A Name Service for Evolving, Heterogeneous Systems[1]

Michael F. Schwartz,[2] John Zahorjan, and David Notkin

Department of Computer Science
University of Washington
Seattle, Washington 98195

Abstract

This paper describes a name service designed for long term use in a continually evolving heterogeneous system. There are two conflicting goals for a name service in this environment: to ease the task of dealing with the distribution and heterogeneity by providing a uniform service throughout the system that masks these characteristics as much as possible, and to keep software development and maintenance costs manageable when faced with the frequent introduction of new system types. A single name service implemented across all system types, which would be an appropriate choice given the first goal alone, is infeasible when the second goal is considered.

In attempting to satisfy these conflicting goals we have designed a software structure that allows the efficient integration of existing heterogeneous implementations of the same generic service. In the specific case of the name service, this allows us to build a global service that makes use of, rather than replaces, the name services of the component subsystems. This approach has a number of desirable properties. For one, the system is scalable, since the processing load is naturally distributed among the subsystems. Second, applications existing in newly introduced subsystems can continue to run unaltered, while the modifications they make in their local name services are automatically reflected in the global name service. This is an important attribute in environments where it is impossible or too expensive to modify all existing application and system code. Finally, our design allows wide latitude in the degree to which an individual subsystem type is incorporated; an amount of integration effort appropriate to the benefits received can be chosen individually for each subsystem type as it is introduced.

A prototype implementation has been built as part of the Heterogeneous Computer Systems project at the University of Washington. This service supports RPC binding and other applications in our heterogeneous environment. Measurements of the performance of this prototype show that it is close to that of the underlying name services, due largely to the use of specialized caching techniques.

1. Introduction

A name service provides the convenience of a runtime mapping from string names to data. The most important current use of such facilities is to determine address information, for example, mapping a host name to an IP address. By performing this address lookup at runtime, the name service provides a level of indirection that is crucial to the efficient management of distributed systems. Without it, changes to the network or system topology would require the recompilation of applications with hardwired addresses, and so would severely limit the size of distributed systems that could be realized in practice.

These name to data mappings are usually encapsulated in a logically centralized service (e.g., BIND [Terry et al. 1984] or Clearinghouse [Oppen & Dalal 1983]) designed especially for this purpose, because managing the data involves specific tradeoffs between efficiency and sophistication of support, tradeoffs that typically make inappropriate other potential mechanisms for the name to data mapping, such as distributed databases and shared file systems. The design described here is presented in terms of a logically centralized implementation. However, it is equally valid for other approaches to naming, such as broadcast-based location protocols [Almes et al. 1985, Cheriton & Mann 1984, Welch & Ousterhout 1986].

The goal of the service discussed in this paper is to provide a name to data mapping facility for an *evolving heterogeneous* system. We want our design to be scalable in the heterogeneous dimension, meaning that it may be applied to environments consisting of a large and increasing number of different system types

[1] This material is based upon work supported in part by the National Science Foundation under Grants DCR-8352098, DCR-8420945, and CCR-8611390, by an IBM Faculty Development Award, by a GTE fellowship, by the Xerox Corporation University Grants Program, and by the Digital Equipment Corporation External Research Program.

[2] Author's current address: Department of Computer Science, University of Colorado, Boulder, Colorado 80309.

but only a few instances of many of these types. We are willing to incur the runtime penalty of an extra level of indirection to achieve this flexibility. In contrast to other efforts to provide a more standardized service on more controlled heterogeneous bases (such as Project Athena [Balkovich, Lerman & Parmelee 1985] and the ITC file system [Satyanarayanan et al. 1985]), the critical consideration in our environment is the cost of integrating new system types into our network service. Thus, solutions based on porting services to each system type are infeasible, and we must instead *accommodate the heterogeneous services* that may already exist on each system type.

In developing our name service for this environment we have utilized a software structure that combines much of the benefit of the standardization approach with the desired ease of new system integration. This software structure appears to be applicable to other application domains, such as filing and mailing. As applied to naming, this software structure has three key characteristics.

- First, our network-wide name service makes use of, rather than replaces, name services and associated data already existing in the individual system components. We call this the *direct access* approach, as distinguished from *reregistration-based* approaches that require transfer of responsibility from existing services to a new network-wide service. The major advantage of the direct access approach is that it allows the underlying subsystems to evolve independently of the global name service, while still reflecting this evolutionary change to the clients of the global name service. Clients that use their own name service, rather than the global name service, are not modified to accommodate this arrangement, but they do not derive any of the benefits of this arrangement either.
- Second, we recognize that a key difficulty of heterogeneous naming is the variety of data semantics and access protocols involved in using the information. Based on this observation, our system separates the management of the global name space from the understanding of the semantics and access protocols of the data in a fashion that makes adding new naming systems and applications as easy as possible.
- Third, because our approach introduces a level of indirection, we use a specialized caching scheme based on locality of reference to *query class* and *name system type* to provide acceptable performance.

We have constructed a prototype name service having these characteristics. The prototype is in use as part of the Heterogeneous Computer Systems (HCS) project at the University of Washington [Black et al. 1985, Black et al. 1987]. The goal of this project is to provide for loose integration through network services, meaning that a set of core services (filing, mail, and remote computation) are provided network-wide, but no attempt is made to mask the heterogeneous aspects of the various systems since this heterogeneity was presumably the motivation for the systems' acquisition. These network services are easily extended to new system types because they are built upon two underlying facilities explicitly designed to accommodate heterogeneity. The first, heterogeneous RPC [Bershad et al. 1987], is based on *emulation*: the heterogeneous RPC (HRPC) mechanism looks to each existing RPC mechanism exactly the same as a homogeneous peer. The other facility, our name service, is based on the notion of direct access, allowing network-wide manipulation of the existing heterogeneous name services through a homogeneous interface.

In the remainder of this paper we explain the software structure used to implement our name service and motivate the choices made in its design. Section 2 describes the model of our name service. Section 3 describes experience with a prototype implementation. Section 4 compares this work with several related efforts. Section 5 offers some conclusions.

2. The HNS Model

In this section we describe in detail our design of the HCS Name Service (HNS). In doing so, we indicate not only what decisions we have made, but also *why* we made those particular choices.

