Published in Joint International Conference on Open Distributed Processing and Distributed Platforms. May 1997. Chapman & Hall.

Federated Naming in an ODP Environment

P. Kahkipuro, L. Kutvonen, L. Marttinen
Department of Computer Science, University of Helsinki
P.O. Box 26, FIN-00014 University of Helsinki, FINLAND
phone: +358 9 708 51, fax: +358 9 708 44441
{Pekka.Kahkipuro, Lea.Kutvonen, Liisa.Marttinen}@cs.Helsinki.FI

Abstract

Open distributed computing heavily relies on a shared infrastructure model. Within the infrastructure, a naming service is the most fundamental component because other components, such as trading and type repository services, rely on it. Currently a naming framework is under standardization within ODP-RM (Open Distributed Processing Reference Model of ISO and ITU). The key requirement–and problem–for a naming framework is an effective model for name federation between sovereign computing systems. This paper proposes a federative model for naming in an ODP environment. The model separates user-level naming, i.e. local needs, from the infrastructure level name management that solves problems arising from global integration. Although the paper was induced by questions related to standardization, the more important goal of this paper is to show two important aspects of distributed systems. First, we illustrate how system autonomy manifests itself; second, we show how to deal with federation.

Keywords

Open Distributed Processing (ODP), Autonomous systems, Distributed platforms, Naming

1 INTRODUCTION

The current architecture for open distributed computing represented by the Open Distributed Processing Reference Model (ODP-RM) (ISO/IEC, 1995a, 1995b) supports the integration of services across organizations. This architecture relies on a shared infrastructure model, but allows the participating computing systems to stay sovereign. Each participant implements and administers the basic infrastructure services autonomously. Every infrastructure server controls and serves only the resources and the active objects of the local computing system–thus controlling a domain. In addition to serving its own domain, the server can request services from other federating servers in order to support a global service across domains.

Naming is one of the most fundamental functions in the infrastructure, and other infrastructure functions rely on naming. For example, the trading service requires the interpretation of type names; the type repository service requires the resolutions of description language names; and the federated binding function requires the translation of interface location names to physical addresses.

A name is a relationship between a term and an entity. The format of the terms is restricted by naming conventions that may be independently designed for different kinds of entities or for different kinds of technical solutions. The named entities may include users, hardware components, protocols, applications–even names. For each class of named entities, there is an authority that is responsible for interpreting the names, such as type managers and traders. The term-entity relationships can have additional properties meaningful to the authority that interprets the relationship. The structure of terms is an example of such a property.

Names that have a common term structure can be gathered into collections that are called naming contexts. A naming context is the scope in which a name needs to be resolved. Logically enough, we may also give names for naming contexts. For addressing entities outside the current naming context, compound names can be used. A compound name is a structured composition of simple names that can be resolved to denote a single entity. For example, a compound name in a UNIX file system is a sequence of simple names, where the last name denotes an arbitrary entity, and all other names denote naming contexts. Thus, compound names allow us to give structure for the names in a system.

There are two traditional solutions for handling names in distributed systems, global naming and context-relative naming. In both solutions, names are compound. In global naming, terms for named entities are arranged into a single tree hierarchy of contexts. To refer to an entity, terms are concatenated along the path from the root to the entity. Context-relative naming supports arbitrary context networks instead of a single hierarchy. The global naming scheme is thus a special case of the context-relative naming scheme.

The context-relative naming scheme has been chosen as a basis for the ODP Naming Framework (ISO/IEC, 1996a). Most modern naming solutions use similar approaches, including the Object Management Group's Name Service (1994) and the ANSA Naming Model (van der Linden, 1993). However, the ODP and ANSA approaches use term-entity relationships as names, while the OMG uses terms as names. The difference affects name communication. If names are communicated by sending and receiving terms, additional mechanisms must be devised to ensure that the receiver can interpret the terms correctly once they have been received. On the other hand, if communication uses term-entity pairs (i.e. relationships between terms and entities), the receiver can immediately use the names because they carry their meaning with them. A name is communicated correctly if it denotes the same entity before and after communication. The actual term may have changed. The use of name-entity pairs simplifies applications because the burden of name interpretation is partly delegated to the infrastructure. Our approach is based on interpreting names as term-entity pairs.

