Integrating Naming and Addressing of Persistent Data in Programming Language and Operating System Contexts

Sape J. Mullender
Martijn van der Valk
Faculty of Computer Science / SPA
University of Twente, Enschede
The Netherlands
e-mail: *mvdvalk@cs.utwente.nl*

There exist a number of desirable transparencies in distributed computing, viz., name transparency*: having a uniform way of naming entities in the system, regardless of their type or physical make up;* location transparency*: having a uniform way of addressing entities, regardless of their physical location;* representation transparency*: having a uniform way of representing data, which simplifies sharing data between applications written in different high-level languages and running on different hardware architectures (interoperability) and finally* invocation transparency*: having a uniform way of invoking operations on entities.*

The advent of persistency in programming language contexts has created a need for the integration of these four important concepts, viz., naming, addressing, representation and manipulation of data in programming language and operating system contexts. This paper attempts to address the first three transparencies, postponing the fourth to a later paper.

First, we make up a list of things that are needed to construct a persistent programming environment and relate this list to existing persistent object models, revealing their inadequacies. We then describe a new model which merges programming language and operating system naming contexts into a global name space which, while enforcing uniformity through the use of globally unique names, still allows the application of personal nicknames. Furthermore, we explain how persistent data is stored and retrieved using a client/server model of interaction, and how it could be acted upon correctly, through the concept of typed data. We conclude by checking how well our model scores on the wish list, listing the current status and future directions for research.

1 Introduction

A number of desirable transparencies in distributed computing may be distinguished, viz., *name transparency*: having a uniform way of naming entities in the system, regardless of their type or physical make up; *location transparency*: having a uniform way of addressing entities, regardless of their physical location; *representation transparency*: having a uniform way of representing data, regardless of the entity by which they were created; and finally *invocation transparency*: having a uniform way of invoking operations on entities. The distinction made between naming and addressing simplifies migration of named entities. Uniform data representations simplify sharing data between applications written in different high-level languages and running on different hardware architectures (interoperability). Invocation transparency simplifies enforcing that data be used in a way intended by the author.

Naming, addressing, [1], representation and manipulation of data within the context of programming languages and operation systems have always been treated separately. This separation severely limits making data persist over time and sharing of data across process or machine boundaries. UNIX is a prime example; process data structures are lost when a process that manages them exits unless they are specifically stored on the file system. The way in which these structures are named as files is application dependent. The lack of a uniform name space discourages sharing over a network. Second, it is difficult to share data across address space boundaries, since the most efficient data reference, viz., the pointer, is interpreted local to an address space. Another possible reference, the file descriptor, can be shared between processes with common ancestry only. The only reference which can be shared in a wider context is the name of a file. Third, mapping high-level programming language data structures onto UNIX files is non-trivial, since UNIX files are unstructured arrays of octets. Finally, UNIX files, apart from a distinction between user files and directories, are untyped, which renders them less useful for storing typed data.

Recently, a number of models have been proposed to merge programming and operating system contexts into a persistent programming model, allowing the use of kernel, networking and other operating system related data to be used from withing familiar programming languages. The object-based programming model seems to be particularly well-suited since it provides data structures with a way of drawing a clear line between their internal and external representations and allows them to provide well-defined interfaces to them with which a uniform invocation mechanism may be constructed. The object-based model does not help in providing the naming and addressing transparencies, though.

1.1 What We Want

This Section explains what we want out of a persistent object layer as was introduced in the previous Section. We envision a distributed computing environment integrating programming language and operating system contexts. It should be possible for user processes to access operating system data concerned with networking and kernel data structures in the same way as internal data structures, using the same naming and invocation mechanisms. Data types should be able to be shared between different architecture machines without having to worry about different data representations.

[1] In this article, the terms *naming*, *addressing* are used rather a lot. Naming is meant as associating an entity with a name and vice versa. Addressing is to be understood as locating the entity given its address, see [Shoch78] for clear definitions and [Saltzer78, Needham89] for an excellent general introduction into naming.

Furthermore, apart from a global naming context on which all processes agree, at different levels in the processing hierarchy, it ought to be possible for processing entities to construct their own view of the name space, according to certain conventions, constructing a mapping function between global and personalised name spaces. This local name space could pertain to individual processes as well as to process groups or even to all processes running on a particular machine. Personalised name spaces could be shared and/or inherited by child processes.

Another way of looking at this is to imagine names to fit a hierarchy in which higher-level names are mapped onto lower-level ones. At the bottom of the hierarchy, there are names which are interpreted by the hardware, such as Ethernet addresses, memory locations etc. However, there is no need for a global name space to be at the top of the hierarchy; it could be fit in at a convenient place by everyone using it.

