution functions and the preferred methods for accommodating the inaccurate information which may appear in the name to be resolved. This approach is similar to our name evaluation model which is based on the semantics of different name services. In theory, the preference hierarchy model is more general and formal than our name evaluation model in the sense of describing naming services. However, the attribute values defined in Univers have been restricted to only a string-value, which limits the scope of the application of Univers.

5. Implementation of a distributed attributed naming service

Based on the name evaluation algorithms and the data structures which store information about attributes, a distributed attributed naming service can be implemented. These data structures will be held in a database—the naming database. How to implement a naming database to support name evaluation efficiently is one of the crucial issues of the development of a distributed naming service. Other issues include how to distributed naming service across the network, and how the name servers cooperate with each other to perform the name evaluation.

In the rest of this section, we briefly discuss the issues of a naming database and naming service distribution. The issue of name server cooperation has been addressed in Goscinski and Ni (1994) and Ni and Goscinski (1994).

A naming database is a data repository which contains information for the purpose of name evaluation. It is different from commercial databases. In addition to the general features supported by general-purpose databases, such as data integrity and reliability, a naming database is required to have the following features:

- the response time should be as short as possible;
- the data in a naming database must persist across system restarts in order to ensure the availability of the name server which uses that data; and
- a typical naming database is relatively small—the size of such a naming database for a large name space is less than 10Megabyte (Birrell et al 1987).
Thus, we can reasonably assume that the entire database fits in the main memory. As a result, a main-memory specialized naming database approach has been considered and developed. A main-memory database is a new class of database which is different from a conventional disk-resident database (García-Molina 1992). This database can be maintained as strongly typed data structures. Enquiries take only the time necessary to access main memory. It is clear that such a database can achieve better performance than traditional disk-resident databases do.

The architecture of the proposed RHODOS naming database is shown in figure 8. It has two main parts: an Index part and a Context part. The Index is a linked ‘T-Tree’ index structure and the Context is a linked naming context graph structure. A DBMS which directly accesses the data in the database is responsible for maintaining the database.

The RHODOS naming database is to be implemented in a distributed environment. This requires the distribution of the naming database to be considered. At this stage, we mainly pay our attention to the efficiency of the naming database. The detailed study of the distribution of the naming database is beyond the scope of this paper. However, we present here several examples of distributed implementations of the naming database.

The first example is a fully centralized naming database (figure 9). The entire database is stored on a single computer, but the logical separation of the sub-databases is retained within the central database. However, such a fully centralized approach could cause two problems: (1) The reliance on a single computer to store all the data could decrease the availability of the naming database and the naming service which uses the database; (2) The naming database’s relying on a single computer to handle all database operations could cause a bottleneck problem, thereby degrading system performance.

The second example is a straightforward distribution based on the logical structure of the naming database. This arrangement is to divide the naming database into several sub-databases with each of them consisting of a user naming database or a few user naming databases, as shown in figure 10. This distribution physically separates the logical sub-databases, but retains the Index and the Context in each sub-database undisturbed.

The third example is based on naming contexts. The distribution will allow the naming database for a few contexts, a context or sub-contexts of them to be distributed.

For example, the file context can be placed close to a file server, or even integrated into the file server, as shown in figure 11.

6. Conclusion

In this paper, we have presented a new and original name evaluation method of attributed names with which a user-friendly name server for a distributed operating system can be achieved. This method is an extension of the two different name evaluation approaches (the relational evaluation and the sequential evaluation). It allows an attributed name to be presented as a set of attributes and to be evaluated as efficiently as if it were provided in a hierarchical format. Attributed names are therefore much more user-friendly to use than partitioned names because they do not require knowledge of the strictly formatted name hierarchy. Moreover, attributed names can be evaluated even if they are not fully specified. This enables users to infer an object name from their own knowledge about the object, without knowing its UName.

In order to achieve high efficiency in attributed name evaluation we use the concept of naming contexts which bind attributed names onto system names. Naming contexts form a hierarchical structure. This structure allows attributed names to be evaluated without sacrificing efficiency.

We have developed a name evaluation model which supports different naming services, namely, the
conventional service, the selection service and the enquiry service. These naming services have different semantics and functionality and place disparate requirements on the results of name evaluation. In order to approve the effectiveness and efficiency of name evaluation and to simplify the implementation, we have developed a set of independent name evaluation algorithms one for each particular naming service. By using different algorithms for different naming services, the complexity of the name evaluation algorithms can be reduced, because only the requirements from a particular naming service need to be dealt with.