The naming framework design should find a solution for two problems. First, the context-relative naming is often used in a way that forces users (end users or application programmers) to know from which context the name resolution should start and identify that context using some form of global naming scheme. Second, in distributed environments we have multiple alternatives for choosing how federation is modeled and designed.

In this paper, we propose a federative model for naming in open distributed environments. Traditionally, applications have been dealing directly with federated names, such as URLs in the Internet environment, and this has complicated application development. In our approach, the problems caused by federation are solved in a generic manner and delegated to the infrastructure. Hence, our model supports application development at a more abstract level.

The model separates user-level naming from the infrastructure-level name management, because these two are separate functions. The user-level naming service fulfills the local needs of a user or a sovereign application object. The infrastructure level naming services solve the problems faced when integrating existing name spaces of independent computing systems. The information needed for federation is embedded in the name objects themselves. This solution makes federation protocols more flexible compared to the case where federation is realized at the name server level.

Section 2 of this paper discusses the separation of user-level and infrastructure names, and sections 3–5 concentrate on naming related activities. The proposed naming solution has its foundations in the ODP Naming Framework, and the presentation follows the ODP viewpoints because we believe that framework standards should be structured along the viewpoints. This paper illustrates some generic features of ODP viewpoint specifications. In the proposed model, most of the consequences of system integration are discussed in the information viewpoint. It is typical for the infrastructure functions and frameworks that distribution is visible, not only in the computational viewpoint, but also in the enterprise and information viewpoints. Section 6 discusses the feasibility of the proposed approach and demonstrates that it is comparable to existing naming solutions. Section 7 concludes the discussion.

2 USER-LEVEL AND INFRASTRUCTURE NAMES

To understand the scope of naming, we distinguish between user-level names and infrastructure names. User-level names are intended for end users and applications. They are always interpreted relative to a private naming context of the object. User-level names are typically simple and user-friendly. Infrastructure names are intended for system-wide use. A special user-level name translation is needed to map user-level names into infrastructure names. All names in a private naming context are mapped to a set of naming contexts in the infrastructure. These contexts together constitute the local naming context offered to that particular user. Hence, local naming context is a view that allows the user to exploit infrastructure names. The solution transfers the burden of naming context management from users to the infrastructure. Similar techniques have been applied in related areas.

The example in Figure 1 illustrates a configuration where user u_1 has a private naming context with user-level names $\mathbf{n_1}$ and $\mathbf{c_2}$. The user-level name $\mathbf{n_1}$ corresponds to the infrastructure name n_1 that denotes some entity e_1. The name $\mathbf{c_2}$ corresponds to the infrastructure name c_2 that denotes some naming context C_2. To reach from the context C_1 the entity e_3, a compound name (c_2, c_3, n_3) can be used. Users do not need names for their private or local contexts. For example, u_3 does not know that his local context has an infrastructure name (c_2, c_3) in the context C_1. In spite of this, u_3 can access the name n_3 in his local context.

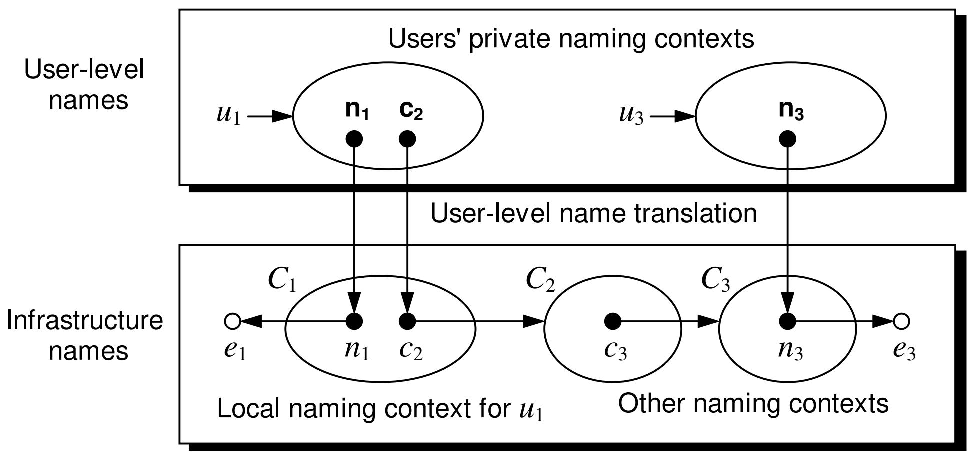

Figure 1 Infrastructure mapping transforms user-level names into infrastructure names.