Direct Access Naming

The single characteristic that most distinguishes the HNS from other name services is that of *direct access*, the direct use of existing name services in managing the data available through our global service. We cannot afford to replace the existing name services of the subsystems because that would requir either modifying existing applications to use the new service or else the periodic reregistration of data from the local name services to the global standard. We rejected the former because of the sheer magnitude of the programming work involved when there are many different system types in the environment. The latter is inappropriate because of problems with name conflicts and consistency of information on the global and local levels, because the reregistration cost is one that continues without end, because the degree of system heterogeneity would be limited by the rate at which the global name service could absorb the reregistrations, and because to be at all scalable the considerable code of this global service would have to be implemented on many different system types to allow for distribution.

The HNS Name Space

The HNS name syntax is designed to avoid naming conflicts and to support mapping from HNS to local name service names, as is required for our direct access scheme. HNS names contain two parts, a *context* and an *individual name*. Roughly, the context identifies the local name service in which the data can be found while the individual name determines the name of the object in that local service. The individual name can be any string, but in the simplest case is identical to the name of the entity in its local name service. This correspondence between the local name and individual name makes it relatively easy for a user to communicate the global name of a local resource to a user on a remote system because most of the global name is already familiar. On the other hand, this scheme creates a global name space that does not conform to any simple syntax rules. This problem can be overcome by enforcing more complex mappings that provide a uniform, global syntax. However, we do not feel that this is necessary since most often the user of the name of a remote resource has been given that name by someone else (who is probably local to that resource), and so the remote user need do no more than repeat the string to make use of it.

The context portion of an HNS name maps onto all or part of the name space managed by a single local name service. This has two beneficial effects. First, since all names within a context must be the responsibility of a single local name service, finding the data associated with an HNS name is simplified. The alternative of locating the appropriate local name server, either through some multicast technique [Cheriton & Mann 1984] or some form of search path, is either too inefficient in our environment, has the flavor of relative name spaces (something we wished to avoid), or requires excessive development cost to attain the needed level of homogeneity. Second, by imposing the additional restriction that the mapping from local names to the individual name portion of HNS names be a function (i.e., produce a unique result), we guarantee that no naming conflicts can ever be created in the HNS name space when combining previously separate systems. Any scheme allowing a context to contain names from more than one local name service either must allow the possibility of name conflicts or must require that the local name services coordinate during name creation. Our approach avoids both these unacceptable effects, instead allowing existing applications to use native name service operations to create new names in the HNS name space, thus providing information to new applications written to use the HNS.

Managing Heterogeneous Naming Semantics

To make the global name space useful, we want to relieve individual applications from the chore of handling heterogeneity. A client would like to present a name and obtain data without regard to the specific underlying name service that happens to be used in the name resolution. Clearly, there must be some code, specific to the particular name service and *query class* (i.e., the type of data to be returned), that can locate the data and convert data formats. The major decision to be made is where this system and application specific knowledge should reside. For instance, code to handle the heterogeneity could be part of the client or part of the HNS itself. Our general goals preclude the former, since that approach requires the modification of large numbers of existing applications each time a new system type is introduced. The potential for sharing, then, suggests putting this function in the HNS rather than in individual applications. The problem with this placement is that the continual introduction of new query classes would require repeatedly changing the HNS, an unacceptably expensive and unwieldy evolutionary process.

For these reasons, instead of placing the system and query class specific code in the applications or in the HNS, we handle naming semantics in remote procedures called *Naming Semantics Managers (NSMs)*. Each NSM understands the semantics of naming for a particular query class and a particular name service. In this way, adding a new system type simply requires building NSMs for those queries to be supported and registering their existence with the HNS. All NSMs for a particular query class have identical client interfaces. Thus, when an application makes a query, it can call whichever NSM handles that query class for the specified context without having to know which name service will ultimately provide the response. The HNS provides the glue for this confederation by keeping track of the existence and location of all name services, contexts, and NSMs. A simplified view is that the HNS directly supports only the context/query class → NSM mapping, while the NSMs do all the "real" work, work that cannot be avoided no matter what software structure is employed. The purpose of the HNS is to improve the manageability of the NSM code. The NSMs are neither HNS nor application code *per se*. Rather, they are code managed by the HNS and shared by the applications.

Client Perspective

In its simplest form, a client calls the HNS using heterogeneous RPC (HRPC), passing the HNS name and query class. Based on the context portion of the name and the query class, the HNS returns an HRPC Binding (a handle to a remote procedure) that allows the client to call the appropriate NSM. The client then calls the NSM using the query specific interface, which includes the original HNS name. The NSM translates the individual name portion of that name to the corresponding local name, interrogates the local name service using this name, and returns the results in a format that is standard for that query class. This scenario is illustrated in Figure 2.1.

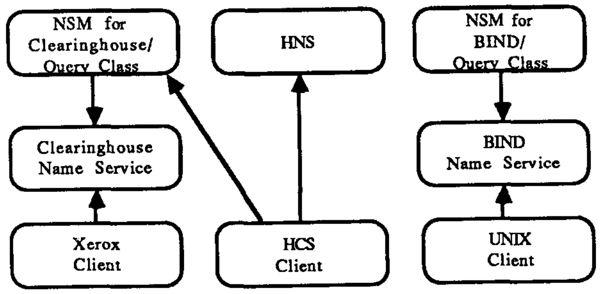
NSM for Clearinghouse/ Query Class
HNS
NSM for BIND/ Query Class
Clearinghouse Name Service
BIND Name Service
Xerox Client
HCS Client
UNIX Client

Figure 2.1: HNS Query Processing

In this figure, the requested name exists in the Clearinghouse, so the client is given a handle to call that NSM. A subsequent call might be for a name in BIND, in which case the client would call the BIND NSM. Since the interfaces provided by both NSMs are identical, the client does not need to be aware of which name service it is calling.

An alternate approach would be to have the HNS call the NSM on behalf of the client. Since each query class requires its own interface, this would require that the HNS be recompiled each time a query class is added. We could define generic interfaces for the clients to call the HNS and then the HNS to call the NSMs, but doing so would require encoding transmitted data into self-describing packages (as in Eden [Almes et al. 1985], for instance), an approach not supported by our HRPC model.