Acknowledgment

The authors would like to thank Dr Anthony Klemm and Damien De Paoli for their help during the writing of the paper. The authors would also like to thank Dr Morris Sloman and anonymous reviewers for their constructive comments.

Appendix


```
search(content_id, attr_num, content_level)
    for (ptr_id = first; ptr_id != NULL; ptr_id = ptr_id->sibling)
        if (content_level = ptr_id->content_level && attr_num = ptr_id->attr_num)
            return (ptr_id);
        else if (content_level < ptr_id->content_level)
            content_id = search(ptr_id, attr_num, content_level);
        if (content_id != NULL)
            return (content_id);
    return (NULL);
```

Algorithm A2. The algorithm of locating a starting context.

```
conventional (CName)
    get the root NULL;
    get the start point;
    if (there are more than one start points)
        return (fail);
    link the start point into the List;
    for (each attribute in the CName)
        if (the attribute is contiguous to the last attribute)
            move to the child context_id;
        if (the context_id does not match the attribute tag)
            return (fail);
    else /* the context_id matches the attribute tag */
        check contexts linked to the context_id with the attr_value;
        if (no context matching the attr_value)
            return (fail);
        else /* existing context matching the attr_value */
            link the context to the List;
    else /* the attribute is not contiguous to the last attribute */
        call the search-gap routine to find the next matching context;
        if (search-gap does not find any matching context)
            return (fail);
    else /* the context has been found */
        if (there are more than one matching contexts)
            return (fail);
        else only one matching context
            move to the matching context;
        check contexts linked to the context_id with the attr_value;
        if (no context matching the attr_value)
            return (fail);
        else /* existing context matching the attr_value */
            link the context to the CList;
    if (the last context in the CList is a leaf)
        return (CList);
    else
        return (fail);
```

Algorithm A3. The algorithm for the conventional service.
search-gap (attribute, content)

*search the sub-context from the last matching context_id;*
if (no context_id matching the attribute)
  return (null);
else
  for (each matching context_id)
    check contents linked to the context_id;
    if (find a context matching the attribute value)
      add the context and context_id to the context-list;
    if (the context list is not null)
      return (context-list);
  return (null);

Algorithm A4. The algorithm of search a gap.

Algorithm A5. The algorithm for the selection service.

Algorithm A6. The algorithm for the enquiry service.

References

Birell A, Jones M and Webber E 1987 A simple and efficient implementation of small databases Operating Syst. Rev. (USA) 21(5) 149–54
Bowman M, Peterson L and Yettas A 1990 Universal attribute-based name server Software-Practice and Experience 20(4) 403–24
Goscinski A and Haddock A 1991 The development and experiences with the RHODOS naming facility Technical Report CS37/91 Department of Computer Science, University College, ADFA, UNSW, November
Goscinski A and Haddock A 1991 A naming and trading facility for a distributed system Australian Computer J. 26(2) 50–65
Evaluation of attributed names


Lehman T and Carey M 1986 A study of index structures for main memory database management systems *Proc. Twelfth Int. Conf. on Very Large Data Bases (Kyoto)* (Palo Alto, CA: Morgan Kaufmann) pp 294–303


Mogul J 1986 Representing information about files *Report No. STAN-CS-86-1103* Department of Computer Science, Stanford University

Neufeld G 1992 Descriptive name resolution *Computer Networks ISDN Syst.* 23 211–27


Ni Y and Goscinski A 1994 Trader cooperation to enable object sharing among users of homogeneous distributed systems *Computer Commun.* 17(3) 218–29


Peterson L 1988 The profile naming service *ACM Trans. Computer Syst.* 6(4) 341–64


Schwartz M A 1990 Scalable, non-hierarchical resource discovery mechanism based on probabilistic protocols *Preprint CU-CS-474-90*, Department of Computer Science, University of Colorado

Vance C and Goscinski A 1989 The logical design of a naming facility for RODOS *Technical Report CS89/15*, Department of Computer Science, University College, ADFA, UNSW


Wirth N 1976 *Algorithms + Data Structures = Programming* (New York: Prentice-Hall)

Yoo A, Ananda A and Koh E 1993 A taxonomv of issues in name systems design and implementation *Operating Syst. Rev.* 27(3) 4–8