It is important to separate user-level names from infrastructure names, because otherwise applications must use unnecessarily complex names. An indication of the complexity of context-relative naming without the proposed distinction can be found in Siegel's example (1996). In this example, a simple application was designed and implemented with eight different CORBA environments. Naming service was identified as a key component in the design phase, but the standard context-relative Naming Service was not used in the implementation. A simple Pseudo Name Service was proposed as a replacement, and the main difference to the standard is that it is not context-relative. The fact that the OMG Name Service forces applications to be fully aware of naming contexts has lead to the use of a non-standard replacement.

User-level names are important for application developers, but a global naming solution always operates at the infrastructure level and only provides a mapping to the user level. Whenever new applications are installed, user-level names are bound to the corresponding infrastructure names. This may sometimes require additional configuration work, but the result is application portability across different naming environments. The rest of this paper concentrates on infrastructure names because they provide the basis for successful naming.

3 SCOPE OF NAMING

This section focuses on the enterprise viewpoint of the proposed naming solution. It presents the different roles (i.e. enterprise objects), activities, and policies that are needed for understanding the purpose and the scope of naming.

The primary purpose of naming is to provide means for identifying entities in information systems. This is necessary for two reasons. First, to interact with an entity, we must distinguish it from all other entities. Second, we need a way to denote an entity even when we are not directly interacting with it. Especially, naming is needed for binding to objects, identifying system resources, selecting objects interactively, etc.

In particular, naming excludes two functions. First, naming is not used for security purposes. For example, the knowledge of a particular name does not imply the right to communicate with the entity it denotes. Second, naming is not used for typing purposes. Name alone should not be the basis for determining the properties of the entity that it denotes. These two restrictions allow us to simplify the naming solution and to make it easier to use.

We define a naming context to be a 3-tuple (T, C, E) where T is a set of possible terms, E is a set of namable entities, and C is a relation between the other two. Hence, the naming context C can be any subset of the Cartesian product $T \times E$. A name is formally defined to be a term-entity pair (t, e) belonging to C. If we have $(t, e) \in C$, we say that the context C contains the term t referring to the entity e. The choice of the set of terms T is implementation dependent, but it must be a subset of a single name space determining those terms that can be used as names. The set E determines which entities can be named. In principle, E is allowed to contain any entities, including names and naming contexts. In practice, implementations may set limits to the extent of E.

Naming domain is the basic environment for all activities that are related to naming. Naming domain is controlled by a single naming authority, and it uses a single naming convention that determines the syntax of names. Formally, we represent naming domain with a 3-tuple $(T, \mathrm{Dom}_D(C), E)$ where $\mathrm{Dom}_D(C)$ denotes those entities in the context C that are controlled by the naming authority of the domain D. Naming authority is an object responsible for controlling the domain's activities. It may impose different kinds of policies for the domain. For example, homonyms may be forbidden or the number of synonyms may be restricted. Note that a single naming context may be divided into multiple parts, each of which is controlled by a different naming authority and hence belongs to a different naming domain.

We define naming federation to be a set of federated naming domains. The purpose of federation is to collectively support naming contexts. Each domain preserves its autonomy of control, but there is a mutual accordance to allow some activities happen over domain borders. Federation agreement determines these joint activities in more detail. Typically, name bindings effectuated in one domain may be visible in other domains through name resolution.

The following roles can be identified for the users of the naming framework:

- **Name creator** associates names to entities.
- **Name user** uses the name resolution mechanism to know what the names denote.
- **Name sender** transmits names to other parties.
- **Name receiver** obtains names from name senders.
- **Naming authority** controls the naming domain.

The above roles are relative to a particular naming domain. It is practical that name creators are also name senders, and name users are in most cases also name receivers. The identified roles participate in the following activities:

- **Name resolution** attempts to find, for a given name, associations to those entities that the name denotes.
- **Name comparison** attempts to resolve two names up to a point where it is possible to determine if they denote the same entity or not. Even if the resolution succeeds, it may still be impossible to establish whether they refer to the same entity. For example, it is possible that an entity does not support operations for revealing its identity, or it may not allow some users to invoke these operations. Hence, name comparison should have four different outcomes: equal, not equal, undetermined, and resolution failed.
- **Name communication** involves transmitting a name from a sender to a receiver through the infrastructure. This may involve name transformations, especially if the sender and the receiver are in different domains.