Scalability

In terms of accommodating the sheer size of the system, say as measured by the total number of names it contains, our design for the HNS shares with most other name service designs the property of being distributable. The basic distribution of the HNS occurs naturally since each new system type introducing a new set of names also includes a name service managing those names that we can take advantage of directly. In terms of accommodating a large number of heterogeneous system types, users of the HNS are confronted with the unavoidable work of providing NSMs for each query class and system type combination to be supported. The users of the system can decide independently which particular NSMs are worth the effort to construct, and so can match implementation effort to benefit.

Summary

The HNS differs significantly from other name services because of the requirements of our heterogeneous environment.

The HNS must use data in existing name services because reregistering names into the HNS is unmanageable in our continually evolving environment. However, to insulate clients from the complexities of distribution and heterogeneity, the HNS provides a single name space. This is difficult since the underlying name services have differing syntax and semantics. We encapsulate these differences by providing, for each query class and native name service, an NSM that manages the syntactic and semantic details.

To perform an HNS query, the client presents an HNS name and query class. From the query class and the context portion of the name, the HNS selects the NSM that can access the appropriate name service for the client. The client then calls the designated NSM, which queries the underlying name service and returns a standardized form of the result. Since each NSM for a given query class has an identical interface, the client can call the NSM that the HNS designates without regard to the name service that NSM uses.

3. Experience

Environment

The current HCS environment consists of a heterogeneous collection of hardware (Suns, VAXen, Xerox D-machines, IBM RTs, and Tektronix 4400-series machines), communication mechanisms (Sun RPC, Courier RPC, TCP/IP message passing, and UDP/IP message passing), and operating systems (XDE, UNIX,[3] and Uniflex). We have built a prototype heterogeneous RPC facility [Bershad et al. 1987] on top of a subset of these systems, capable of communicating with each of the other systems by emulating Sun RPC, Courier RPC, and TCP or UDP message-based communication using a single RPC-style interface. We have also built a prototype HNS and a set of key network services – filing, mail, and remote computation – based upon HRPC and HNS.

The prototype HNS currently provides integrated naming with two widely available underlying name services: BIND [Terry et al. 1984], which operates in conjunction with the UNIX component of our prototype environment, and the Clearinghouse [Oppen & Dalal 1983], which operates in conjunction with the Xerox component. We plan to introduce additional name services as they become available to us.

Implementation

Although all data associated with individually nameable entities is kept in the underlying name services, the HNS maintains additional *meta-naming* information needed for managing the global name space. This information consists of the names and binding information for each name service and each NSM, the names of all contexts, and the mappings from contexts to name services.

While the HNS is logically a single, centralized facility, its implementation must be distributed and replicated for the usual reasons of performance, availability, and scalability. Because the implementation problems associated with these properties are for the most part successfully addressed in previous name services, we chose to ease our implementation effort by making use of an existing name service to store the meta-naming information. In particular, we use a version of BIND, modified to support both dynamic updates and also data of unspecified type [Schwartz 1987]. The HNS itself is a collection of library routines that access this version of BIND.[4] We have also built a collection of NSMs for our initial set of applications.

The primary HNS function is the call to locate an NSM, FindNSM. This call maps a context and query class to the information, called an HRPC *Binding*, needed for making an HRPC call to the NSM. FindNSM is implemented as the following sequence of mappings:

1. Context $\rightarrow$ Name Service Name
2. Name Service Name, Query Class $\rightarrow$ NSM Name
3. NSM Name $\rightarrow$ HRPC Binding for the NSM

Mappings 1 and 2 are each BIND lookups. The NSM binding information stored in the HNS contains, among other information, the host name on which the NSM resides; hence, mapping 3 involves translating the host name to a network address. This in itself is an HNS naming operation requiring a call to FindNSM. Thus, this mapping actually requires two more mappings (i.e., mappings 1 and 2, to find the NSM that can perform the host address lookup). Further recursion is avoided by linking instances of the NSMs that perform this mapping directly with the HNS, so that their network addresses need not be found.

While we recognize that the lookups made by FindNSM could be collapsed into fewer calls (e.g., by mapping the Context and Query Class directly to the Binding for the NSM), we chose to keep these mappings separate, because this allows more flexibility and requires less redundant information. For example, if more than one context is stored on the same name service, the binding information for that name service need only be stored once. Further, we realized that caching would greatly reduce the cost of these mappings, and so decided to adopt them for the flexibility they afford. (The effects of caching on performance are described below.)

An HNS Application: HRPC Binding

HRPC binding, the process of connecting clients with servers, was the first application of the HNS, because it presented the most immediate need for the HCS project as a whole. It is also a good test for the HNS because of the difficulties of binding in a heterogeneous environment. In particular, the information needed for binding is stored in different places depending on system type. Worse yet, each system type typically has its own binding proto-

[3] UNIX is a trademark of AT&T Bell Laboratories.

[4] Note that the version of BIND used to implement the HNS is separate from the conventional version of BIND. The former serves only as a simple repository for the HNS meta-information, while the latter holds actual naming data.

col. Thus, binding presents a practical "stress test" for our design.

The HRPC design involves the careful specification of clean interfaces between the five principal components of an RPC facility: the *stubs*, which are interposed between the client (also the server) and the run-time support; the *binding protocol*, which allows a client to locate a particular server; the *data representation*, which determines how data values are marshalled; the *transport protocol*, which determines how data is carried from one host to another; and the *control protocol*, used internally by the RPC facility to track the state of a call. An RPC client (or server) and its associated stub can view each of the remaining four components as a "black box". These black boxes can be "mixed and matched" to emulate different communication protocols at call-time. The set of protocols to be used is determined dynamically at bind-time – long after the client (or server) has been written, the stub has been generated, and the two have been linked.

In homogeneous systems, the choice of RPC components is fixed at *implementation time;* at run-time, the code simply begins executing as a monolithic unit once the binding process is complete. With HRPC, these components have been separated from each other and made dynamically selectable, and hence binding must perform the additional processing needed for component selection. Also, insular clients/servers have established binding protocols that they execute, and they expect their peers to execute the corresponding parts of the protocol. While the binding process is similar for most RPC systems, the actual mechanisms employed for naming, server activation, and port determination vary considerably. Hence, HRPC binding must proceed in a manner that can emulate the binding protocols for each of the systems being accommodated.