1.2 Concepts

This Section describes the basic concepts of the sort of distributed computing model at which our persistent object layer has been targeted. We envision this to consist of a large, physically dispersed conglomerate of administrative domains (sites). Sites consist of machines (nodes) which are connected by a fast, highly reliable LAN. Sites are connected through a slower, less reliable WAN.

The distributed system model will typically provide support for many different kinds of data with a wide range of characteristics, ranging from multi-lingual text, executable code, graphical data, audio, video and other kinds of continuous media data. Most of this data critically depends on it being handled correctly. The canonical example of mistreatment would be to cache a video stream. Therefore, it is necessary to associate types with data. We believe that the object abstraction, see [Snyder93] for a terminology roundup, is useful for describing this concept. In the object paradigm, a type describes the structure of the data, as well as the operations which may be performed on the data. Each object is associated with a tag which identifies its type.

Objects represent the passive elements of our model. The active, processing elements are processes. Processes may consist of multiple threads of control, sharing an address space. A set of processes which manages objects is called a service. A service implements a name space in which the objects are put. Services may be implemented in a distributed fashion as separate servers. A process which wants to access objects on a service is called a client. Clients and servers communicate using remote procedure call (rpc) [Birrell84]. One of the key points of our rpc mechanism is that its transport connections are relatively long lived. The reason for this is that clients are authenticated to the service that they want to connect to before they may communicate. Authentication is a relatively expensive operation which one would like to see limited to connection setup time. Furthermore, since communication is likely to be encrypted, either end of the communication path will have to keep state. Once a connection has been set up, it remains open until explicitly closed by either client or service. Furthermore, the system itself may close a connection when it needs the resources. Connections are established and re-established transparently.

Application programmers are not restricted to one specific programming language for producing applications which use the features of our persistent object layer. The object-based programming paradigm can be used through the C++ interface which attempts to fully integrate program and operating system naming and invocation mechanisms, but the ordinary C language may still be used as well.

2 Related Work

In this Section, we briefly discuss other research on persistent object layers and how this related to our goals.

2.1 Arjuna

A project called Arjuna, done by the University of Newcastle upon Tyne, focuses on the design and implementation of an object-oriented programming system for creating fault-tolerant, distributed applications ([Shrivastava91, Dixon89]). It is based on the C++ language and the current implementation runs on top of UNIX. Arjuna object classes inherit from a base class that allows objects to persist when the process that has created them has exited.

A number of observations can be made about the system. First, naming and addressing of objects are intertwined in Arjuna. Arjuna runs on top of UNIX and stores its objects in UNIX files. The hierarchical name space of a UNIX file system is used to reflect the position of an object in the Arjuna class hierarchy. The leaf name of the file is the uid of the object. In order to distinguish between different UNIX file systems, a string name representing a particular server may be added to the name. Since the name for an object also hints at its location by including a node name, moving an object is not transparent to its use. Furthermore, naming objects is rather restricted, since it impossible to give an object a place in the hierarchy other than the one reflecting its position in the class hierarchy. Second, the deficiencies of C++ as a data structure definition language, such as the inability to know whether a character pointer points to a single character or to a string, can only be circumvented by assuming default behaviour, restricting the expressive power of the language. Finally, the way in which data structures are physically mapped onto secondary storage is hidden from the application programmer, which makes importing data in foreign software more difficult and prevents interoperability. In conclusion, we remark that Arjuna limits flexible naming, addressing and representation of data.

2.2 Opal

The Opal project at the University of Washington focuses on building an operating system which supports a single global virtual address space, to be shared by processes running on many (e.g., thousands) of nodes [Chase93]. Current state-of-the-art hardware allows for 64-bit wide address spaces to be used which may never run out of virtual addresses, even if memory is never reused; in [Chase92] it is said that a full 64-bit address space, consumed at the rate of 100 Mb/s, will last for nearly 5000 years.

In Opal, processes, although sharing the virtual memory, will run in their own separate protection domains and specialised hardware may be constructed that controls the access to virtual memory pages [Koldinger91]. A single virtual address space is context-independent, i.e., regardless of which entity utters a virtual memory address, all other entities will be able to interpret that address. This desirable property simplifies data sharing since pointers will still make sense across protection domain boundaries. Opal lifts the difference between names and addresses, because data is named after the region of virtual memory it possesses. Since virtual memory is never returned, once a piece of data is mapped one may be certain it will never be moved (i.e., renamed) under one's nose, thus avoiding the need for pointer translation on copy.