At the enterprise level, the proposed naming solution has important benefits for the users of the naming framework:

- Name receivers, users, and senders do not need to know whether their naming domain belongs to a federation or not, and whether the names are local or federated. They simply receive, use, and send names as if they were always local.
- Name creators that create bindings for local entities can also ignore naming federation. Name creators need to be aware of federation only if they initiate name bindings that involve the activity of a different naming domain.
- Naming authorities need to be aware of naming federations at all times.

4 BASIC NAMING CONCEPTS

This section describes the information viewpoint of the proposed solution. We first define the basic semantics of name processing without federation and, then, we give the additional definitions needed for naming federations. Finally, the general information contents of names will be discussed.

4.1 Semantics of name processing

The purpose of name processing is to manage naming contexts. Naming contexts can be changed by two atomic actions, name binding and name unbinding. The former adds a new term-entity pair to a context, and the latter removes an existing one. For a given naming context C, term t, and entity e, we define the actions as follows:

$\text{Bind}(C, t, e) : C \;\alpha\;\; C \cup \{(t, e)\}$,
$\text{Unbind}(C, t, e) : C \;\alpha\;\; C - \{(t, e)\}$.

Other actions can be defined using the above two actions. For example, a renaming action is an atomic sequence of Unbind(C, n_1, e) and Bind(C, n_2, e) for some entity e. A name rebinding action is an atomic sequence of Unbind(C, t, e_1) and Bind(C, t, e_2) for some term t. In principle, every naming context is initially empty. In practice, however, naming contexts are seldom created at system start-up time and they usually contain some pre-installed names that allow the system to start its operation smoothly. For example, the name of the trading service is usually available so it can used it for reaching other services.

We can easily extend the set of entities that can be reached from a given naming context by combining the naming context's name with the name of some entity in that context. Formally, we define a compound name in context C_1 to be a sequence of terms $(t_1, \ldots, t_k)$ where $(t_i, C_{i+1}) \in C_i$ for all $i = 1, \ldots, k{-}1$. Thus, we require that all name components except the last one are bound to a naming context. Implementations often use character strings to represent both simple and compound names. Syntactical conventions indicate how the component names can be found from the string. For example, slashes are used to separate individual terms in the UNIX file system path names.

Name resolution is a process that attempts to find, for a given term t and a given naming context C, all pairs (t, e) that belong to C. We use the symbol Res(C, t) to denote the set of found term-entity pairs. Name resolution does not necessarily find all suitable pairs because it may be affected by other concerns, such as those related to security. To support compound names, we specify recursively the results of the name resolution process:

$\text{Res}(C_1, t) \subset \{(t, e) \mid (t, e) \in C_1\}$,
$\text{Res}(C_1, t_1, \ldots, t_k) \subset \{(t_k, e) \mid \exists\, C_k : (t_k, e) \in C_k \text{ and } (t_{k-1}, C_k) \in \text{Res}(C_1, t_1, \ldots, t_{k-1})\}$.

The first line specifies name resolution for simple names. The second line corresponds to compound names. The intention of the second line is to separate the last element from the full sequence so that the recursive definition can be applied again to the first part of the compound name. In practice, name resolution is a stepwise process where each step corresponds to one application of the recursive definition. The process starts from the first element t_1 of a compound name $(t_1, \ldots, t_i)$, and it proceeds by creating one by one the needed intermediate results Res(C_1, t_1), Res(C_1, t_1, t_2), etc.

Name resolution can terminate in three ways: success, failure, and undetermined. If the whole compound name can be processed and the final set Res(C_1, t_1, …, t_k) is non-empty, the resolution succeeded. If the final set is empty, the resolution failed. Finally, if the name cannot be processed because one of the intermediate sets Res(C_1, t_1, …, t_i), where $i < k$, is empty, the result of name resolution is undetermined.

Name communication transmits names from senders to receivers through the infrastructure. The communication takes place between the private contexts. During communication, the in-

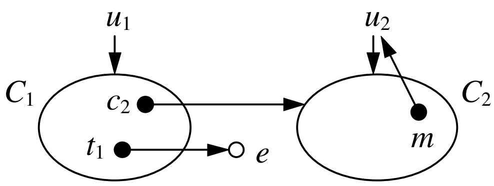

Figure 2 User u_2 cannot receive a synonym for t_1.

frastructure may attempt to optimize the name (e.g. to minimize the resolution time) by replacing it with a synonym. For simple names, this means that the term t_1 may be received as a different term t_2 if both (t_1, e) and (t_2, e) exist in the naming context for some entity e. For compound names, also the number of elements in the sequence may change if the infrastructure is able to optimize the name. The name may change significantly during communication, especially if the communication takes place between users having different local contexts.