Using the HNS, the client's view of binding is fairly straightforward. The client presents a name and is returned a Binding to an NSM that understands exactly how to do binding on the system type from which the name came. The client then calls this NSM, which returns an HRPC Binding to the server of interest. This Binding is system-independent from the point of view of the client, even though the means by which this information is gathered by the NSM varies widely from system to system. A major advantage of this mechanism is that adding a system with a different RPC binding protocol only requires implementing a new binding NSM and registering its presence with the HNS.

As an example of the use of the HNS, suppose a client of the HRPC system issues the import call:

```
Import(ServiceName: "DesiredService",              (*in*)
       HostName: "BIND,fiji.cs.washington.edu",    (*in*)
       ResultBinding: DesiredBinding)              (*out*)
```

`Import` now acts as a client of the HNS to obtain a binding to "`DesiredService`" to return to its caller. `Import` uses the `HostName` specified by the client to construct the HNS name context ("`HRPCBinding-BIND`") needed to look up the binding information in the HNS. A call with query class "`HRPCBinding`" is then made to the HNS to obtain a Binding to the appropriate NSM:

```
FindNSM(BindingToHNS: HNSBinding,                   (*used by HRPC*)
        QueryClass: "HRPCBinding",                  (*in*)
        HNSName:
          (Context="HRPCBinding-BIND",
           Name="fiji.cs.washington.edu"),          (*in*)
        NSMBinding: TheNSMBinding)                  (*out*)
```

The `Import` code then uses the `NSMBinding` to call the NSM, passing it the `HNSName`:

```
BindingNSM(BindingToNSM: TheNSMBinding,          (*used by HRPC*)
           ServiceName: ServiceName,             (*in, from
                                                   Import call*)
           HNSName:
             (Context="HRPCBinding-BIND",
              Name="fiji.cs.washington.edu"),      (*in*)
           ClientBinding: ResultBinding)           (*out*)
```

The NSM looks up the local name ("`fiji.cs.washington.edu`") in the name service, and then determines the needed port number for the `ServiceName`, using whatever binding protocol is appropriate for that particular system. The completed Binding to the service is then returned to the `Import` code, which returns the information to the client.

The binding NSMs for both the BIND and Clearinghouse subsystems are about 230 lines each. About three weeks were spent adding and modifying the code required to implement HNS-based binding. The majority of this time consisted of implementing and measuring alternative approaches.

Performance

Every HNS naming request requires two steps in addition to the effort expended by the underlying name service: determining which NSM should handle the query, and calling that NSM. These two steps are the basic overhead of HNS naming.

Our initial implementation of `FindNSM` required elapsed times of 460 msec. per call. This poor performance, which was expected, was due to the cost of the many BIND lookups needed to access the meta-naming information. By installing a cache, we were able to reduce this cost to 88 msec. The remote call to the NSM takes 22-38 msec., depending on the RPC system used. The remote call is avoided when the needed information is cached.

In total, the basic overhead of HNS naming is between 88 and 126 msec. By way of comparison, a BIND name to address lookup takes 27 msec., and a Clearinghouse name to address lookup takes 156 msec.[5]

The above figures indicate the basic overhead inherent in HNS-based naming. As an example of the costs of HNS-based naming in an application, we found that HRPC binding, including overhead, requires between 104 and 547 msec., depending on where the HNS and NSMs are located and how caching is done (described below). To give these numbers some significance, it is worthwhile to make comparisons with alternative binding mechan-

[5] Clearinghouse accesses are slow because each access is authenticated, and virtually all data is retrieved from disk [Oppen & Dalal 1983]. In contrast, BIND does no authentication and keeps all its information in primary memory.

isms. The interim HRPC binding mechanism, used prior to the construction of the HNS prototype, was based on information reregistered in replicated local files. Binding using this scheme took 200 msec. We should also compare our HNS-based binding timings with a scheme in which a name service holds all of the (reregistered) data. We implemented such a scheme on top of the Clearinghouse, and found that binding took 166 msec.

While it may be possible to improve the performance of such a scheme (e.g., by using BIND instead of the Clearinghouse to store the data), this comparison shows that the tuned HNS performance is reasonably close to that of homogeneous name services.

Caching And Colocation

There are two ways to improve performance of the HNS: one can use caching to reduce the number of calls made by the HNS and NSMs to access their data, or one can link the HNS and NSMs with the client so that local procedure calls can be used between them. Because the HNS accesses its data from other servers (BIND for the meta-naming information, and the underlying name services for application data), even the HNS can be linked locally. Similarly, the NSMs can be linked with any process.

The freedom to link the HNS and NSMs with any process, rather than embodying them in a particular set of servers, provides several possible designs for any particular HNS client. We call the choice of where the HNS and NSMs are linked for each client the *colocation arrangement*. This flexibility allows a tradeoff between performance and ease of management. On one hand, locally linked NSMs are harder to manage than remote NSMs, since adding a new one or modifying an existing one requires relinking all affected clients.[6] On the other hand, where the NSMs and HNS are linked affects performance, since local calls are cheaper than remote ones.

The performance tradeoffs involved in the colocation arrangement are actually more complicated than just described because of caching. The reduction in call overhead realized by linking procedures locally is at odds with the fact that caching is more likely to be effective in long-lived remote servers than in locally linked copies. A natural question to ask is how big a cache hit improvement in the remote location is required to compensate for the increased calling cost. Letting $C(\cdot)$ mean "cost of" and using p to represent the cache hit fraction with locally linked copies and q the increase in the cache hit fraction obtained from remote location, we have

$$C(\textit{remote location}) = C(\textit{remote call}) + (p+q)C(\textit{cache hit}) + (1-p-q)C(\textit{cache miss})$$

$$C(\textit{local location}) = C(\textit{local call}) + (p)C(\textit{cache hit}) + (1-p)C(\textit{cache miss})$$

Since $C(\textit{local call})$ is effectively zero in the time scale of the other terms, remote location is preferable whenever

$$q > \frac{C(\textit{remote call})}{C(\textit{cache miss}) - C(\textit{cache hit})} \qquad (1)$$

We will make use of this relationship shortly, after presenting measurements providing the cost measures needed to apply it.