So far, so good. There is a catch, however, which is introduced by the fact that virtual memory is immutable: once a portion of virtual memory is assigned to a piece of data, it can never be reclaimed by the system. In a system consisting of thousands of nodes, virtual memory will be consumed rather quickly. Furthermore, since a portion of virtual memory is immutable, it cannot be allowed to grow. This means that a process requiring an a priori unknown amount of virtual memory, which is not uncommon, has always

to grab enough just to be sure. The problem how to determine how much is enough is non-trivial. It is for this reason that Opal does not allow objects to grow linearly. Immutable virtual memory certainly allows for easy copying of data since pointers may be preserved. However, in the case of heterogeneous architectures, there may still be a need for endian or floating point conversion. It is because of this that Opal does not support heterogeneity. Furthermore, the real problem in any large system is *finding* information. Being able to name and address data through virtual memory addresses, instead of having two separate schemes with a mapping in between, does not solve the problem of finding out whether a piece of data is physically present on a node or where it is stored on secondary storage.

We believe that a wide address space may certainly be used to simplify data sharing between a limited number of nodes (i.e., several dozens), but will not scale to a large conglomerate of nodes [Bartoli93]. This means that in the end, the same methods for sharing data over address space boundaries will be needed as is the case in today's process-based systems, albeit for sharing across a WAN or large LAN.

2.3 Clouds

The Clouds distributed operating system, see [Dasgupta90, Dasgupta88] is object-based in that all data, programs, devices and other resources are modelled as objects. Objects are the passive elements in the system, whereas threads are the active ones. An object can be viewed as a persistent virtual address space in which threads may start executing at certain entry points. In this respect, objects are rather coarse-grained. Objects are structured into segments: code segments are executed by threads. Data segments are accessible by executing threads, but not from outside the object.

Threads may shift execution between objects by calling another object's entry point with appropriate parameters. However, sharing of data between objects is restricted to parameter passing by threads: since each object implements its own address space, addresses cannot be shared. The mechanism of threads moving between persistent objects is termed *object-memory*.

Clouds objects bear globally (in a LAN context) unique names which do not contain addressing information. A name server translates between strings and unique names. Multicast is used to locate the object associated with a unique name. The underlying micro kernel provides a distributed shared memory abstraction with a coherence controller and a lightweight process mechanism. An object roughly maps onto a segment of memory in this environment.

Clouds does separate naming and addressing of objects. However, using multicast for locating objects in a distributed computing environment, is not likely to scale well. Since Clouds objects contain data as well as code, it does not seem easy to share them between different architecture machines, so representation is not really an issue here. Object invocation is through a well-specified entry-point in the code segment of the object which contains a jump table of additional entry-points. Static type checking is performed on the object and its entry points by the compiler. No run-time type checking is done.

2.4 Plan 9 from Bell Labs

Plan 9 is an operating system which was designed and implemented by researchers at Bell Labs. Plan 9 is to be viewed as a follow up to UNIX. In UNIX, the file system plays an important rôle in integrating the naming of files on secondary storage and devices. Plan 9 takes this idea a bit further, by making everything in the system appear to be a file. In this respect, the operating system may be viewed as file-based instead of object-

based[2] [Pike90]. Because all data is to be treated as files, each server which governs data implements its own file name space, e.g., the kernel maintains a `/proc` file tree, in which subdirectories represent processes, whose code, data and state may be examined by reading the constituent files. Also, network connections can be examined and controlled using normal file operations etc.

Plan 9 does not pursue the goal of providing each process with a single global name space in which all server name spaces are embedded. Instead, processes implement their own name space by mounting server name spaces as they please, an idea pioneered in [Comer86]. This seemingly anarchic system works through the following of certain conventions. As an example, a process running on a DEC station would typically mount `/usr/bin.mips` as `/bin` in its personal name space, whereas a process running on a HP 700 would mount `/usr/bin.snake` under that name. Furthermore, Plan 9 allows child processes to inherit a name space and process groups to share name spaces.

Plan 9 uses a standard message protocol for client/server communication, called $9P$. $9P$ is connection-oriented and it consists of rather simple operations for doing name space operations and primitive data transfer and file operations. Since many servers encode data transferred in $9P$ messages using Unicode, their contents may be easily shared.