Name communication often induces several name bindings at the infrastructure level. Suppose that we have two local naming contexts C_1 and C_2, such that C_1 has a name for C_2 but C_2 has no name for C_1. This is illustrated in Figure 2, where the context C_1 sees the context C_2 and may have access to its names through compound names. On the other hand, the context C_2 cannot see the context C_1 and, consequently, cannot see the term t_1 in C_1. Suppose that u_1 sends the term t_1 denoting the entity e to user u_2. As far as naming is concerned, this is possible because C_1 can denote u_2 with the compound name (c_2, m). However, there is no way for C_2 to offer u_2 a meaningful synonym for t_1 in the given situation.

The solution is to create a new name binding during name communication. The purpose is to produce a new synonym for the term t_1 in u_2's local naming context C_2. Name communication can then proceed normally by delivering the new synonym to u_2. There are two possibilities. The first possibility is to use Bind(C_2, t_2, e) to bind a new term t_2 directly to e. In this case, user u_2 receives the term t_2. The second possibility is to support indirect access through C_1 by using Bind(C_2, c_1, C_1). In this case, user u_2 receives the compound name (c_1, t_1). These two possibilities are indicated by the two alternatives in Figure 3. The choice depends on the implementation. In traditional context-relative naming, u_2 would receive both the context C_1 and the term t_1. This would force u_2 to be aware of two local naming contexts. In the long run, there would be a multitude of naming contexts for every user in the system–obviously an unsatisfactory solution.

4.2 Federated name resolution

Naming federations make name processing more complicated. Naming domains are characterized by those term-entity pairs that are controlled by the domain's naming authority. The authority ensures that the terms conform to the naming convention and that naming policies are

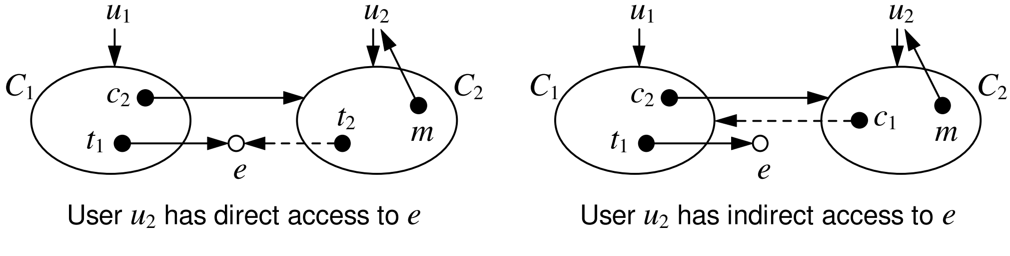

Figure 3 User u_2 gets synonyms for t_1 after new bindings.

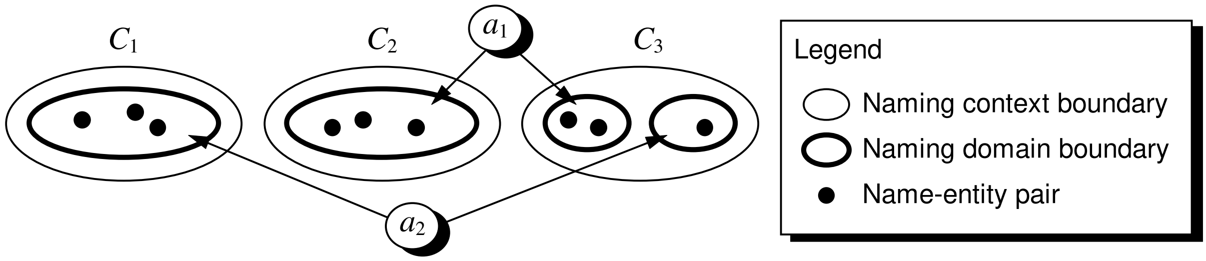

Figure 4 Naming federation is controlled by naming authorities a_1 and a_2.

respected. The federation agreement determines the ownership of all existing and future term-entity pairs in each naming context. Naming contexts may be divided into multiple partitions, and each partition may be controlled by a different authority. Figure 4 illustrates a configuration with two naming authorities and three naming contexts. Authority a_1 controls the context C_2 and part of C_3, and authority a_2 controls the context C_1 and part of C_3.

