An Overview of UNP

Larry L. Peterson[1]
University of Arizona

Abstract

The paper gives a brief introduction to UNP: the Universal Naming Protocol. UNP specifies a generic *attribute-based* name server upon which a variety of high-level naming services—including white- and yellow-page services—can be built.

1 Introduction

This paper reports on a new protocol, called the Universal Naming Protocol (UNP), that is being designed for use in the DARPA/NSF/NASA Internet. UNP provides a common ground for sharing naming information among the organizations that make up the Internet, while preserving each organization's autonomy. A full specification of UNP and a demonstration implementation are expected to be released later this year.

UNP supports an *attribute-based* naming paradigm in which clients identify objects by giving a set of attributes or properties that describe the object [Debr87]. The set of attributes used by a client to describe a particular object is called an *attribute-based name*. A name server that supports the attribute-based naming paradigm maintains a database of attributes for each of a collection of objects, and when given an attribute-based name, computes the set of objects described by the attributes. For example, "domain name is bodoni.arizona.edu", "internet address is 192.12.69.22", "architecture is 68020", and "mips is 2" are attributes that might be registered for a processor. A client might submit the latter two attributes to learn the first two attributes, or vise versa.

UNP specifies the behavior of a single name server; it does not provide a naming service. In particular, UNP specifies the semantic behavior of a name server and the syntactic form of queries; it does not specify how a collection of name servers cooperate to define a name space, nor does it specify the syntax of names accepted by clients of a UNP name server. Implementing a naming service on top of UNP involves defining conventions for the data that is to be put into one or more UNP server and writing client programs that take advantage of this data. For example, an organization could provide a *white-page* service by loading a UNP server with attributes for users, and it could provide a *yellow-page* service by stocking a UNP server with attributes for computational resources such as hosts, printers, file servers, databases, and so on. Moreover, whether or not multiple UNP servers cooperate to provide a wide-area naming service depends upon the extent to which individual UNP servers contain attributes that describe other UNP servers, and in turn, how client programs take advantage of this information.

In other words, UNP moves knowledge of the naming service from the name server to the client program. One important consequence of this design is that a single UNP server can participate in multiple high-level naming services. For example, a UNP server operated by an autonomous organization might function as the

[1]Author's address: Department of Computer Science, University of Arizona, Tucson, AZ 85721.This work supported in part by National Science Foundation Grant DCR-8609396 and National Aeronautics and Space Administration Grant NCC-2-561.

organization's component name server in the domain naming system (DNS) [Mock87a]; participate in one or more global user directory services [ISO88, Pete88, Pick79, Solo82]; serve as the organization's resource locator [Pete87, Symb86]; implement the existing UNIX name mapping library; and support direct queries from users and programs.

While UNP's underlying architecture is designed to allow a single UNP server to serve as the back-end to multiple, heterogeneous naming clients, we envision specific implementations of UNP also being able to provide a common interface to multiple, heterogeneous name servers. Such implementations would allow a client to query a UNP server using a UNP-style naming query, where the UNP server would resolve the query by engaging one or more existing (and possibly remote) name servers. In doing so, a UNP server plays the same "name server clearinghouse" role as the the HNS system developed at the University of Washington [Schw87].

2 Architecture

This section gives a structural overview of UNP's three major components: a *database*, an *interpreter*, and an *access protocol*. The database contains a set of attributes for each of a collection of resources. The interpreter defines the syntactic form and the semantic meaning of queries submitted to a UNP server. That is, the interpreter translates a given "naming program" into a sequence of operations on the database. The access protocol provides a method for submitting naming programs to a UNP server and retrieving the results from a UNP server.