Our observations about Plan 9 are the following. First, Plan 9 on purpose does not implement any global name space since its designers see this as impossible because of the great variety of different kinds of naming schemes used in different kinds of networking hardware. Instead they provide a way of naming through the use of conventions. Second, Plan 9 has tried to solve the problem of a uniform representation, albeit at a price. It remains to be seen whether this scheme is good enough for data types that are more complicated than integers and simple strings. Third, Plan 9 does provides a uniform invocation mechanism on data, through the file metaphor. However, this sometimes leads to rather contrived constructions, and in some areas the metaphor is simply dropped, e.g., processes may not be created or terminated by creating or deleting directories in the `/proc` file tree.

3 Description

As part of the Huygens[3] project [Mullender91], which is the University of Twente umbrella project for research on distributed systems, research on continuous media and its implications on distributed system design is being carried out under the cover of the Pegasus[4] project [Leslie93]. A primary goal of Pegasus is to provide support for continuous media at the operating system level. This Section describes the design of a persistent object layer in the Pegasus distributed computing environment which we are currently developing.

3.1 Naming and Binding

This Section discusses how clients refer to objects managed by a service through binding names to handles. A service implements a name space in which its objects are placed. A user process, acting as a client, may traverse this name space, storing references, called handles, to these objects in a private name space. In order for a client to manipulate an object managed by a service, the client should be able to take a name appearing in its

[2] "File systems are cool things (Rob Pike)"

[3] The Huygens Project is partially supported by the ESPRIT BRA projects Pegasus (BRA 6586) and BROADCAST (BRA 6360); Digital Equipment Corporation; Xerox EuroPARC Cambridge; Olivetti Research Laboratory Cambridge; and Hewlett-Packard Laboratories, Palo Alto.

[4] The Pegasus Project is a project of the Universities of Twente and Cambridge, supported by the European Communities' ESPRIT Programme through BRA project 6586. It is partially supported by the Cambridge Olivetti Research Laboratory and a grant from Digital Equipment Corporation.

name space and use this for setting up a connection with the corresponding object. This involves additional concepts and some machinery.

The client uses a mechanism called name resolution, which is discussed in Section 3.2, to bind a name to the corresponding handle. A handle is always associated with exactly one object in some service's name space. Handles have a limited life time, hence they may be cached. Handles may be in two states: connected and unconnected; a handle is said to be connected when it forms the client-end of a communication channel with the service managing the object.

An object handle is implemented as a fixed-size data structure that stores the address of the object to which it belongs, as a server name/object uid pair, and further encapsulates connection state information, such as the network protocol used, a connection identifier, a message counter to detect lost/duplicate messages, and additional state having to do with encryption. When an object is used for the first time, the handle is bound to an association between the client and the service managing the object. In our model, the association takes the form of a long-lived rpc connection. Section 3.5 discusses a way of using existing mechanisms for implementing object handles.

The two levels of binding that occur between names and objects are depicted below. The first binding is performed by the name resolution mechanism; the second by the networking software.

$$name \longrightarrow handle \longrightarrow association$$

Once a name has been bound to a handle, the name service may be circumvented in subsequent communication; a similar approach is taken in the V distributed operating system [Cheriton88]. [Saltzer93] provides more background in the different occasions at which binding occurs.

3.2 A Name Service

In our model, we incorporate the idea of the per-process mount table, as in Plan 9. Each process maintains a private name space containing as leaves, object handle/object name pairs. When a process is created, it inherits its name space from the parent process. The child may either share its name space or decide to "go its own way". Initially, the only object handle present in the name space corresponds to the root of a special service, called the name service. The address of the name service therefore has to be known a priori. As a process obtains a new object handle, it may put it in its own name space wherever it chooses.

The name service maintains a name space containing object handles to other services in the distributed computing model, paired with names. Therefore, when a service wants to make itself known to the world, it has to register itself with the name service, i.e., leave its address and a name at the name service. Although processes may construct name spaces to their own liking, as long as they hold on to this name service handle, they will always be able to resolve global names. Since the name service is such an important service, it will typically be replicated within a site. Mechanisms for keeping the data bases of the respective replicas consistent are beyond the scope of this paper.

Name resolution starts in the per-process name space. If during traversal of the name space an object handle is reached, the resolution process continues at the corresponding server name space. Recursion is possible. At this point we have not said anything about what a service address looks like, as it is stored in an object handle. If it is a network address, port pair, this would mean that an object handle becomes stale whenever a host is moved to another network attachment point on the LAN. However, if it is a symbolical name, it may the case that name resolution enters a loop, when the name being resolved is stored on a name server which own name starts with the name being resolved, consult [Lampson86] for sufficient restrictions on names to circumvent this problem.

