What's in a name?
A Distributed, Federated Naming System in Python.

Martin Chilvers, David Arnold, Andy Bond, Richard Taylor
19th April 1996

CRC for Distributed Systems Technology
Level 7 Gehrmann Laboratories
University of Queensland
St. Lucia, Queensland 4072
Phone: +61 7 3365 4310
Fax: +61 7 3365 4311

Email:
arnold@dstc.edu.au
bond@dstc.edu.au
martin@dstc.edu.au
taylor@dstc.edu.au

Abstract

Nearly all computer applications use some form of naming system; they just might not realise it! File systems use names to locate files on storage devices, and spreadsheets use names to associate cells with values and macros. We discuss the generic model for naming systems proposed in the X/Open Federated Naming Specification (XFN) and highlight its advantages over traditional approaches. We also look at a prototype implementation of the specification written in Python for the Hector distributed object environment.

1 Introduction to Naming Systems

Naming systems are everywhere! A file system uses names to refer to files and directories (e.g. the name '/usr/local/bin/python' refers to the file that resides at a particular location on a storage device), and a spreadsheet uses names to refer to cells and macros (e.g. the name 'B4' refers to some representation of the contents of a particular cell).

Some applications even allow the user to specify combinations of names from different systems. For example, in Excel on the Apple Macintosh you can refer to a cell in a spreadsheet by using the name 'HardDisk:Finance:Budget:Fiscal1992:HouseHold!B4'. This name has two components; the first, 'Hard-Disk:Finance:Budget:Fiscal1992:HouseHold' is the pathname of the spreadsheet in the file system, and the second, 'B4' is the name of a cell within that spreadsheet. The name therefore spans multiple naming systems (in this case where *multiple* equals two!), and in naming circles is known as a *composite name*. Another example is the Unix command rcp which accepts names such as 'myhost:/usr/local/bin/python'. Again, the name is a composite name with two components; the hostname 'myhost', and the pathname '/usr/local/bin/python'.

The problem is that each application defines its own rules for composing, parsing and looking up (resolving) these composite names. If you are familiar with Excel on the Macintosh you are used to composing names with '!', not ':' as required by rcp. The user is also required to remember which applications permit composite names, and which do not (e.g. the Unix command cp). A further problem is that applications which support composite names need to be changed every time a new type of naming system is added.

Now consider these problems in terms of a heterogeneous distributed environment where the applications themselves are potentially spread across a global network. Every system in the environment is capable of defining many different sets of naming rules leaving users and programmers with the impossible task of trying to incorporate and coordinate them all.

2 XFN Federated Naming

One solution to these problems is proposed in the X/Open Federated Naming specification[1]. This specification defines a general model for naming systems that fits a large class of naming problems. It also provides a model for joining arbitrary naming systems together to form what is known as a *federated naming system*. Such a system allows composite names to be resolved across multiple autonomous naming systems and provides two major advantages over the traditional approach to naming:-

1. A single uniform naming interface is provided to clients (distributed or otherwise.)
2. Adding new naming systems does not require changes to applications or other naming systems.

The basis of the XFN model is the mapping of a composite name to a value. In XFN terms the mapping is called a *binding* and the value bound to a composite name is called an *object reference* or (more usually) a *reference*. The naming system provides facilities to resolve a composite name to find its associated reference, and also for binding, unbinding, and renaming names etc. In the Unix file system example, the composite name 'myhost:/usr/local/bin/python' is said to be bound to a reference to a file. The reference contains enough information to allow the client of the naming system (in this case, the file system itself) to locate the file on some storage device.

3 Elements of the XFN Model

The XFN model specifies application programming interfaces (API's) for naming system functions (defined in, surprise, surprise, C!), along with the abstract data types of the objects that are passed across these interfaces. We will introduce the fundamental interfaces and data types of an XFN naming system, but will leave some of the optional facilities as an exercise for the interested reader!

3.1 Names

There are three kinds of names specified in XFN.

1. A *composite name* contains zero or more components composed according to rules defined by XFN, and can span multiple naming systems. Each component of a composite name is either a compound name or an atomic name.
2. A *compound name* is a name from a single naming system that contains one or more atomic names composed according to a naming system specific syntax or *naming convention* (e.g. the Unix pathname 'usr/local/bin/python' is a compound name consisting of four atomic names, 'usr', 'local', 'bin' and 'python', where the naming convention specifies that the order of composition is left to right, and the separator is '/').
3. An *atomic name* is an indivisible component of a compound name such as 'usr' in the above example.