2.1 Database

A UNP database contains information about a set of *objects*, each of which corresponds to either some "external" resource that exists outside the name server—e.g., a printer, a processor, a user, another name server—or some "internal" abstraction that is meaningful only within the name server itself—e.g., functions, types, contexts. A UNP object, in turn, is defined by a set of *attributes*, each of which is a syntactic entity that denotes a property or characteristic of the resource or abstraction represented by the object.[2] For example, the following set of attributes might define an object that corresponds to a printer in the local computing system:

```
{(name,llp),(address,lw6),(architecture,postscript),(location,723),
    (pagesperminute,8), (owner,llp)}
```

In addition, two high-level structures, *types* and *contexts*, are imposed on the database objects. The following subsections describe attributes, types, and contexts in more detail.

2.1.1 Attributes

Although from the user's perspective an attribute is given by a *(tag,value)* pair—e.g., attribute `(address,lw6)` has the tag `address` and the value `lw6`—the attributes stored in the database are given

[2]Except when necessary to clarify the discussion, we do not distinguish between the UNP object and the resource it represents, nor do we distinguish between an attribute and the property it denotes.

by the following triple:

attribute = (tag, compound_value, preference)

Attribute tags are simple ascii character strings that label an attribute. Tags serve two major purposes: they are used as "keys" when searching for an object in the database, and they serve as "hints" about the meaning of an attribute value returned to a client. A collection of well-known attribute tags are defined as part of the UNP specification; additional tags are expected to be added to this set over time.

Attribute values denote the actual property of an object; e.g., its address, name, location, or load. Values are represented in the database as compound entities defined by the following structure:

compound_value = (print_value, compare_value, indirect_value)

An attribute's print_value is an ascii character string that gives a "printable" or "user-readable" version of the value; i.e., it can be returned in response to a client's query. Because UNP restricts each object to one occurrence of a given attribute tag, an attribute value may encode a set of properties. For example, attribute `(address, (192.12.69.1,128.196.6.1))` denotes a processor's addresses. An attribute's compare_value gives the version of the value against which client queries are compared. Compare_values are represented by Ascii character strings that can be interpreted according to a grammar that is rich enough to encode sets of integers or sets of strings. Note that in many cases, an attribute's print_value and compare_value are the same. Finally, an attribute's indirect_value is a set of references (internal pointers) to other objects in the database. An attribute's indirect_value is defined whenever the attribute "expands" into another object in the database. For example, one of the attributes associated with a printer object might be an `owner` attribute that identifies the user that owns the printer. The indirect_value associated with this attribute might point to the database object representing the user that owns the printer.

Attribute preferences provide a mechanism for distinguishing among various "qualities" of attributes. Intuitively, it is reasonable to give some attributes more importance than others when resolving a name. UNP supports six preferences that are defined along two orthogonal dimensions.

First, it may be the case that an attribute describes an object, but this fact is not known to the name server. Suppose attribute *a* is guaranteed to be registered for every object it describes and attribute *b* is not so guaranteed. Since the name server is working only with information that is registered, but its responses are being interpreted as statements about certain objects being described by certain attributes, it is intuitive that we have more faith in responses based on the attribute *a* than on responses based on *b*. Moreover, we may wish to treat specially those attributes that identify objects unambiguously; e.g., social security numbers, serial numbers. Thus, we partition the set of attributes into the following three classes:

- An attribute is *unambiguous* if and only if it denotes a property that is true for at most one object.
- An attribute is *complete* if and only if it is registered for every object for which it denotes a true property.
- An attribute is *valid* if and only if it denotes a property that is true for the object for which it is registered.

Note that *unambiguous* ⇒ *complete* ⇒ *valid*.

Second, it is reasonable to consider the time interval over which an attribute describes an object, thereby accommodating changes in the property of objects over time. For example, an address might change for

an object, while the serial number and architecture remain constant throughout the lifetime of the object. Intuitively, one can imagine preferring attributes that are always true for an object over attributes that change over time, and of those attributes that change, attributes that are currently true are preferred over attributes that were true at one time, but not necessarily at the current time. Thus, we partition attributes into the following three classes:

- An attribute is *always* if and only if it denotes a property that remains true over the lifetime of that object.
- An attribute is *now* if and only if it denotes a property of that is true for that object at the present time.
- An attribute is *sometime* if and only if it denotes a property that is true for that object at sometime, but not necessarily at the present time.