3.3 Client/Server Communication

An important part of the communication between clients and servers has to do with name space operations, such as creation, listing, traversal and removal of directories, and connection management, such as establishing, authenticating, duplicating and subsequent discarding of connections. In order to come up with a suitable protocol, we have taken the Plan 9 client/server protocol, called $9P$, and isolated these operations in a new protocol called $2P$ (pronounced *Tuppence*), a name chosen to commemorate its ancestor. Since name space and connection management operations are independent of the type of objects which are managed by the service, the $2P$ protocol is known to all of them. $2P$ is combined with a object-type specific protocol, called $7P$. The resulting protocol is aptly named $9\frac{1}{2}P$, since no protocol is without overhead, as we will soon see.

The $9\frac{1}{2}P$ protocol is a reliable fifo-ordered message protocol. The $2P$ part is used by clients to traverse the server name space, duplicating and discarding connections to objects in it and performing raw data transfer operations, as well as other standard file operations, such as getting and setting the status of an object, renaming it and creating and removing objects and directories. The $7P$ part is used to invoke type-specific operations on objects, so as a consequence, $7P$ differs per object type. In order to switch between the two protocols, we need an additional message, which accounts for the $\frac{1}{2}P$.

Clients do not use $9\frac{1}{2}P$ messages directly. Instead, they use a procedure call library which is linked with the executing code. Since $2P$ is generic it may be statically linked with the client. It is possible and potentially useful to delay linking the $7P$ libraries with the client until it is first invoked, using dynamic linking. The $\frac{1}{2}P$ switch is used to let the server know that it should send the necessary data needed by the client to install the corresponding $7P$ library. The details of how this is done are beyond the scope of this article.

3.4 Programming Language Interface

This Section briefly discusses a possible C++ interface which hides as much of the complexity of delayed linking as possible in cleverly overloaded operators. For example:

```
ObjHand cm = root/"usr/local/cmfs"; // 1
CMdoc p2i = cm/"films/PassageToIndia"; // 2
p2i.open(); // 3
p2i.play(); // 4
```

In statement 1, an alias for "usr/local/cmfs" (a service) is constructed. Note that `root` is a predeclared variable and linked into the code. This alias is extended in statement 2; `p2i` is an instance of a class inheriting from `ObjHand`. The switch between $2P$ and $7P$ is thrown in statement 3, after which the library corresponding with `CMdoc` is automatically fetched and dynamically linked with the application code; details are beyond the scope of this paper. Statement 4: the invocation of a method belonging to type `CMdoc`.

3.5 Object References

This Section discusses using existing mechanisms for implementing the concepts mentioned in Section 3.1. Figure 1 depicts a client and a service (drawn in grey). Object handles, as recorded in the client mount table, can be implemented using pointer pairs, called *maillons* by Marc Shapiro. A maillon contains a data and a code pointer. The code pointer refers to a procedure which is present in the stub at creation time. It is able to initiate communication with the service and install the methods pertaining to the object with which the handle is associated. The data pointer refers to a library (in OOP, this is called *method table*), which is able to communicate through the stub with the server end of the channel. Part of the functionality of this library, viz., the $7P$ part, depends on the

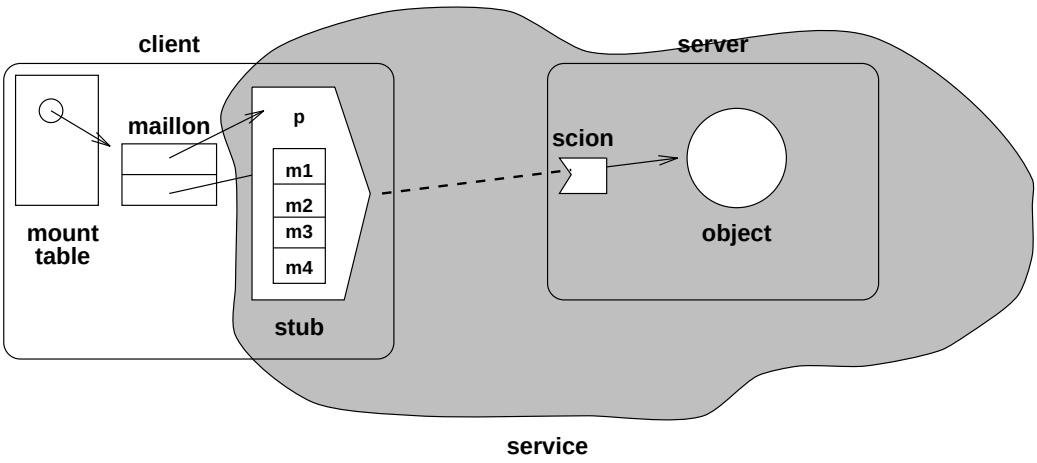

Figure 1. Referring to an Object

type of the object. To indicate that the stub and method table are installed by the server, they extend into the grey area. Although not shown in the picture, the service could insist on the client installing additional machinery on its side for communicating with it, e.g., a client cache manager in the case of a distributed file service, etc.