Naming federation affects the name resolution process because every naming authority determines which term-entity pairs in its domain can be used in name resolution. Since the components of a compound name can be controlled by multiple authorities, name processing has to be distributed. Therefore, name resolution is no longer an atomic operation and name resolution changes accordingly. The result of name resolution can sometimes be undetermined. For example, the communication may fail or some of the naming authorities may decide to hide information from the resolution process.

We use the symbol $\mathrm{Dom}_D(C)$ to denote the set of term-entity pairs $(t, e) \in C$ that are controlled by the naming authority of the domain D. A naming context may have elements from multiple naming domains, but every element must belong to one domain. Hence, for every naming context C,

$$C = \bigcup_D \mathrm{Dom}_D(C).$$

To control the name resolution process, each domain authority defines a view for every principal (i.e. name user) initiating name resolution. The view determines what term-entity pairs can be used in name resolution. We use the symbol $\mathrm{View}_{D,p}(C)$ to denote those pairs $(t, e) \in \mathrm{Dom}_D(C)$ that the principal p is allowed to use during name resolution.

Views have an important role in the semantics of name processing. First, they reflect the dynamic nature of distributed systems, because the expression $\mathrm{View}_{D,p}(C)$ is typically evaluated at run-time and it may change its contents from one evaluation to another. Second, views provide means for hiding naming information. For example, adding a term-entity pair to a context does not necessarily induce any changes in the views. Third, views can also model name transformations that take place in many real systems. An element (t, e) in context C does not necessarily cause the same element to be visible in $\mathrm{View}_{D,p}(C)$. For example, the authority of the domain D may decide that p must not see the entity e and decides to include another pair (t, e') in the view instead. In other words, a view is a realization of a name interceptor that can both transform and conceal naming information.

To model name resolution initiated by a principal p, we must consider the total view seen by p. The symbol $\mathrm{View}_p(C)$ denotes all pairs $(t, e) \in C$ that p can use for name resolution:

$$\mathrm{View}_p(C) = \bigcup_D \mathrm{View}_{D,p}(C).$$

By definition, we have

$\text{View}_{D,p}(C) \subset \text{Dom}_D(C) \subset C$,
$\text{View}_{D,p}(C) \subset \text{View}_p(C) \subset C$

for every principal p and domain D. We have now the means to define name resolution in the presence of naming federation and naming authorities. When a principal p initiates name resolution, the resulting term-entity pairs are characterized by the following recursive definition:

$\text{Res}_p(C_1, t) \subset \{(t, e) \mid (t, e) \in \text{View}_p(C_1)\}$,
$\text{Res}_p(C_1,t_1,\ldots,t_k) \subset \{(t_k, e) \mid \exists\, C_k : (t_k, e) \in \text{View}_p(C_k) \text{ and } (t_{k-1},C_k) \in \text{Res}_p(C_1,t_1,\ldots,t_{k-1})\}$.

The definition is the same as in section 4.1 except that the naming contexts in each resolution step are replaced by views that are visible to the principal p. This way, naming authorities can control the progress of name resolution. The resolution of traditional path names, such as "/usr/share/man", proceeds component by component, and the intermediate results correspond directly to the path name. However, the resolution process and the views involved can sometimes produce very unconventional intermediate results. This may happen, for example, when the resolution step moves from one naming convention to another.

4.3 Information contents of term-entity pairs

The use of federated naming and the cooperation between naming domains changes the structure of naming information. Properties that were previously defined for complete name servers must now be specified for individual names. This allows the naming authority to fully control naming activities. In this section, properties are defined for each term-entity pair individually, but implementations may support grouping for easier management. For example, an implementation with a tree-structured name space may allow properties to be defined for complete sub-trees.

Mutability policy determines when and how term-entity pairs can be changed:

- **Immutable**. No changes are allowed.
- **Conditionally mutable**. Changes are allowed, but the naming authority must perform some additional actions when changes occur. This may involve negotiation with other authorities, announcements, or keeping record of the change for some given time period. Conditionally mutable names usually have additional properties related to the change protocol in use.
- **Mutable**. Changes are allowed.