As a first attempt to characterize these tradeoffs, we ran a series of experiments to determine how the various colocation arrangements and caching strategies affect performance. Although these experiments focus on HRPC binding, the results should apply to other applications as well. Our timings were made between two MicroVAX-II's at light load, with the BIND server used by the HNS and the public BIND server each residing on lightly loaded MicroVAX-II's. All machines were joined by an Ethernet.

Table 3.1 shows the measured performance for the case of HRPC import of Sun RPC servers. The two dimensions of the table are colocation arrangement and the results of cache lookups. (In the experiment for row 2 a single process remote from the client acted as the client's agent, making local calls to the HNS and then to the NSM. This structure provides a mixture of colocation efficiency and ease of NSM update, as the code to be modified with changes to the NSM is well contained.)

Column A illustrates the effect of the colocation arrangement for the case of no cache hits. (Since the overhead required to determine that a reference is a miss is about 0.1% of the total times in column A, these times can also be interpreted as those required when no caching is implemented.) Each row represents a different choice of colocation. The configuration of row 1 requires no remote calls among the client, HNS, and NSM, those of rows 2-4 each require one call, and that row 5 requires two calls. In all cases the client resided on separate hosts from the HNS/NSMs whenever they were not directly linked together. (Locating them on the same host reduces the timings by about 20 msec. in applicable configurations.) As can be seen in this column, the colocation of the client, HNS, and NSMs can have only a modest influence on the total cost of an HNS query. The reason for this is the large fixed cost associated with the many remote accesses the HNS must perform to determine a handle for an NSM. Thus, reducing the number of remote calls by one or two has only a marginal effect. (Although colocation with the BIND service used by the HNS can have a fairly large effect, this is not a generally applicable approach and so is not considered further.)

[6] In systems that support shared libraries (e.g., Multics [Daley & Dennis 1967]) this is less of an issue because multiple clients can be extended by updating a single shared library. However, to our knowledge no systems support shared libraries across machine boundaries, and hence updating locally linked clients would still require updates to each machine in turn. In contrast, since NSMs are called using HRPC, registering an NSM with the HNS extends the functionality of all machines at once.

Colocation Arrangement	A. Cache Miss	B. HNS Cache Hit	C. HNS and NSM Cache Hit
1. [Client, HNS, NSMs]	460	180	104
2. [Client] [HNS, NSMs]	517	235	137
3. [HNS] [Client, NSMs]	515	232	140
4. [NSMs] [Client, HNS]	509	225	147
5. [Client] [HNS] [NSMs]	547	261	181

Table 3.1: Performance of HRPC Binding for Various Colocation Arrangements (msec.)
[ ] indicates colocation.

While it is possible to eliminate some of the indirection used by the HNS, and so reduce the number of remote calls, this would also decrease the resilence of the HNS to reconfigurations of the distributed system. Instead, we improve the speed of each lookup through caching. Both the HNS and the NSMs were modified to cache the results of remote lookups. Column B of Table 3.1 shows the performance observed when the HNS has a cache hit but the NSM has a cache miss, while column C shows the performance when both phases have hits.[7] It is clear from the table that caching results in a significant performance improvement over the base case. Further, most of this improvement is attributable to the HNS. This is not too surprising since the basic HNS scheme requires six data mappings, each of which involves a remote call in the case of a cache miss, while the NSM needs only a single remote call.

Based on the observation that the HNS cache is the most important determinant of performance, we experimented with the idea of preloading that cache. (We also considered preloading the NSM caches, but that would be less effective). The motivation for this is simple. In those cases where the HNS used by the client is a local copy, the cost of the many remote lookups required on the initial reference to various pieces of meta-naming information might exceed the cost of preloading the relatively small amount of information (currently about 2KB) required to guarantee HNS cache hits. The actual preload cost was measured to be about 390 msec.[8] Since the cost of preloading plus a cache hit falls between one and two cache miss times, preloading seems to be effective in situations where two or more calls to the HNS for different context/query classes will be made.

The major lesson to draw from the measurements of Table 3.1 is that the potential benefit of caching far exceeds that obtainable solely by colocation. The reason for this is clear: at most two remote calls can be eliminated by colocation while each cache hit eliminates many.

[7] Cached data is tagged with a time-to-live field for cache invalidation. While this simplistic mechanism can cause cache consistency problems, it would not make sense to use a more sophisticated scheme because the source of our cached data (BIND) also uses this mechanism for cache invalidation. Given our assumption that data changes slowly over time, we feel that this mechanism will suffice.

[8] The BIND zone transfer mechanism, used by BIND secondary servers to request data transfers from primary servers, was employed to preload the caches.

Finally, there is still the question of whether colocation of the HNS or NSMs with the client is worthwhile. Beginning with the base case of remote HNS and NSMs, consider the effect on performance of making the HNS local. Using equation (1), and estimating $C(\textit{remote call})$ as 33 msec., $C(\textit{cache hit})$ as 261 msec., and $C(\textit{cache miss})$ as 547 msec., we calculate that the cache hit fraction obtained when the HNS is remote must exceed that when it is local by an additional 11% for the remote case to provide better performance. Now consider also making the NSMs local. Applying equation (1), and estimating $C(\textit{cache hit})$ as 147 msec. and $C(\textit{cache miss})$ as 225 msec., an additional 42% cache hit must be experienced by the remote NSMs for them to be preferable to local copies. Neither of these increments leads to a clear cut decision about the most efficient location for the HNS or the NSMs. Further work on the dynamic cache hit ratios achieved in practice will be required to make this decision for any particular workload. It is important to remember, however, that from a software maintenance point of view there is a high price for colocation in terms of the difficulty of maintaining the software in the face of changes to the HNS and NSMs.

Although unrelated to the specific purposes of the HNS, we have had some interesting experience with the cache we implemented. In the initial version, we kept data in its marshalled form, and demarshalled it upon every access, expecting that marshalling was a minor expense. To our surprise, the cost of marshalling was very high: the time taken to perform BIND lookups depended heavily on the number of BIND *resource records* returned.[9] As shown in Table 3.2, by simply changing the cache to keep demarshalled information, the times decreased dramatically.