3.2 References (aka object references)

A reference is the value associated with a composite name. It consists of a *reference type* and a list of one or more communication endpoints (*addresses*). The reference type is intended to identify the type of the object that is referenced. An address is an item which can be used with some communication mechanism to invoke operations on the object.

3.3 Bindings

A binding is the association of an atomic name with an object reference.

3.4 Contexts

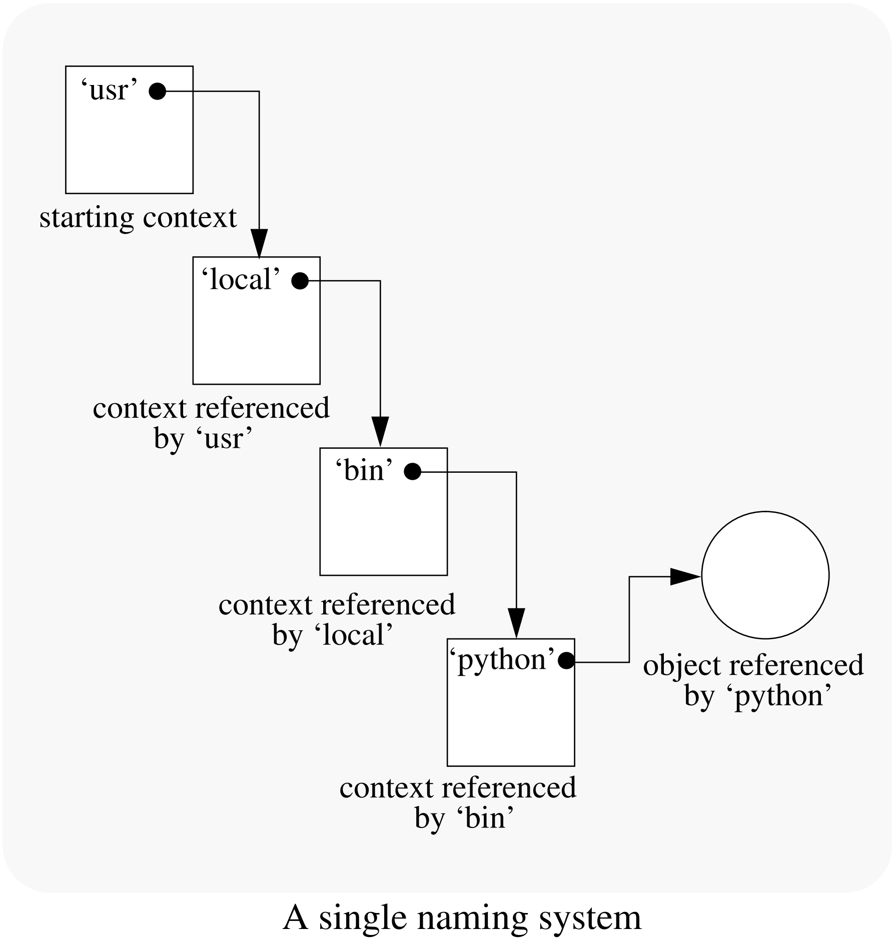
Contexts are the major (and we mean *major*!) component of the XFN model, and the API used to interact with them is known as the *base context interface*. A context is a set of bindings with distinct atomic names bound to references. Every context has an associated naming convention, and provides a lookup operation so that it can participate in composite name resolution. Atomic names in a context can be bound to references to other contexts (called, not surprisingly *sub-contexts*!), thereby giving rise to compound names. In the earlier example, the atomic name 'usr' is bound in some context to a sub-context in which 'local' is bound to another sub-context, and so on.

4 Name Resolution

The fundamental facility offered by a federated naming system is the resolution of composite names. Resolution always begins in some starting context and continues through zero or more *intermediate* contexts to the *target* context, where the lookup or other operation (e.g. bind, unbind) is performed. It is important to note that it is the context objects in individual naming systems that do the bulk of the work when resolving composite names. The federating naming system's job is to guide the resolution across the naming system boundaries.

As an introduction to name resolution, we will look at two examples. The first is a simple case where a composite name is resolved within a single naming system; the second a more complex example of resolving a name that spans multiple naming systems.