Note that *always* $\Rightarrow$ *now* $\Rightarrow$ *sometime*. Also, if not explicitly defined by the client that registers the attribute with the server, an attribute's preference is assumed to be *valid* and *sometime*.

2.1.2 Types

A type system is the first high-level structure imposed on the object database. Intuitively, a type represents a set of objects that exhibit structural similarities. For example, all printers have certain characteristics in common: they all have a pages per minute speed, a well-defined architecture, a mnemonic name, and an address. Formally, type T is denoted by a nonempty set of tags $\{tag_1, tag_2, \ldots, tag_n\}$. We refer to the set of tags that define a type as the type's *necessary tags*. An object is of type T if and only if the tags of its attributes are a superset of the tags in T. Thus, an object with n attributes belongs to 2^n different types. Note that an object's type is not explicitly stored as part of the object, but rather, an object is determined to be of a given type by comparing the tags of its attributes with the tags that define the type.

Types are represented as first class objects in a UNP database. That is, just as there are user objects and printer objects in the database, there are a collection of "type objects", each of which is defined by a set of attributes. Thus, every time we introduce a new type $T = \{tag_1, tag_2, \ldots, tag_n\}$, we are able to give the database object that represents type T as follows:

$\{$(name,T), (necessary_tags,(tag_1,tag_2,...,tag_n))$\}$

Moreover, because the collection of type objects in a UNP database share the same set of attribute tags, they are of the same type. The type of all type objects, conventionally denoted *Type* : *Type*, is defined by the following UNP object:

```
{(name,type), (necessary_tags,(name,necessary_tags))}
```

That is, all type objects have `name` and `necessary_tags` attributes.

Note that while in the abstract there are 2^N types associated with a database if there are N distinct tags, only a small subset of these types are of any practical use. It is this small collection of types that are actually "named" and stored as objects in the database. On the other hand, when a particular type is not explicitly stored in the database, it may still be used implicitly in a query. This is because every query involves the specification of a set of attributes; the attribute tags used by the query specifies a type which limits the query to an interesting set of objects.

2.1.3 Contexts

The second high-level structure imposed on objects, called contexts, partitions the set of objects in a UNP database based on the authority that is responsible for administering the objects. For example, a UNP server implemented at a particular site might support one "system" context, a collection of "application" contexts, and a set of "user" contexts. In this example, the system context would be owned by a trusted system authority (e.g., super-user) and contain a UNP object representing each resource in the computing system; each application context would be administered by the user or group responsible for the application and contain UNP objects representing the abstractions used by the application (e.g., type objects); and each user context would be administered by a user and contain a collection of user-specific objects such as private aliases, private applications, and so on. Intuitively, the owner of a given context decides what objects to put in the context and sets the policy governing how information in the context may be accessed. Also, the contexts in a given name server are independent, although one context might define an object that represents another context.

Contexts are also first class objects in a UNP database, and as such, have the following type definition:

```
{(name,context), (necessary_tags,(name,server,owner))}
```

That is, a context has a mnemonic name (denoted `name`) it is implemented in some UNP server (denoted `server`), and it is owned by some user (denoted `owner`). Note that in many cases, the `server` and `owner` attributes might include references to other objects in the database.

There exists a distinguished context in each UNP name server, called the *meta context*. Intuitively, the names of the objects contained in the meta context play the role of "reserved words" in UNP. A name server's meta context contains a database object corresponding to each context (including itself) in the server's database; the type object for types, contexts, and functions; and objects corresponding to two distinguished functions named `equal` and `all`. (Functions are defined in Section 2.2.)

In addition to the meta context, there exists a *session context* for each client connected to a name server. A client's session context is temporary: it is created when the client establishes a connection to the server, it holds intermediate results generated during the connection, and it is destroyed when the client terminates the connection.

