Distributed Naming in a Factored Operating System

by

Nathan Beckmann

B.S. Computer Science, University of California, Los Angeles (2008)
B.S. Mathematics, University of California, Los Angeles (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

© Nathan Beckmann, MMX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
September 3, 2010

Certified by . . . .
Anant Agarwal
Professor
Thesis Supervisor

Accepted by . . . .
Terry Orlando
Chairman, Department Committee on Graduate Theses

Distributed Naming in a Factored Operating System

by

Nathan Beckmann

Submitted to the Department of Electrical Engineering and Computer Science
on September 3, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

A factored operating system (fos) is a new operating system design for manycore and cloud computers. In fos, OS services are separated from application code and run on distinct cores. Furthermore, each service is split into a *fleet*, or parallel set of cooperating processes that communicate using messages.

Applications discover OS services through a distributed, dynamic name service. Each core runs a thin microkernel, and applications link in a user-space library called *libfos* that translates service requests into messages. The name service facilitates message delivery by looking up service locations and load balancing within service fleets. *libfos* caches service locations in a private cache to accelerate message delivery, and invalid entries are detected and invalidated by the microkernel.

As messaging is the primary communication medium in fos, the name service plays a foundational role in the system. It enables key concepts of fos's design, such as fleets, communication locality, elasticity, and spatial scheduling. It is also one of the first complex services implemented in fos, and its implementation provides insight into issues one encounters while developing a distributed fos service.

This thesis describes the design and implementation of the naming system in fos, including the naming and messaging system within each application and the distributed name service itself. Scaling numbers for the name service are presented for various workloads, as well as end-to-end performance numbers for two benchmarks. These numbers indicate good scaling of the name service with expected usage patterns, and superior messaging performance of the new naming system when compared with its prior implementation. The thesis concludes with research directions for future work.

Thesis Supervisor: Anant Agarwal
Title: Professor

Acknowledgments

I would like to thank Prof. Agarwal for being an excellent advisor and my colleagues on the fos project for discussing ideas and providing the system in which this research was conducted. In particular, I would like to thank Harshad Kasture for discussing numerous bootstrapping issues and design decisions, and Charles Gruenwald III for discussing usage of the name service.

I would like to thank my girlfriend, Deirdre Connolly, as well as my family, Ed, Shelley, and Jessica, for providing love and support. Finally, I would like to thank Leela Cat for providing a much-needed distraction.

Contents

1 Introduction **13**

2 Overview of fos **17**

2.1 System architecture 17

2.2 Messaging 19

2.3 Fleets 21

2.3.1 Elasticity 23

2.3.2 Scheduling 23

2.4 Implementation 24

3 Motivation for a Name Service **27**

4 Design **29**

4.1 Application Interface 29

4.2 Messaging 34

4.2.1 Mailbox Structure 34

4.2.2 Name Service 36

4.2.3 User-space Name Cache 37

4.2.4 Message Delivery 38

4.3 Name Service 40

4.4 Distributed State 44

4.4.1 Programming Model 45

4.4.2 Code Flow 47

4.4.3 Concrete Example . . . 52
4.4.4 Management . . . 53
4.4.5 Summary . . . 55

5 Implementation **57**
5.1 Scope . . . 57
5.2 Bootstrapping . . . 59
5.3 Development Issues . . . 62
5.4 Experiences . . . 65

6 Results **67**
6.1 Methodology . . . 67
6.2 Micro-benchmarks . . . 68
6.3 End-to-End . . . 72
6.4 Discussion . . . 74

7 Future Work **77**
7.1 Expectations versus Reality . . . 77
7.2 Local Sub-domains . . . 77
7.3 Fairness in Conflict Resolution . . . 78
7.4 Bootstrapping Cloud Computers . . . 79

8 Related Work **81**
8.1 Full System . . . 81
8.2 Naming and Distributed Data . . . 83

9 Conclusion **85**

List of Figures

1-1 Introduction to name service 14

2-1 System overview 18
2-2 Messaging system 20

4-1 Two-phase commit code listing 47
4-2 Arbiter code listing 49
4-3 `Insert` code listing 52

6-1 Lookup results 68
6-2 Registration results 69
6-3 Registration results under contention 71
6-4 Messaging performance 72
6-5 File system results 73

List of Tables

4.1 Messaging API 30
4.2 Name service API 35
4.3 Name service management API 41
4.4 Distributed data store API 42
4.5 Microkernel name registration API 43
4.6 Distributed data store management API 45
4.7 Modify lock API 54

Chapter 1

Introduction

fos is a new research operating system targeted at manycore processors and cloud computers. These two emerging areas have introduced unprecedented hardware parallelism to the OS. In multicore, research chips from major manufacturers already have 32 or 48 cores, and embedded vendors currently offer general-purpose processors with 64 or 100 cores. Within a decade, chips will be available with hundreds of cores, and research prototypes will have thousands. Similarly, the cloud has brought large numbers of cores within a single logical system. However, current cloud solutions create artificial barriers in the management of these systems by introducing a virtual machine layer. fos eliminates these barriers by using a single OS image across multiple computers in a cloud.