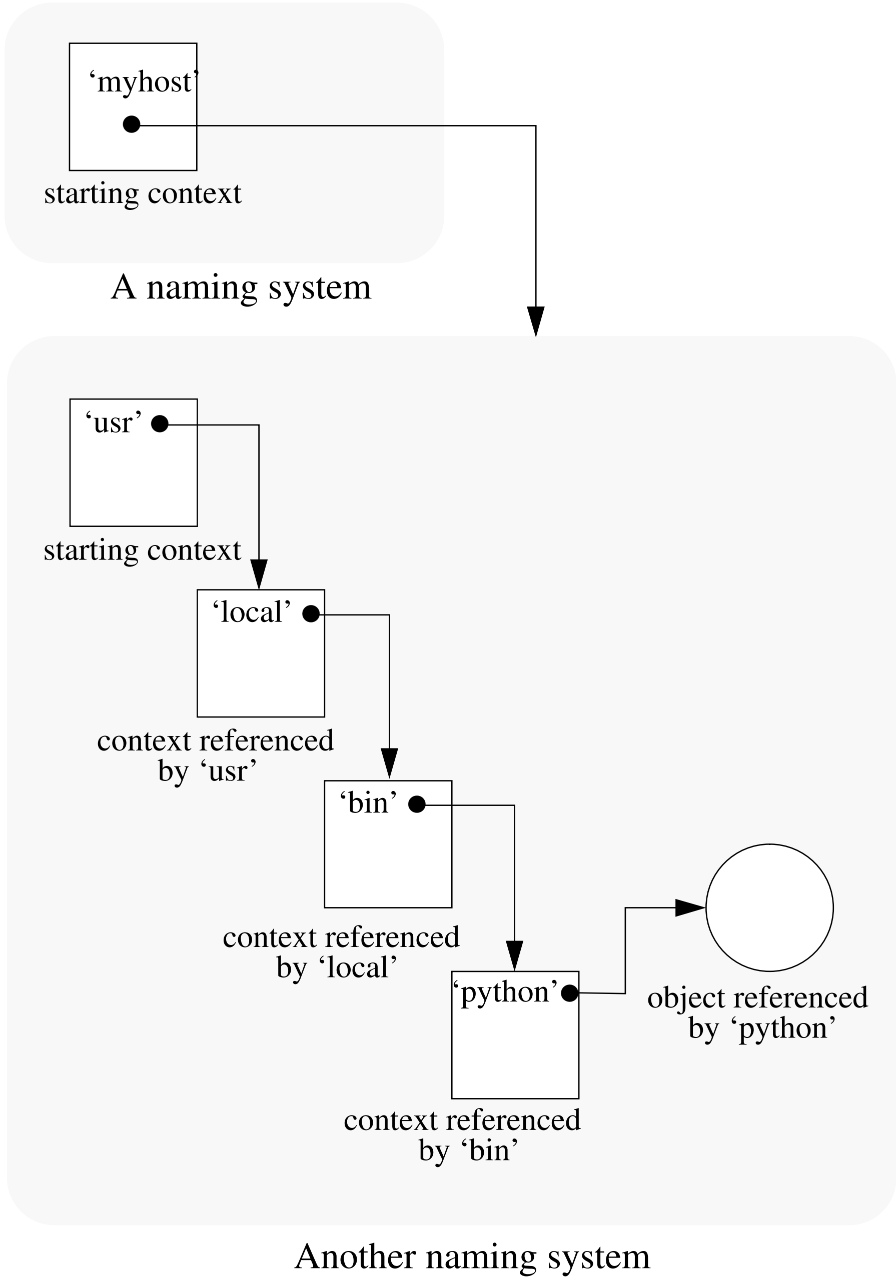
Figure 1: Composite name resolution within a single naming system



Example 1: Resolution within a single naming system (see Figure 1).

Suppose we have a composite name consisting of a single component containing the compound name 'usr/local/bin/python'. Resolution begins in the starting context of a naming system and the atomic name 'usr' is resolved to a reference to a sub-context. The rest of the name 'local/bin/python' is then handed to the sub-context where it resolves the atomic name 'local' to a reference to another sub-context. Resolution continues in this manner through the other intermediate contexts until we reach the target context which contains the name 'python' bound to a reference that gives us the information required to locate the appropriate file object.

Figure 2: Composite name resolution across multiple naming systems.



Example 2: Resolution across multiple naming systems (see Figure 2).

Suppose we have a composite name with two components ('myhost', 'usr/local/bin/python'). The federating naming system begins resolution in some initial naming system passing it the entire composite name. The initial naming system resolves the first component of the composite name ('myhost') and finds the reference bound to it. By looking at the reference type the naming system determines that it cannot continue the resolution any further and passes the reference back to the federator along with the unresolved portion of the name ('usr/local/bin/python'). The federating naming system now extracts the address of the next naming system from the resolved reference, and contacts that system handing it a composite name consisting of the single component 'usr/local/bin/python'. Resolution in this naming system continues as in example 1 and a reference to the appropriate file object is returned.