This result is surprising, especially in light of the fact that the HCS file system found that marshalling/demarshalling accounted for only 1% of its call time [Black et al. 1987]. Upon further investigation, we determined that the marshalling routines we used for BIND were significantly more expensive than those used by the HCS file system. This complexity was the price we paid for the RPC-style structure we built for our BIND interface: rather than use the standard BIND library routines (which include the code to marshal, send/receive, and interpret BIND client-server messages), we built an HRPC interface to BIND. This interface is built on top

[9] BIND data is stored as a collection of *resource records*, each of which can be up to 256 bytes of data. Separate resource records are intended to store alternate data for one name, e.g., multiple network addresses for gateway hosts.

Resource Records Per Name	Cache miss	Marshalled cache hit	Demarshalled cache hit
1	20.23	11.11	0.83
6	32.34	26.17	1.22

Table 3.2: The Effect of Marshalling Costs on Cache Access Speed (msec.)

of our *Raw* HRPC protocol suite [Bershad et al. 1987], which allows HRPC clients to make calls to any message passing program that conforms with the basic RPC paradigm of "make a request and wait for a response". Instead of writing complicated low-level marshalling routines to handle the BIND message format, we described this format using our interface description language, and used the marshalling code generated by our stub complier.

The problem is that the generated marshalling routines, although correct, incur a good deal of overhead in procedure calls, indirect calls to marshalling routines, unnecessary dynamic memory allocation, and unnecessary levels of marshalling. In particular, the standard BIND marshalling routines corresponding to the cases measured in Table 3.2 take .65 msec. and 2.6 msec. for one and six resource record lookups, respectively. While this experience shows that our HRPC-based marshalling scheme is quite a bit more expensive than necessary, it also shows that we were able to retain the advantages of this scheme at reasonable performance by making a simple change to our cache implementation.

4. Related Work

To recap, the goal of our design is to allow the integration of existing name services to form a uniform, global facility. It is important in our environment that existing applications be allowed to run unaltered while at the same time reflecting the naming updates made by them to the network-wide clients of our name service. In this section we contrast our work with various earlier efforts in both heterogeneity and naming.

Internet Mail Systems

Most early work in heterogeneous naming concerned internet mail systems [Redell & White 1983], the most well-known example being UNIX sendmail [Allman 1985]. Sendmail uses rewriting rules to describe how to parse heterogeneous mail names. Although sendmail has allowed the interconnection of a large variety of electronic mail networks, this technique has several drawbacks. First, sendmail centralizes the understanding of mail naming in a single component (which is replicated on each host); the proliferation of interconnected networks [Quarterman & Hoskins 1986] makes this approach difficult to manage. Second, sendmail depends on being able to discern naming semantics based on the syntactic structure of names. Doing so impedes name space administration [Terry 1985, Terry 1986] and reflects the complexity of heterogeneous naming to clients and users of sendmail-based mail agents.

CCITT has undertaken an effort to standardize mail naming and protocols for a world-wide electronic mail network [CCITT 1984, Cunningham 1983]. While standardization would clearly be very beneficial in the long term, it does not provide a solution in the short term: as made evident by the ongoing transition to Domain-style naming [Postel 1984], renaming on such a grand scale is expensive and difficult. In addition, approaches based on standardization require a large effort, which would not be effectively amortized in our environment, since we have only a few instances of many different system types.

DEC SRC Global Name Service

There has been recent work at the DEC Systems Research Center concerned with constructing a replicated name service intended to scale to the level of billions of names distributed throughout the world [Lampson 1986]. The major contribution of this work is a scheme for combining name services by allowing the root to be extended arbitrarily (in contrast to systems such as the Domain Internet naming scheme, which only grows downward from a fixed root). The major difficulty encountered is dealing with changed names: extending the root causes the absolute names of some (or all) entities to change; such global renaming is made feasible by the provision of mechanisms that smooth the transition. One such mechanism involves leaving temporary forwarding links. A second mechanism involves allowing applications to "change roots", the idea being that only applications that actually need to access names in the newly extended root need change their root to the global root. A third mechanism involves keeping a list of important names that were reachable from previous roots in the "superroot". This scheme is intended for interim use only, as otherwise the superroot processing and storage requirements would not scale to the desired level.

Although this work, like the HNS, is concerned with combining name services, the focus and characteristics of the systems are quite different. The DEC SRC system is primarily concerned with scalability in the *size* dimension, where as the HNS work is more concerned with scalability in the *heterogeneity* dimension. The HNS assumes a single globally rooted context name space, concentrating on allowing the individual names to vary in syntax and semantics. The DEC SRC system allows its name space to grow upwards, concentrating on ways to allow this freedom without the troubles typical of renaming and relative naming.

Decentralized Name Interpretation

Cheriton and Mann [Cheriton & Mann 1984, Cheriton & Mann 1986] have developed a facility for global naming with no central authority. The primary idea is that names are interpreted by the services that provide named entities, rather than by a logically centralized name service. The point of this method is that it saves accessing a second party service, gaining efficiency and robustness.

The HNS name space may also be classified as being decentrally interpreted. However, there are more differences than similarities in our work. In V, the emphasis is on decentralized interpretation as a *new* naming scheme, oriented towards increasing efficiency and reliability in naming. In the HNS, the emphasis is on accommodating multiple *existing* naming facilities. Direct access naming is based, in part, on decentralized interpretation, but decentralized interpretation does not necessarily imply direct access.

Portal Based Naming

Lantz et al. designed and implemented a naming system based on entities they call *portals* [Lantz, Edighoffer & Hitson 1985]. A portal is an active entity associated with an action to be taken when an entity is referenced. This introduces a level of indirection in name interpretation, and supports monitoring, access control, and "domain switching", i.e., stripping off part of the name and passing the rest on to a new "domain" to continue its interpretation. This latter aspect could help support integration of heterogeneous name services.

While they bear some resemblance to NSMs, portals are intended to support a broader variety of functions. NSMs are intended primarily for dealing with heterogeneous naming semantics, and we have focused on the issues relevant to this purpose. In addition, there is no indication of how portals should be accessed or managed. The HNS provides the support needed to manage NSMs, separating the issues of understanding semantics from name space administration.