Similar mutability policies are also used in the ODP Type Repository Function (ISO/IEC, 1996b). In addition to the mutability policy, also the target of the policy must be selected:

- If the policy is **term related**, it controls only the terms, and the corresponding entities can change without restrictions. For example, the pair (t, e) is term-immutable if the term t is guaranteed to denote some entity (not necessarily e). A term-immutable pair can be removed from the context, but this can only happen through name rebinding.
- If the policy is **name related**, both the term and the entity are considered important. For example, a name-immutable pair cannot be removed or changed.

Name-immutable pairs can be safely replicated for implementing efficient and reliable name resolution in distributed environments. Term-immutability, on the other hand, is useful for guaranteeing that a given service can always be reached through a known naming context, but the service implementation can still evolve over time. Sometimes term-immutability is strengthened with conditions that determine the type of the entity that can be bound to the fixed name.

To complete the semantic discussion, we briefly describe two additional properties that characterize term-entity pairs in a naming context. First, every term-entity pair has a controller that controls its life cycle, and decides whether it participates in a given name resolution process or not. The controller's impact on name processing has already been covered with naming domains. Second, an identifier of the entity e may be considered as a property of the pair (t, e). Its purpose is to identify unambiguously the entity. In practice, name resolution is often implemented as a function that transforms the term t to a low-level identifier of e. There may be additional predicates for term-entity pairs in a naming context, but they depend on the selected implementation and name resolutions techniques.

5 NAME SERVICE

Name service is the computational equivalent of the controller of a naming domain. It offers a name service interface for its clients with operations for binding, unbinding, resolving, comparing and communicating names. When using the interface, the client is always attached to an implicit local naming context that determines how names are interpreted and where name resolution starts. The client does not identify its local naming context in any way. Special operations, externalize and internalize, can be used for name communication between two or more users. The externalize operation converts a name into a generic name-string that can be sent to anyone through the infrastructure. The receiver can use the internalize operation to convert the generic name-string back to a name that can be used in the receiver's private naming context.

Naming service is made responsible for creating and interpreting any internal structures that may be present in the name. Typically, compound names are created by separating name components with a separator character, but this choice is implementation dependent. The client simply uses both compound and simple names as if they were character strings.

Name communication does not require the existence of a universal name representation or a naming tree with a global root. The reason is simple: if two infrastructure implementations can send and receive data, they also have the means to mutually agree on any conversations between different name representations. In practice, when two infrastructures implementations are linked together, they also agree on how names externalized in one environment are internalized in the other environment.

6 DISCUSSION

In our proposal, the name service takes responsibility for functions that were previously delegated to applications and are not included in traditional name services. Transparent name translation is needed to provide clients with their single local naming context, and name communication is also necessary. Questions may rise about the feasibility and the performance of the proposed solution. We discuss this topic through four different aspects.

The first aspect is concerned with applications that are aware of naming domains. A traditional name service returns a reference to an entity when given a valid name and a starting context. This approach is very simple, but it cannot be used as such in fault tolerant applications that provide alternative ways to access services when the normal methods cannot be used. This is because real names have different degrees of reliability. For example, there are names that can always be trusted (e.g. my own name), reliable local names, less reliable names within the same domain, and unreliable names in external domains. Fault tolerant naming aware applications must distinguish between these names and maintain name related information to support actions during partial failures. However, traditional name services do not provide such information at name level and, consequently, the application cannot implement fault tolerant behavior even if this was possible in theory.

In the proposed solution, the name service (not the application) is fully aware of the different name categories and the existence of naming federation. Therefore, it can provide the needed fault tolerance behavior for the applications. The fact that predicates are now defined at the name level allows the name service to be flexible. The new and the traditional approaches probably use similar algorithms for overcoming system failures and their resource consumption is at the same level. Hence, the fact that naming federation is now handled within the infrastructure, increases system flexibility and simplifies the applications. The amount of processing and, consequently, the system efficiency remains approximately the same.

The second aspect deals with the quality of service (QoS) concerns. Traditional name services do have some QoS functionality, e.g. related to the name's origin, but usually this is related to the whole service and not to individual names. The proposed solution supports the specification and utilization of QoS attributes at the name level, such as the name's mutability. This offers increased functionality than was previously implemented at the applications, and sometimes was not possible to implement because of the lack of necessary information.