5 Motivation - Distributed Systems in Python

Our interest in naming systems is motivated by our current work involving the implementation of an open, distributed object environment (codenamed Hector)[2]. Hector is implemented in Python and is intended to support the development and deployment of scalable distributed applications. It is also intended to be dynamically extensible to incorporate future communications protocols.

In Hector, application objects live in a support environment known as a *capsule*. Capsules provide a variety of services to objects and must include a mechanism to allow objects to locate other (possibly) distributed objects and the services that they provide. This mechanism must be dynamic to allow objects to change their physical location and to vary the communications protocols that can be used to contact them.

This is exactly the role performed by a federated naming system. Given the name of an object we can use the naming system to retrieve an object reference and obtain the addresses that can be used to communicate with the object itself. Either the object, or some entity responsible for it, can update the information in the naming system should either its location or means of contact change.

Capsules in Hector therefore offer a *name service* to the objects that live within. The name service is responsible for communicating with autonomous naming systems (called *name servers* in Hector) and controlling the resolution of composite names that span multiple servers. The name service is therefore Hector's federating naming system.

Python was chosen as the implementation language for our distributed environment (and our naming system) primarily for its support for rapid prototyping and the relatively ease with which it can be interfaced with existing languages and libraries (and because we all like it, and it's not C!).

6 Building a Federating Naming System for Hector

The XFN specification is biased towards the C world. All of the XFN interfaces are defined in C and to be able to participate in an XFN federation, implementations are required to support one or more of the standard XFN protocols (currently DCE and ONC).

In order to implement a naming system in Python we therefore had to read the specification (wearing Python coloured glasses!) and separate the semantics of an XFN naming system from its C syntax. We decided to forego initial support for the XFN protocols, opting instead to implement the XFN semantics in a Python object model and to provide an XFN API 'wrapper' to make future integration with the outside (C) world relatively straightforward.

6.1 Implementing the XFN Abstract Data Types

The first stage of the implementation was to construct Python classes for the basic XFN abstract data types. Table 1 shows the mapping of the XFN data types to the Python classes that implement them.

Table 1: Mapping of XFN Abstract Data Types to Python Classes

Abstract Data Type	XFN Type	Python Classes
Compound name	FN_compound_name_t	CompoundName
Composite name	FN_composite_name_t	CompositeName
Reference	FN_ref_t	Reference, ContextReference
Status	FN_status_t	Status
Context	FN_ctx_t	Context, ContextHandle, InitialContext, InitialContextHandle, ServerContext, ServerContextHandle

6.1.1 class CompoundName:

The *CompoundName* class contains a sequence of one or more atomic names composed according to some naming convention. Methods are provided to create a compound name from a string, to reproduce the string representation of the name, to access the individual atoms of the name, and to return a new compound name consisting of a specific slice (XFN only specifies functions to return prefixes and suffixes of names, but with __getslice__ we get the extra flexibility for free!).

6.1.2 class CompositeName:

The *CompositeName* class contains a sequence of zero or more components composed according to rules defined by XFN. Methods are provided to create a composite name from a string or a compound name, to reproduce the string representation of the name, to access the individual components of the name, and to return a new composite name consisting of a specific slice.

6.1.3 class Reference:, class ContextReference(Reference):

Two classes are used to implement XFN references. Standard references are represented by the *Reference* class which contains a reference type and list of addresses. References that refer to sub-contexts are represented by the *ContextReference* class (derived from *Reference*) which contains a reference type and a single context address (know as a *context handle*).

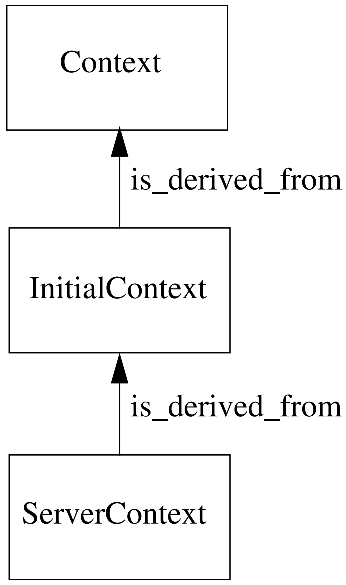
6.1.4 class Status:

The *Status* class is used to represent the outcome of naming operations. It contains information about how the operation completed and, if an error occurred, the nature of the error and some clues as to where it happened. The XFN specification defines the contents of a status object for many situations in order to provide enough information for a federating name system to guide resolution across naming system boundaries. For example, if a naming system is unable to fully resolve a composite name, the status object includes the most recently resolved reference, along with two composite names representing the resolved and unresolved portions of the name.