Traditional, monolithic OSes were designed in an era of uniprocessors, and their design reflects this. These OSes time multiplex OS processing with application code via a system call interface. This design is extremely efficient with processor time, but it neglects other factors which are increasingly important in multicore; *e.g.*, data locality (cache behavior), unnecessary data sharing, and performance degradation due to lock contention within a single service. Massive parallelization efforts have begun in the monolithic OS community, but it remains unlikely that monolithic OSes represent the optimal design for manycore and cloud computers.

fos addresses these scalability challenges by *factoring* the operating system into individual services, each of which run independently of each other on separate physical

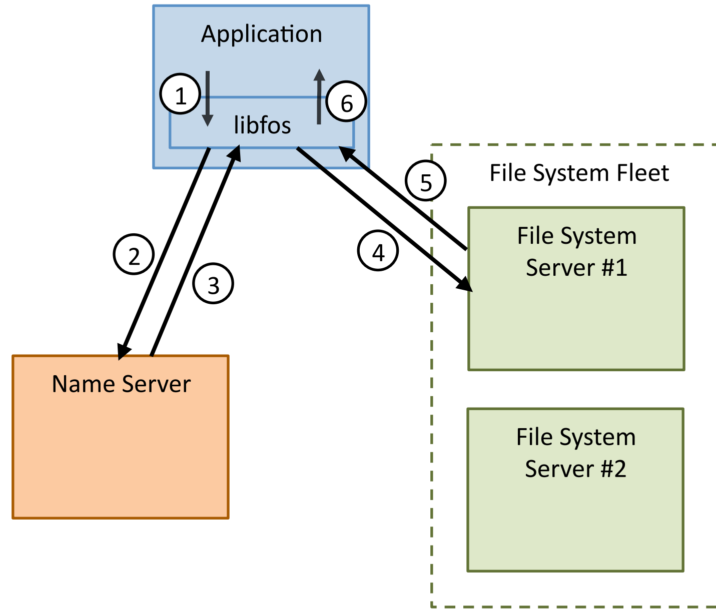

Figure 1-1: Using the name service to access a file system. Application requests are translated through *libfos* into messages, whose routing is facilitated by the name service.

cores. Additionally, fos splits each service into a *fleet* of cooperating processes, which in aggregate implement the service. For example, in a fos system there will be a name service fleet, a scheduling service fleet, a file system fleet, etc.. Factoring the OS into fleets provides a natural framework in which to address scalability, as each service is now an independent entity. In summary, fos is a collection of distributed systems which collectively implement a traditional OS interface.

fos services interact with applications and each other exclusively through messages (in particular, they do not share memory). This is done to clearly expose sharing of data, which can often become scalability bottleneck, as well as to provide a communication mechanism that can be implemented on a variety of architectures. This is important in order to support cloud computers, embedded non-shared-memory systems, and future architectures where global shared memory may prove inviable.

Applications link in a user-space library, *libfos*, that translates requests to services into messages. *libfos* maintains a cache of service locations that are discovered via a dynamic, distributed name service. Figure 1-1 shows the name service in action.

The system contains an application, a name service fleet with a single member, and a file system fleet with two members. The application makes a request to the file system, which is translated into a message in *libfos*. *libfos* messages the name service to resolve a member of the file system fleet, and the name server responds with a specific member of the fleet. This member is subsequently messaged and responds to complete the request.

As messaging is the primary communication medium in fos, the name service plays a foundational role in the system. Its performance is paramount, as the service is often on the critical path to message delivery. Furthermore, the name service enables some of the main ideas in fos: *service fleets*, by load balancing among several members of a fleet; *spatial scheduling*, or layout of processes within the system; and *elasticity*, or dynamically growing and shrinking fleets to reflect the current environment. Finally, the name service simplifies the construction of some services (*e.g.*, network stack) by allowing them to offload some of their distributed state into the naming system.

This thesis discusses the design and implementation of the naming system in fos. This includes both the name service itself and the naming/messaging library in *libfos*. In order to give high performance, it is necessary for applications to privately cache responses from the name service. However, since the name space is being constantly updated, these cached entries will become invalid. Therefore, there must exist error detection and recovery mechanisms to guarantee correct behavior. This design spans the messaging library and the name service, and both must enter into a contract in order to yield a functional system.