In [shapiro92], a distributed object reference mechanism is described which is based on chains of client and server stubs, called SSPs, which stands for stub-scion pairs. SSP chains allow for objects to migrate or be removed from a service, and references to be passed around, almost without additional overhead on the side of the client. In addition, it carries support for garbage collection of objects that have become unreachable.

An SSP chain consists of one or more pairs of stubs which together form a reference to an object managed by a service. A client may pass the reference to other clients, extending the chain in its direction. Also, the service may move the object, extending the chain the other way. A long SSP chain may be shortened by adding a weak link which points directly to the service storing the object. The complementary strong link is used to record all spaces which hold a reference to the object; this information is used by a distributed garbage collection mechanism.

4 Implementation

This Section fleshes out some of the details concerning the implementation of the elements comprising our persistent object model: name server, object handle, $9\frac{1}{2}P$ protocol. Name resolution as described in Section 3.2 is a multi-stage process: names are partionally resolved inside a per-process context. When successful, name resolution returns an object handle which is used to connect to a server storing the rest of the name. The rest of the name is resolved by traversing the name space implemented by the server using the $2P$ protocol.

A process stores object handles of interest in a data structure called the mount table. The mount table is a per-address space data structure. Threads executing in the address space may share a mount table but they may also construct their own. We believe that it is useful for mount tables to be passed to other processes, crossing address space boundaries. Therefore, we need a way to efficiently marshal and unmarshal mount tables.

The mount table data structure is a directed acyclic graph. It links entries with hierarchical names. Non-leaf entries are called directories; leaf entries contain object

handles. Directories are used to add structure to the name space by clustering object handles; they act pretty much like directories in a file system. The hierarchy implied by directories cuts an object handle's name into a finite number of strings which are separated by slash characters; each except the last one pertaining to a directory.

For implementational simplicity, it is much simpler for the name space to be a tree instead of a directed acyclic graph (dag). However, this limitation would prohibit the use of personal nick names or short cuts in name resolution. Therefore, we allow for different names to pertain to the same entry; these names are called aliases, although once created there is no fundamental difference between a name and its aliases. An entry is kept in the mount table until its last alias has been removed.

4.1 Mount Table

The mount table is an in-core data structure, managed by one or more threads. A mount table is stored as a fixed-size array of entries which is allocated at creation time. The mount table can be expanded when it becomes full. Entries store part of a hierarchical name, some type-specific data and a number of pointers. The pointers maintain the dag structure of the name space; each entry stores references to its respective parent, first child and prior and next siblings. In addition, each entry stores a reference to the prior and next free entries in the table.

The advantage of pre-allocation of a large array is that all the data that makes up the structure is stored in consecutive virtual memory which makes for efficient copying and removing of entire mount tables. Furthermore, it simplifies implementation. Entries are small and fixed size, so the overhead in terms of memory consumption can be kept to an acceptable level.

4.2 Service Registration

As has been discussed in Section 3.2, in order to be reachable by clients, servers have to make their presence known to them by registering at well-known points, viz., a name service. Initially, an object handle to the name service root is the only object handle stored in a mount table.

We intend to use the Domain Name System (DNS) [Mockapetris87b] as a name service. At present, DNS only allows the storage of a limited set of different kinds of records in its data base, mainly to do with IP address resolution, e-mail routing and the name resolution mechanism itself. By adding new types of records to its data base, we intend to extend the use of DNS as a general purpose name service, storing information on a large variety of services. The name space implemented by DNS consists of a hierarchy of domains which are named by dot-separated strings. A front end to DNS can be used to translate between our naming scheme and the DNS, restricting the use of wild cards where necessary.

4.3 The 2P Protocol

The protocol used by a client to communicate with a server is called $9\frac{1}{2}P$. $9\frac{1}{2}P$ consists of two protocols, $2P$, which is used for server name space, connection management and raw data transfer operations, and $9\frac{1}{2}P$, a data-type specific protocol.