6.1.5 class Context:, class ContextHandle:

As mentioned previously, contexts are the major component of an XFN naming system and as such they require the bulk of the implementation effort. We decided to implement contexts using the Python class hierarchy shown in Figure 3 below. We also introduced the notion of context handles which are simple classes that provide clients with a way of referencing a context while allowing the context itself to be opaque.

Figure 3: The context class hierarchy



The *Context* class is an abstract class that defines the interface offered by XFN contexts. All concrete context classes must offer this interface (shown below) which provides the majority of the base context interface as specified by XFN.

```
class Context:
    “““ Abstract class for XFN contexts. “““

    def init__(self):
        “““ Create a new and empty context. “““

    def lookup(self, name, status):
        “““ Return the reference bound to ‘name’ relative to this context. “““

    def list_names(self, name, status):
        “““ List all atomic names bound in the target context ‘name’ relative to this context “““

    def list_bindings(self, name, status):
        “““ List all bindings in the target context ‘name’ relative to this context. “““

    def bind(self, name, ref, exclusive, status):
        “““ Bind the reference ‘ref’ to the composite name ‘name’ relative to this context. “““

    def unbind(self, name, status):
        “““ Remove the terminal atomic name in ‘name’ from the target context. “““

    def rename(self, oldname, newname, exclusive, status):
        “““ Bind the reference bound to ‘oldname’ relative to this context to ‘newname’. “““

    def create_subcontext(self, name, status):
        “““ Create a new context bound to ‘name’ relative to this context. “““

    def destroy_subcontext(self, name, status):
        “““ Destroy the sub-context named by ‘name’ relative to this context. “““
```

```
        def get_ref(self, status):
            """ Return a reference to this context. """
```

The interface for the ContextHandle class is as follows:-

```
    class ContextHandle:
        """ Context handles allow contexts to be opaque! """

        def __init__(self, ctx):
            """ Create a handle to the context 'ctx'. """

        def get_context(self, status):
            """ Return the context from the handle. """
```

6.1.6 class InitialContext(Context):

The *InitialContext* class is used to provide clients of the name service with a starting point for name resolution. When an object requests an initial context from the name service a new *InitialContext* instance is created and bindings are added to provide references to contexts that exist in any of the available name servers. XFN specifies policies for these initial bindings, including policies for global namespaces (such as DNS and X500) and enterprise namespaces (such as organisational units, hosts and users). It is important to note that *InitialContext* instances are created by the name *service* (see motivation), and that they are transient (that is, they only exist for the lifetime of the client object).

6.1.7 class ServerContext(InitialContext):

The *ServerContext* class provides the same semantics as the *InitialContext* class with one important addition; the state of a *ServerContext* instance is stored persistently on a shelf. This class is used in name servers to implement their persistent namespaces.

6.2 Implementing the Name Server

The *NameServer* class is simply a wrapper around the persistent *ServerContext* class. The wrapper implements the C-like XFN base context interface and offers it via Hector's communications service.

6.3 Implementing the Name Service

The *NameService* class is a little more complex as it is responsible for the resolution of composite names that span multiple name servers. To use the service, client objects must first request an initial context (see the *InitialContext* class) that provides a starting point for name resolution. The name service guides resolution from the initial context through (possibly) multiple servers to the target context by examining the status object returned by each intermediate server.

7 Future work

Our current system does not yet implement all of the optional facilities specified by XFN. In particular, we do not support the notion of XFN attributes. This extension provides a flexible way to associate arbitrary properties with object references without cluttering the basic naming model.

We also aim to provide support for one of the XFN protocols. This will be achieved via a C extension module and will allow our naming system to become part of the global federation.

8 Conclusion

Naming systems are all around us, whether or not we actually realise it. Many naming systems are buried deep inside applications making it impossible to provide arbitrary composite name support. The XFN specification gives us a general solution to a large class of naming problems and provides us with details of the interfaces and abstract data types required to implement a generic naming model. More importantly, XFN gives us a framework for the federation of naming systems enabling us to provide potentially global composite name support covering both new and legacy namespaces.

The majority of standards organisations produce work biased towards the more traditional development languages such as C. What our work has shown is that a ‘standard’ model can be implemented in an interpreted language such as Python with all of the associated development gains and without compromising interoperability with more traditionally developed systems. Python’s support for rapid prototyping is a major advantage as complex specifications such as XFN contain many hidden ‘intricacies’ that only tend to reveal themselves at implementation time!

References

[1] “CAE Specification - Federated Naming: The XFN Specification”
X/Open document number: C403 ISBN: 1-85912-052-0

[2] “Hector: Distributed Objects in Python”
David Arnold, CRC for Distributed Systems Technology