The name service is also one of the first complex services implemented in fos, and certainly the service that has the most distributed state. The experience of its implementation therefore gives insight into developing distributed OS services. The implementation involved several bootstrapping issues unique to the name service, but many of the difficulties were generally applicable.

The major contributions of contained herein are as follows:

- This thesis presents the complete design of the naming system, including the name service fleet and messaging in *libfos*. It demonstrates how the name service

enables fleets and simplifies their construction.

- This thesis presents the design of the first distributed key-value store in fos, and experiences gained from its development.

- This thesis serves as evidence that fleets are a viable design for foundational OS services, and that their performance and scalability can exceed naive, monolithic implementations.

This thesis is organized as follows: Chapter 2 overviews the design of fos to give context for later discussion. Chapter 3 motivates the name service in light of fos's design. Chapter 4 covers the design of the name service, including its external interface and internal algorithms and data structures. Chapter 5 discusses experience gained while implementing the name service. Chapter 6 presents results for the name service. Chapter 7 discusses future research directions. Chapter 8 discusses related work, and Chapter 9 concludes.

Chapter 2

Overview of fos

fos addresses OS scalability by implementing OS services as independent distributed systems running on separate cores from applications. fos employs a microkernel design in order to factor traditional OS services into user-space, where they are further split into *fleets*. A user-space library, *libfos*, provides messaging functionality for both applications and services and transparently multiplexes different message delivery mechanisms behind a unified messaging API. This chapter gives an overview of fos's system architecture that enables this goal. The central concepts in designing fos services are discussed next, and the state of the current fos implementation concludes. This chapter is a review of the fos project and represents the work of many contributors. Details can be found in [26, 27].

2.1 System architecture

Figure 2-1 shows a high-level depiction of a manycore fos system. Several applications are running, and OS services run on distinct cores from the applications. The OS is split into individual services, each of which is further parallelized into a fleet of cooperating servers. Fleet members are distributed throughout the chip to minimize communication costs with applications. For example, consider the file system fleet (shown as FS in the figure). This fleet has four members which are dispersed over the chip. The fleet members must collaborate in order to provide the interface of a

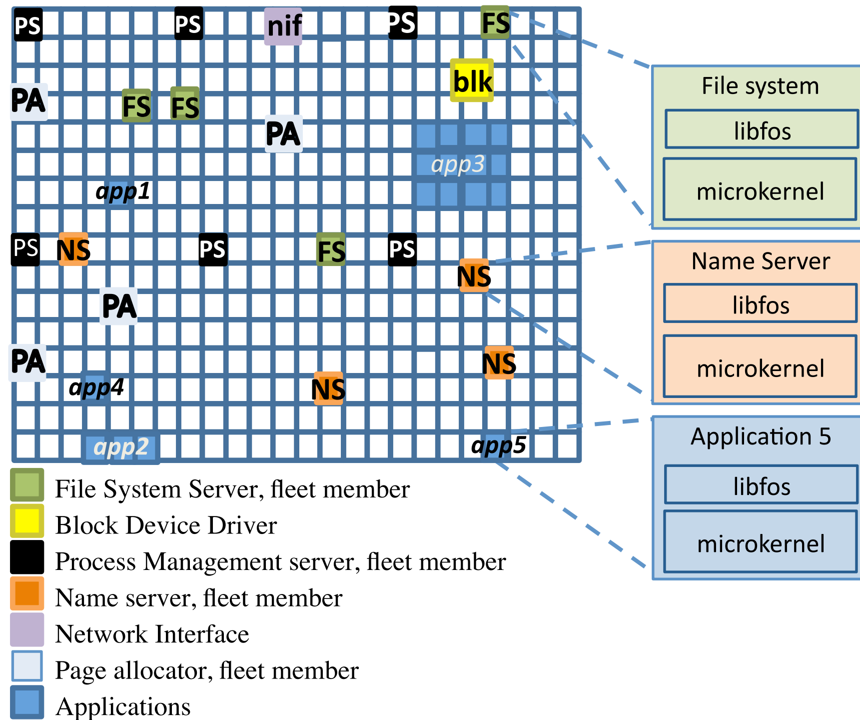

Figure 2-1: A depiction of a typical manycore fos system.

single file system to applications.

For the remainder of the thesis, "fleet member" and "server" both refer to processes that implement an OS service, as opposed to an "application," and "process" is used when discussion applies equally to servers and applications.

Within a single core, each application or service runs atop a minimal microkernel as shown in Figure 2-1. This microkernel provides protection (*e.g.*, of page tables) via a capability system and a basic messaging delivery mechanism. The microkernel must also handle hardware interrupts, faults, and certain bootstrapping operations. This thesis improves upon previously published work in fos [26, 27] by factoring naming out of the microkernel into *libfos*.