2.2 Interpreter

The interpreter parses a naming program submitted by a client and invokes a sequence of operations on the underlying database. The following is a partial list of the operations can be invoked on the object database:

- *SELECT(Object_Set, Tag, Op_Code, Compare_Value)*: For the given set of objects, return the subset for which the specified tag and compare_value satisify the op_code.
- *PREFER(Object_Set, Tag, Preference)*: For the given set of objects, return the subset for which the attribute that satisfy the given tag, preference pair.
- *PROJECT(Object_Set, Tag)*: For the given set of objects, return the attribute for each object in the set that has the specified tag.

- *DEREFERENCE(Object_Set, Tag)*: For the given set of objects, return the set of objects referenced by the object's attribute with the specified tag.
- *ADD_ATTRIBUTE(Object_Set, Attribute)*: Add the given attribute to each element of the specified set of objects.
- *DELETE_ATTRIBUTE(Object_Set, Attribute)*: Delete the specified attribute from each element of the given set of objects.
- *SELECT_TYPE(Object_Set, Type_Name)*: For the given set of objects, return the subset that are of the named type.
- *SELECT_CONTEXT(Context_Name)*: Return the set of objects in the named context.
- *ADD_OBJECT(Object, Context_Name)*: Add the given object to the named context.
- *DELETE_OBJECT(Object, Context_Name)*: Delete the specified object from the named context.

In addition to these database operations, the interpreter also supports a small collection of set operations (e.g., *UNION*, *INTERSECTION*), a conditional, and recursion.

The program submitted by a client to a UNP server minimally contains a naming *function* applied to an attribute-based name. The result of this function can optionally be composed with other naming functions. Like types and contexts, functions are first class objects; i.e. each function is described by a set of attributes. The following specifies the function type:

```
{(name,function), (necessary_tags,(name,program))}
```

That is, each function object has an attribute that gives its name and an attribute that gives the UNP naming program that implements the function.

Making functions first class objects has two desirable consequences. First, the implementation of a function is given by an attribute that is stored in the database, thereby simplifying the interpreter. For example, a UNP server contains the two distinguished functions objects named `equal` and `all`: The former function returns at most one object that is matched by all the specified attributes, while the later function returns any object that matched one of the specified attributes. Second, a UNP server is easily extended to provide new naming services by loading new function objects into its database. These function objects can, in turn, be passed between clients and servers.

Associated with the interpreter is an *environment* consisting of the meta and session contexts. This environment is used to resolve type, context, and function names. That is, when trying to locate a type, context, or function object by name, the interpreter applies the `equal` function to the name and the set of objects in the meta context and then the session context. The interpreter is satisfied with the first such object it finds. For example, suppose a UNP server supports a system context that contains objects for all the users and computational resources in the local computing system and an application context containing a collection of type and function objects that implements a particular yellow-page service. A client that wishes to apply the application types and functions to the objects in the system context might connect to the UNP server and "load" the objects in the application context into its session context. The client would then apply interesting functions in the session context to the set of objects in the system context.

Rather than define the interpreter in full detail, we give some examples that illustrate the expressiveness of the programs it accepts. First, the program

```
get_processor(architecture=68020, load<2)
```

reports all the attributes known for those processors that have both 68020 architectures and a load less than two. The function named `get_processor` is given by an object in the database that is of type `function` and has the attribute `(name,get_processor)`. The `program` attribute for that object would contain a UNP program that selects all objects of type `processor` and then applies the `equal` function to those objects and the given attributes. The function might, for example, return the set of objects:

```
( ( (name,bodoni.arizona.edu),
    (address,192.12.69.22),
    (architecture,68020),
    (load,1),
    (speed,2),
    (owner,John Doe) ),
  ( (name,jenson.arizona.edu),
    (address,192.12.69.33),
    (architecture,68020),
    (load,0),
    (speed,2),
    (owner,John Smith) ) )
```

Note that the attributes in an attribute-based name taken as an argument by the naming functions differ from the attributes contained in the UNP database. The database attributes denote a property of an object. For example, `(load,1)` exactly and completely describes the load of some processor. The attributes used to select objects—i.e., those passed to naming functions as arguments—are indefinite attributes; they don't necessarily describe an object completely. Instead, indefinite attributes specify a range of potential values for an attribute. Thus, `(load<2)` matches all load attributes with a value less than two.