Administrative Autonomy

Peterson defines a notion of heterogeneity concerning an internet that consists of autonomous organizations [Peterson 1985]. The key problem is the lack of a single system-wide user creation operation that assigns a high-level name to a user at creation time. His main concern is allowing users to name each other without forcing users to register explicitly with a name service. Like the HNS, his mechanism integrates autonomous name spaces. Unlike the HNS, Peterson's system provides a collection of tools that support a bottom-up construction of the naming network, continuously combining name spaces, rather than joining all name spaces under a single global root. Names thus seem to be relative to the current root, and change if the current root changes. Peterson also allows a less restrictive naming syntax than usual: names are sets, where each element of the set is described by a regular expression.

Jasmine File System

Jasmine [Marzullo & Wiebe 1986] is a system consisting of workstation tools and network services to help programmers develop, release, and maintain large software systems. The Jasmine file system integrates heterogeneous file systems by using names that map, via syntactic transformations, to the names of files in the underlying file systems. The file system presents a Fetch/Store interface. To fetch or store a file, the system first consults a database to determine the file location and file system type. Based on the file system type, a call is made to a particular "plug-in" procedure to handle the operation [Wiebe 1987]. These plug-in procedures are similar to NSMs, in that they implement identical interfaces to different underlying systems, and new ones can be added by dynamically loading them.

There are several differences from our work. First, NSMs are potentially remote procedures. Hence, the method of adding new ones differs from Jasmine. It is easier to add HNS applications because NSM registration is done in one place, instead of on each host. On the other hand, the Jasmine procedures are more efficient than the most general HNS case, since they are always local procedures, with a less expensive selection protocol. Second, the HNS is a more general mechanism, since it allows arbitrary naming interfaces; Jasmine did not need to be this general. Third, Jasmine maintains location information for each file. This would be inappropriate in the HNS because it would make the location database comparable in size to the database of information to which names map. This is not a problem in a file system, since files are typically large relative to naming data.

Heterogeneous Databases

The database community has been working on integrating heterogeneous systems for several years. The goal is to allow users to read and manipulate data from several independently created/administered databases, each of which has different data formats, access protocols, and manipulation languages. The methods used for accessing these databases vary from multilevel translation (between query languages, data formats, etc.) [Templeton et al. 1986] to meta-query languages that allow the user to name various databases and define relationships between them, for manipulation, privacy, and equivalence dependencies [Litwin & Abdellatif 1986]. These schemes support joining of data in different database schemas, and broadcasting of user intentions over a number of database schemas with varying naming rules for data with similar meanings.

These projects differ from our work in several significant ways. First, their goals are often different: Some systems want to allow users to perceive "varying views of reality" [Litwin & Abdellatif 1986], whereas the HNS is intended for allowing a more coherent view of abstractly similar subsystems. Second, these approaches are not typically *factored* in such a fashion to allow easy introduction of new database types, whereas reducing the cost of integration is the primary goal of our work. Third, the implementation techniques differ significantly: the database work often involves language translation, whereas our scheme uses registered agents to handle particular access protocols and data semantics.

5. Conclusions

We have described a new approach to providing a name service for continually evolving systems that are composed of a heterogeneous collection of subsystems, with the overall goal of reducing the cost of integrating new system types into an existing environment. Rather than implementing a new global standard, our approach is based on integrating existing name spaces through a structure that separates name space administration from knowledge of the semantics of naming in each of the assimilated subsystems. A major advantage of this approach is ease of integration: newly added system types can participate in the larger system without modification, and systems that use the name service can take advantage of the services provided by new systems without modification.

Based on measurements of our prototype, we have shown that a specialized caching scheme based on locality of reference of query class and name system type can provide acceptable performance, that caching of meta-naming information potentially saves more time than the difference between local and remote calls, and that the set of colocation alternatives represents a spectrum of tradeoffs in performance for ease of management, from which programmers can choose what best suits each particular application. We are continuing our effort towards improving the performance of the HNS without decreasing its flexibility.

Another major contribution of our work is the software structure we have defined. Relieving clients from the complexities of distribution and heterogeneity through the use of a global intermediary service (e.g., the HNS) and a set of agents that access existing services (e.g., the NSMs) is a generally applicable structure. We are pursuing this structure in the context of both an electronic mail system and also a heterogeneous file system that mediates access to the set of local file systems present in the environment.

Acknowledgments

A major strength of our effort is that it arises within a larger context – the Heterogeneous Computer Systems (HCS) project – in which a number of investigators are working together on broad solutions to the various problems that arise in accommodating heterogeneity. HCS project participants in addition to the authors include Brian Bershad, Andrew Black, Fran Brunner, Dennis Ching, Sung Kwon Chung, Bjorn Freeman-Benson, Kimi Gosney, Edward Lazowska, Henry Levy, John Maloney, Cliff Neuman, Brian Pinkerton, Jan Sanislo, Mark Squillante, James Synge, and Doug Wiebe.

Special thanks to Ed Lazowska for many careful readings and helpful suggestions. Jan Sanislo deserves special mention for overseeing the implementation effort, and particularly for making HRPC work as well as it does.

References

[Allman 1985]
E. Allman. Sendmail - An Internetwork Mail Router. *UNIX Programmer's Manual, 4.2BSD*, 2C, Comput. Sci. Division, EECS, UCB, Berkeley, CA, June 1985.

[Almes et al. 1985]
G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe. The Eden System: A Technical Review. *IEEE Trans. Software Eng.*, SE-11(1), pp. 43-59, Jan. 1985.

[Balkovich, Lerman & Parmelee 1985]
E. Balkovich, S. Lerman and R. P. Parmelee. Computing in Higher Education: The Athena Experience. *Commun. ACM*, 28(11), pp. 1214-1224, Nov. 1985.

[Bershad et al. 1987]
B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Trans. Software Eng.*, SE-13(8), pp. 880-894, Aug. 1987.

[Black et al. 1985]
A. P. Black, E. D. Lazowska, H. M. Levy, D. Notkin, J. Sanislo and J. Zahorjan. An Approach to Accommodating Heterogeneity. Tech. Rep. 85-10-04, Comput. Sci. Dep., Univ. Washington, Seattle, WA, Oct. 1985.

[Black et al. 1987]
A. P. Black, E. D. Lazowska, H. M. Levy, D. Notkin, J. Sanislo and J. Zahorjan. Interconnecting Heterogeneous Computer Systems. Tech. Rep. 87-01-02, Comput. Sci. Dep., Univ. Washington, Seattle, WA, Jan. 1987. Submitted for publication.