$9\frac{1}{2}P$ is a connection-oriented protocol. A connection is setup by sending a `connect` message. Before a client may start making requests to a server, the server may indicate its desire for the client to authenticate itself. At present, $2P$ supports a simple challenge/response algorithm. A connection corresponds to exactly one object in the server's name space. The object at hand may be changed by the client using the `walk`

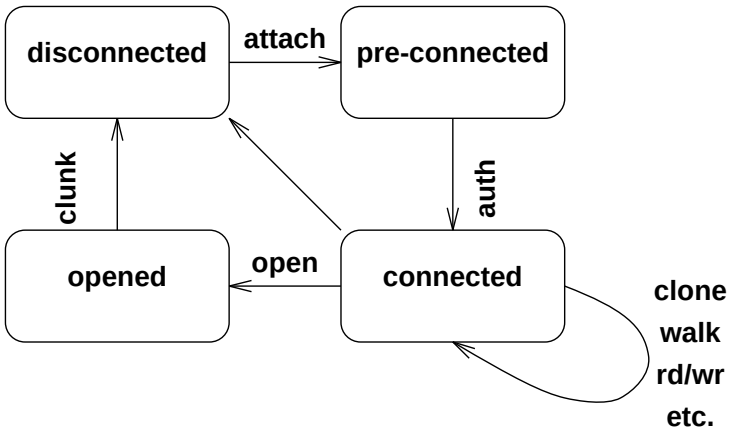

Figure 2. $2P$ **State Machine**

operation. A connection may be duplicated using `clone`, which means that a new connection is created. Cloning a channel instead of manually setting up a new connection is desirable since it may circumvent the relatively expensive authentication procedure.

The client may perform simple operations on a server object using $2P$, such as create, remove. Furthermore, $2P$ supports the concept of a directory, which may be used by the server to cluster objects. Directories may be created, removed etc. When a client wishes to start using the object according to its type, it has to open it explicitly using `open`. Afterwards, it may `close` it. The precise nature of these two operations has not been defined yet.

A connection may be discarded by `clunk`ing it. Communication is usually initiated by the client in the form of requests to which the server replies. The server may however call back on the client in the case of an emergency, such as a change of state of the object with which the connection is associated. If the server implements a file system, it may call back on the client to invalidate a cached copy. This form of call back is always fatal for the connection which is automatically tore down.

Note that the previous discussion by no means exhaustively listed all $2P$ messages. The $2P$ protocol state machine is shown in Figure 2.

Client and server threads do not directly use the $9\frac{1}{2}P$ messages when making requests. Instead, an object handle is used through which invocations are made using two sets of procedure calls, one of which implements $2P$, the other of which implements $9\frac{1}{2}P$. The object handle maintains the state of a connection, including a message count for detection of lost or duplicate messages. In addition to the states that the protocol may be in, the object handle can be in a state called *limbo*, which means that the object which address it records has become stale. An object handle reaches this state when the server calls back to invalidate the object. The result of this is that the addressing information is removed from the handle.

5 Conclusions

We conclude by relating our model to each of the four transparencies and list the current status of the project.

Our model makes a clear distinction between naming and addressing entities. The per-process name service allows processes to construct their own personalised view of the service/objects hierarchy and pass this view around to others. Name resolution is partially done in the context of the client, and partially by the server. Succesful name

resolution results in an object handle which may be used in subsequent communication with the server managing the object, circumventing the name service. The object handles corresponding to entry points of servers name spaces are registered by these servers at a name service.

Object handles could be made to tolerate object migration through the use of SSP chains to record server/object addresses. At this moment we have no clear view on what format to use for representing data structures in a uniform way; the view that Plan 9 takes to represent everything as a Unicode string does not appeal to us for reasons of efficiency. The way in which objects are invoked is through a procedure call library which is type dependent and linking with the client process may be postponed until first invocation. The direction that Arjuna takes with tying object classes to the names of persistent objects does not appeal to us. We make a clear distinction between class hierarchies and naming hierarchies.

Currently, procedure libraries exist for constructing and interpreting $2P$ mes-sages. Client/server communication using $9\frac{1}{2}P$ has been tested using a prototype multi-threaded server which runs on top of UNIX, using a user threads package. As network layers, both conventional TCP/IP [Postel80] and the Multi Services Network Layer [McAuly89] have been used. MSNL has been designed to be used on ATM hardware, which we intend to apply as soon as such technology becomes available to us. We are interested in mapping each client/server rpc connection onto an ATM virtual circuit. Currently we are implementing the mount table and object handle abstraction in C++.

A C interface for querying the DNS has been finished [Mockapetris87a]. We intend to use it as the basis for our name service. Unfortunately, the current implementation of `named` does not allow its data base of resource records to be updated. We intend to construct a patched version which will allow us to do this.