Second, if the client is only interested in learning the internet addresses of the processors returned by the first query, the preceding function might be composed with the `project` function, where `project` is the syntactic representation of the database operation *PROJECT*.

```
project(get_processor(architecture=68020, load<2), address)
```

Finally, if the client wants to learn about the owner of those processors that satisfy the original query, the following program might be submitted

```
dereference(get_processor(architecture=68020, load<2), owner)
```

perhaps yielding the result

```
( ( (name,John Doe),
    (mailbox,jd@arizona.edu),
    (login,jd),
    (phone,555-1234),
    (office,723),
    (workstation,bodoni),
    (printer,lw6) ),
  ( (name,John Smith),
    (mailbox,js@arizona.edu),
    (login,js),
    (phone,555-4321),
    (office,732),
    (workstation,jenson) ) )
```

Note that extra white-space was introduced into the results is to make the examples more readable; each naming function simply returns the set of attributes for each of the set of objects matched by the query.

2.3 Access Protocol

The access protocol defines a method for submitting naming programs to a UNP server and retrieving the results from a UNP server. The access protocol does four things. First, it manages client connections via some transport protocol; e.g., TCP [USC81], UDP [Post80], or RPC [Sun86]. Second, it authenticates the client in order to establish what objects the client is allowed to access and modify. Third, it initializes (at connect time) and deletes (at close time) the environment used by the interpreter. Fourth, it pre-processes the naming programs submitted to the interpreter and the results returned from the interpreter. This pre-processing supports an extended syntax that removes extra white-space from naming programs and inserts extra white-space into results. The reason for this pre-processing is that UNP is designed to be used directly by users, as well as by client programs. For example, for the last example given in the previous section, the access protocol would accept the program:

```
dereference(
     get_processor(architecture=68020, load<2),
     owner
)
```

It would also return a result of the form:

```
(
     (
          (name,John Doe),
          (mailbox,jd@state.edu),
          (login,jd),
```

```
            (phone,555-1234),
            (office,723),
            (workstation,bodoni),
            (printer,lw6)
        ),
        (
            (name,John Smith),
            (mailbox,js@state.edu),
            (login,js),
            (phone,555-4321),
            (office,732),
            (workstation,jenson),
            (printer,lw5)
        )
    )
```

Note that we distinguish between the syntax accepted by the interpreter and the syntax accepted by the access protocol because a given UNP server is free to support multiple access protocols. That is, additional protocols may be used to relay queries to a UNP server, where such queries are translated in a naming program that can be submitted to the interpreter. For example, one might implement the DNS protocol that accepts messages at the well-known DNS address, translates each such message into a UNP program, submits the program to the interpreter, and translates the results into a reply message [Mock87b]. In this way, a single UNP server can appear to implement any of a number of existing naming protocols, and as a consequence, participate in several high-level naming services.

3 Policy

In addition to the basic architecture defined in the previous section, one must define a set of policies that govern how UNP is to be used. We classify these policies as being either *local* or *global*. In the case of local policies, UNP only provides the necessary mechanism; the policy decision is left to each organization that runs a UNP server. Global policies, on the other hand, specify a set of rules that all organizations participating in Internet-wide naming services must observe. Although several global policies are included in the UNP specification, they are defined as external conventions rather than an intrinsic part of the UNP architecture. This is because the global policies are expected to be modified and extended from time to time.

3.1 Well-Known Types

In addition to those internal object types that are essential to the operation of a UNP server—`type`, `function`, and `context`—UNP also specifies five external object types—`user`, `organization`, `processor`, `server`, and `printer`—each of which is defined in terms of a collection of necessary tags. A given organization is free to define additional types, including both subtypes and supertypes of these five

well-known types. Note that if type T_1 is a subtype of type T_2, then the set of tags associated with type T_1 is a superset of the tags associated with T_2.