The third aspect deals with ordinary applications without any special concerns for naming or name federation. Such applications usually do not require efficient name processing because they use the name service fairly seldom, mainly to discover system resources at startup time. Hence, they use the name service as such, without any additional tricks. There is no observable performance penalty for these applications when the improved name service is introduced. However, the applications become more intelligent because the improved name service can offer them improved functionality without any need to change the application logic. Typically, such applications become more fault tolerant because now the name service can attempt alternative techniques when a traditional name service would simply report an error.

The last aspect is concerned with the implementation of efficient naming. If the proposed name service is used within a single naming convention, the extra overhead is typically the addition of a prefix before giving the name to the application, and the mapping of this prefix to a context identifier when receiving the name from the application. This kind of additional processing is very small compared to the cost of node-to-node communication. On the other hand, if a name uses an external naming convention or is otherwise complex to process, the extra overhead is significantly bigger. However, the same overhead would probably also incur with traditional name services, but it would have to be implemented within the application.

Since the proposed name service is able to optimize its behavior across applications, caching and other similar techniques can improve its performance. The key issue is the locality of names. The working set for names is usually relatively small because users in one organization tend to work with the same things. Hence, name caching across applications and users can yield significant performance improvements. In addition, applications often organize their names in contexts that reflect the application structure, and name pre-fetching can be implemented on a per-context basis. If resources are accessed by navigating through naming contexts (e.g. in interactive systems), context pre-fetching may result in dramatic performance improvements. As a result, the overall performance of the new naming solution may be superior.

7 CONCLUSION

In this paper, we have presented a federative naming solution. It is based on traditional context-relative naming, but it uses term-entity pairs instead of terms as names. Furthermore, the solution makes a distinction between user-level and infrastructure names. We have also presented a semantic model for the solution to illustrate how federation affects different naming related activities, and we have briefly sketched an example name service.

The main contribution of this paper is to demonstrate that complex naming solutions can be simplified by understanding that names are in reality term-entity pairs. For example, traditional naming solutions, such as global naming, support only "term communication" and the necessary name conversions are left to the applications. The presented approach allows applications and end users to employ names that are best suited for their work. For example, the OMG's name service standard uses a complex data structure for representing arbitrary names. Applications that use the OMG name service must always convert names between this particular data structure and other formats that are used by other system service. On the other hand, our solution takes automatically care of the necessary conversions and, therefore, it is directly compatible with a large number of already existing interfaces using names.

The proposed solution fits well to the RM-ODP's infrastructure model that attempts to delegate all tedious tasks to the computational infrastructure instead of forcing applications to do them. The simplicity of the proposed solution implies that it can also be used inside the infrastructure without significant performance penalty. Compared to traditional naming, we have relocated two name related functions. First, the responsibility for naming federation and for handling the cooperation of multiple naming domains has been moved from the applications to the infrastructure. Second, the quality of service concerns and name attributes are now specified at name level and handled by the infrastructure. The proposed solution is congenial with the ODP Naming Framework. The contribution of this paper is to further develop certain parts of the framework, particularly the ones related to naming federation.

While the solution increases name processing in some cases, we have demonstrated that the efficiency of the new solution is comparable to the existing alternatives. In some cases, the performance of the new solution can even be better by using caching and other techniques that were not reasonable in traditional solutions. Our solution gives enhanced functionality with less overhead costs than would incur if applications themselves implemented the same functionality.

REFERENCES

ISO/IEC (1995a) *Reference Model of Open Distributed Processing Part 2. Foundations*. International Standard IS10746-2.

ISO/IEC (1995b) *Reference Model of Open Distributed Processing Part 3. Architecture*. International Standard IS10746-3.

ISO/IEC (1996a) JTC1/SC21 N 10390, *ODP Naming Framework*, May 1996.

ISO/IEC (1996b) JTC1/SC21 N 10389, *ODP Type Repository Function*, May 1996.

Object Management Group (1994) *Common Object Services Specification, Volume I*. OMG Document 94.1.1, Framingham, Massachusetts, USA.

van der Linden, R. (1993) *The ANSA Naming Model*, APM.1003.01. Architecture Projects Management Ltd., Cambridge, United Kingdom.

Siegel, J. (1996) *CORBA Fundamentals and Programming*. John Wiley & Sons Inc., New York, USA.