[CCITT 1984]
CCITT. *Recommendation X.400, Message Handling Systems: System Model - Service Elements.* CCITT, Study Group 5/VII, Oct. 1984.

[Cheriton & Mann 1984]
D. R. Cheriton and T. P. Mann. Uniform Access to Distributed Name Interpretation in the V-System. *Proc. 4th Int. Conf. Distrib. Comput. Syst.*, pp. 290-297, May 1984.

[Cheriton & Mann 1986]
D. R. Cheriton and T. P. Mann. A Decentralized Naming Facility. To appear, *ACM Trans. Comput. Syst.*, 1986. Available as Tech. Rep. CSL-TR-293, Comput. Sci. Dep., Stanford Univ.

[Comer & Murtaugh 1986]
D. Comer and T. P. Murtaugh. The Tilde File Naming Scheme. *Proc. 6th Int. Conf. Distrib. Comput. Syst.*, pp. 509-514, May 1986.

[Cunningham 1983]
I. Cunningham. Message-Handling Systems and Protocols. *Proc. IEEE*, 71(12), pp. 1425-1430, Dec. 1983.

[Daley & Dennis 1967]
R. C. Daley and J. B. Dennis. Virtual Memory, Processes and Sharing in MULTICS. *Proc. 1st ACM Symp. Operating Syst. Prin.*, pp. 306-312, Oct. 1967.

[Lampson 1986]
B. W. Lampson. Designing a Global Name Service. *Proc. 5th ACM Symp. Principles Distr. Comput.*, pp. 1-10, Aug. 1986.

[Lantz, Edighoffer & Hitson 1985]
K. Lantz, J. Edighoffer and B. Hitson. Towards a Universal Directory Service. *Proc. 4th ACM Symp. Principles Distr. Comput.*, pp. 250-260, Aug. 1985. Reprinted in Operating Syst. Review 20(2).

[Litwin & Abdellatif 1986]
W. Litwin and A. Abdellatif. Multidatabase Interoperability. *IEEE Computer Magazine*, 19(12), pp. 10-18, Dec. 1986.

[Marzullo & Wiebe 1986]
K. Marzullo and D. Wiebe. Jasmine: A Software

System Modelling Facility. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pp. 121-130, Palo Alto, CA, Dec. 9-11, 1986. Appears as SIGPLAN Notices 12(1), Jan. 1987.

[Mockapetris 1983]
P. Mockapetris. Domain Names - Concepts and Facilities. RFC 882, USC Information Sci. Institute, Nov. 1983.

[Notkin et al. 1987]
D. Notkin, N. Hutchinson, J. Sanislo and M. Schwartz. Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity. *Commun. ACM*, 30(2), pp. 132-140, Feb. 1987.

[Oppen & Dalal 1983]
D. C. Oppen and Y. K. Dalal. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. *ACM Trans. Office Information Syst.*, 1(3), pp. 230-253, July 1983.

[Peterson 1985]
L. L. Peterson. Naming Users in a Heterogeneous Internet: Framework for a New Approach. Tech. Rep. 85-28, Dep. Comput. Sci., Univ. Arizona, Tucson, AZ, Nov. 1985.

[Postel 1984]
J. Postel. Domain Name System Implementation Schedule - Revised. RFC 921, USC Information Sci. Institute, Oct. 1984.

[Quarterman & Hoskins 1986]
J. S. Quarterman and J. C. Hoskins. Notable Computer Networks. *Commun. ACM*, 23(10), pp. 932-971, Oct. 1986.

[Redell & White 1983]
D. D. Redell and J. E. White. Interconnecting Electronic Mail Systems. *IEEE Computer Magazine*, 16(9), pp. 55-63, Sep. 1983.

[Satyanarayanan et al. 1985]
M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector and M. J. West. The ITC Distributed File System: Principles and Design. *Proc. 10th ACM Symp. Operating Syst. Prin.*, pp. 35-50, Dec. 1985.

[Schwartz 1987]
M. F. Schwartz. Naming in Large, Heterogeneous Systems. Ph.D. Diss., Tech. Rep. 87-08-01, Comput. Sci. Dep., Univ. Washington, Seattle, WA, Aug. 1987.

[Sheltzer, Lindell & Popek 1986]
A. B. Sheltzer, R. Lindell and G. J. Popek. Name Service Locality and Cache Design in a Distributed Operating System. *Proc. 6th Int. Conf. Distrib. Comput. Syst.*, pp. 515-522, May 1986.

[Shoch 1978]
J. F. Shoch. Inter-Network Naming, Addressing, and Routing. *Proc. 17th IEEE Comput. Society Int. Conf.*, pp. 72-79, Sep. 1978.

[Sun Microsystems 1985a]
Sun Microsystems. *Remote Procedure Call Programming Guide.* Sun Microsystems, Inc., Mountain View, CA, Jan. 1985.

[Sun Microsystems 1985b]
Sun Microsystems. *Remote Procedure Call Protocol Specification.* Sun Microsystems, Inc., Mountain View, CA, Jan. 1985.

[Templeton et al. 1986]
M. Templeton, D. Brill, A. Chen, S. Dao and E. Lund. Mermaid - Experiences with Network Operation. *Proc. 2nd IEEE Int. Conf. Data Eng.*, pp. 292-300, Feb. 1986.

[Terry 1985] D. B. Terry. Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments. Ph.D. Diss., Tech. Rep. UCB/CSD 85/228, Comput. Sci. Division, EECS, UCB, Berkeley, CA, 1985.

[Terry 1986] D. B. Terry. Structure-Free Name Management for Evolving Distributed Environments. *Proc. 6th Int. Conf. Distrib. Comput. Syst.*, pp. 502-508, May 1986.

[Terry et al.]
D. B. Terry, M. Painter, D. Riggle and S. Zhou. The Berkeley Internet Name Domain Server. *Proc. USENIX Association Summer 1984 Conf.*, pp. 23-31.

[Welch & Ousterhout 1986]
B. Welch and J. Ousterhout. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System. *Proc. 6th Int. Conf. Distrib. Comput. Syst.*, pp. 184-189, May 1986.

[Wiebe 1987]
D. Wiebe. Personal Communication. Comput. Sci. Dep., Univ. Washington, Seattle, WA, Jan. 1987.