One of the services we intend to use for experiments with our name resolution mechanism is a service managing Active Badge readings [Want92] and a replication service.

References

[Bartoli93] Alberto Bartoli, Sape J. Mullender, and Martijn van der Valk. Wide-address spaces - exploring the design space. *SIGOPS*, **27**(1):11–17, jan 1993.

[Birrell84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, **2**(1):39–59, February 1984.

[Chase92] Jeffrey S. Chase, Henry M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical report 92-03-02. Department of Computer Sc. and Engineering, U. of Washington, Seattle, WA (USA), March 1992.

[Chase93] Jeff Chase, Valerie Issarny, and Hank Levy. Distribution in a Single Address Space Operating System. *OSR*, **27**(2):61–5, Apr 1993.

[Cheriton88] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, **31**(3):314–33, March 1988.

[Comer86] Douglas Comer and Thomas P. Murtagh. The Tilde File Naming Scheme. *6th International Conference on Distributed Computing Systems* (Cambridge, MA, USA), pages 509–514. IEEE, May 1986.

[Dasgupta88] Partha Dasgupta, Richard J. LeBlanc Jr, and William F. Appelbe. The Clouds Distributed Operating System. *Proceedings of 8th International Conference on Distributed Computing Systems* (San Jose, CA), pages 2–9. IEEE Computer Society Press, catalog number 88CH2541-1, 13–17 June 1988.

[Dasgupta90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, **3**, 1990.

[Dixon89] G. N. Dixon, G. D. Parrington, S. K. Shivastava, and S. M. Wheater. The treatment of persistent objects in Arjuna. *The Computer Journal*, **32**(4):323–32, 1989.

[Koldinger91] Eric J. Koldinger, Henry M. Levy, Jeffery S. Chase, and Susan J. Eggers. The protections lookaside buffer; efficient protection for single address-space computers. Techical Report 91-11-05. Departement of Computer Science and Engineering, University of Washington, Seattle, WA 98195, nov 1991.

[Lampson86] Butler W. Lampson. Designing a global name service. *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, page 10. ACM, 1986.

[Leslie93] Ian Leslie, Derek McAuley, and Sape J. Mullender. Pegasus - operating system supports for distributed multimedia systems. *SIGOPS*, **27**(1):69–78, jan 1993.

[McAuly89] Derek Robert McAuley. *Protocol Design for High Speed Networks*. PhD thesis, published as Technical report 186. University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, January 1990.

[Mockapetris87a] P. Mockapetris. Domain names — implementation and specification. Request for comments 1035. ARPA Network Working Group, nov 1987.

[Mockapetris87b] P. Mockapetris. Domain names – concepts and facilities. Request for comments 1034. ARPA Network Working Group, November 1987.

[Mullender91] The Huygens project: multimedia and distributed systems research at the University of Twente. Twente University, Netherlands, Draft of 13 May 1991.

[Needham89] R. M. Needham. Chapter 5: Names. In Sape Journal Mullender, editor, *Distributed Systems*, Frontier Series, pages 89–101, first edition. New York, 1989.

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *UKUUG Summer 1990* (London), pages 1–9. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.

[Postel80] J. Postel. *Transmission Control Protocol*, Technical report RFC–761. USC Information Sciences Institute, January 1980.

[Saltzer78] J. H. Saltzer. Naming and binding of objects. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: an Advanced Course*, volume 60 of Lecture Notes in Computer Science, pages 99–208. Springer-Verlag, Berlin, 1978.

[Saltzer93] Jerome Saltzer. On the Naming and Binding of Network Destinations. Request for comments 1498. ARPA Network Working Group, Aug 1993.

[shapiro92] Marc Shapiro, Peter Dickman, and David Plainfosse. SSP chains: robust, distributed references supporting acyclic garbage collection. Technical report 1799. Institut National de Recherce en Informatique et en Automatique, nov 1992.

[Shoch78] John F. Shoch. Inter-network naming, addressing, and routing. *Proceedings of COMPCON 78* (Spring 1978), pages 72–9. IEEE, Spring 1978.

[Shrivastava91] Santosh Shrivastava, Graeme Dixon, and Graham Parrington. An overview of the Arjuna Distributed programming system. *IEEE Software*, January 1991.

[Snyder93] Alan Snyder. The essence of objects: Concepts and terms. *IEEE Software*, pages 31–42, January 1993.

[Want92] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, **10**(1):91–102, jan 1992.

Contents