In order to define these five types, UNP also defines the corresponding set of tags. For each tag, the UNP specification gives a description of how the value associated with the tag is to be interpreted. These descriptions have varying degrees of precision. For example, certain tags are meant to prescribe an interpretation of the value that is uniformly applied throughout the Internet (e.g., Internet addresses). Other tags prescribe a loose interpretation of the attribute value. Such a weak specification implies either that the value is only meant to be read by users (e.g., a person's name) or that a more precise interpretation is left as a local policy matter (e.g., processor load).

3.2 Well-Known Functions

As with types, UNP specifies a set of useful naming functions. This set includes the distinguished functions `equal` and `all`, the suite of functions defined by the Profile naming service [Pete88], and a suite of functions for identifying computational resources; e.g., `get_processor`, `get_printer`, and so on. Also, a simple name-to-address mapping function is specified. This later function is trivially represented in a UNP database by the following object:

```
{(name,get_addr),(program,project(equal(arg(1)),address))}
```

Note that by defining well-known functions that correspond to existing naming systems one is able to use a UNP server as an umbrella for those systems. For example, one could define the function named `dns` as a front-end for the domain naming system. In this case, the implementation of the function would be given by a call to the domain naming system rather than a UNP program.

4 Implementation

UNP allows for a wide latitude in implementation. Open implementation issues include the representation of the database, the implementation of the interpreter, the mechanism used to checkpoint updates of the database to nonvolatile storage, and the mechanism used to replicate a UNP server across multiple machines. A demonstration UNP server, called Univers, is currently being implemented at the University of Arizona. In order to demonstrate the universality of UNP, we plan to implement several existing naming services, including Profile and X.500, on top of UNP.

Acknowledgements

Mic Bowman, Karen Sollins, and Andrey Yeatts have made significant contributions to the design of UNP.

References

[Debr87] S. Debray and L. Peterson. *Reasoning About Naming Systems.* Tech Report TR 87-16, Univ. of Arizona, Tucson, Ariz, July 1987. Submitted for Publication.

[ISO88] ISO. Information processing systems: open systems interconnection—the directory—overview of concepts, models, and service. Draft International Standard ISO 9594-1:1988(E).

[Mock87a] P. Mockapetris. Domain names—Concepts and Facilities. *Request For Comments 1034*. USC-ISI, Marina del Rey, Calif., Nov. 1987.

[Mock87b] P. Mockapetris. Domain names—Implementation and Specification. *Request For Comments 1035*. USC-ISI, Marina del Rey, Calif., Nov. 1987.

[Pete87] L. Peterson. A Yellow-Pages Service for a Local-Area Network. In *Proceedings ACM SIGCOMM '87 Workshop*, Aug. 1987, 235-242.

[Pete88] L. Peterson. The Profile Naming Service. ACM Trans. on Computer Systems 6, 4, Nov. 1988, 341-364.

[Pick79] J. Pickens, E. Feinler, and J. Mathis. The NIC Name Server—A Datagram Based Information Utility. In *Proceedings 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, Aug. 1979.

[Post80] J. Postel. User Datagram Protocol. *Request For Comments 768*, USC Information Sciences Institute, Marina del Ray, Calif., August 1980.

[Schw87] M. Schwartz, J. Zahorjan, and D. Notkin. A Name Service for Evolving Heterogeneous Systems. In *Proceedings of 11th Symposium on Operating System Principles*, Nov. 1987, 52-62.

[Solo82] M. Solomon, L. Landweber, and D. Neuhengen. The CSNET Name Server. *Computer Networks 6*. 1982, 161-172.

[Sun86] Sun Microsystems Inc. *Remote Procedure Call Programming Guide*. Feb. 1986.

[Symb86] Symbolics Inc. *Networks*. June 1986.

[USC81] USC Information Science Institute. Transmission Control Protocol. *Request For Comments 793*, Marina del Ray, Calif., September 1981.