Whenever possible, functionality is moved into user-space in order to exploit fos's service model. For example, the handling of network packets is exported to the network interface service. The microkernel receives interrupts from the network card and messages the network interface service with the raw data, where processing takes

place. The process management service also handles many tasks that are typically found in the kernel, even in microkernel designs, such as setting up the page table of a new process.

Applications interact with fos via a library layer, *libfos*. *libfos* consists of two parts: (i) common code that is needed by all applications and services, such as messaging and naming; and (ii) library components from each OS service that provide local functionality for applications (*e.g.*, caching) and translate requests into messages to the service. OS services themselves are implemented atop *libfos*, but do not commonly exercise the library components of other services. However, because the OS is implemented as essentially unprivileged, user-space processes, services can make full use of other components of the OS if needed. For example, the process management service uses the file system to load programs off disk, and all services use the name service identically to implement messaging.[1]

2.2 Messaging

The name service is intimately related to fos's messaging system, and the behavior of the name service is heavily influenced by the guarantees that the messaging system provides. Therefore, it will be discribed in some depth to give proper context for later discussion.

fos uses messages as the primary communication medium between applications and the OS as well as within fleets themselves. Although fos supports shared memory for applications, OS services interact exclusively through messages. This is done for several reasons: Although perhaps more difficult to implement, messaging explicitly shows what data must be shared, and therefore it can lead to better encapsulation and early consideration of scalability bottlenecks. More importantly, messaging can be implemented on a wide variety of architectures and systems. In particular, for multiple-machine fos systems (*e.g.*, cloud computers) shared memory is unavailable, and one is forced to use a message-passing model. Embedded systems are often

[1]Including, to a large extent, the name service itself (Section 5.2).

implemented without shared memory support, and it also may be the case that global shared memory proves unscalable in large multicores. Message-passing provides the greatest flexibility in the underlying hardware.

fos's messaging system uses a mailbox-based application program interface (API). A service (or application) creates a mailbox on which it can receive messages. These mailboxes are registered with the name service using a hierarchical name space much like a UNIX path or web address. For example, the file system may register its mailbox as `/sys/fs`. In order to send to a service, the sender must have the name[2] of its mailbox and an associated capability. These capabilities are created upon mailbox creation, or can be manually created and added by request. The messaging API then contacts the name service and microkernel as necessary to complete message delivery.

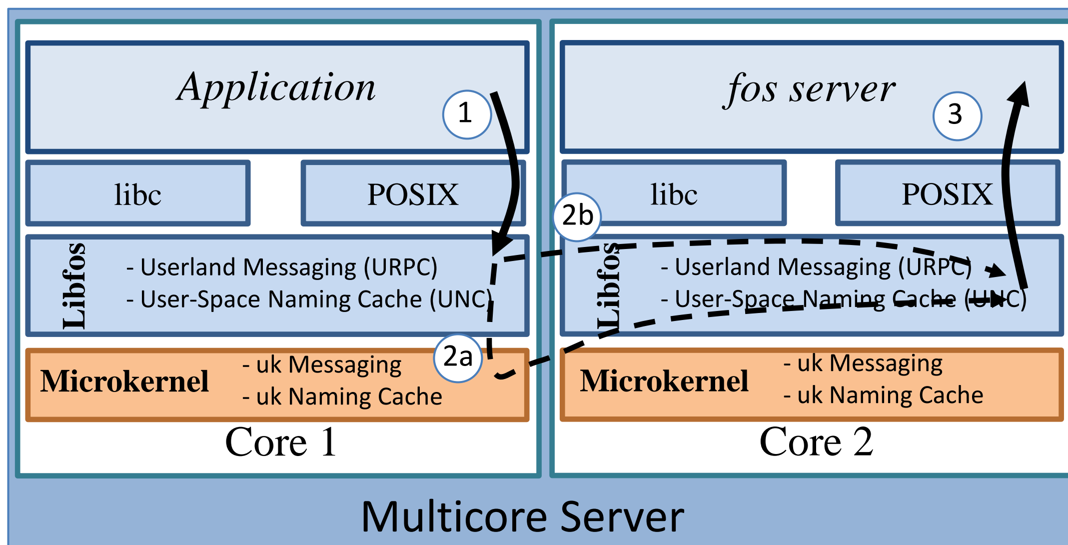

Figure 2-2: fos's messaging system. This figure shows the paths followed by a message sent within a multicore. Two paths are possible, either through the microkernel or through an optimized user-space channel.

As shown in Figure 6-4, fos currently supports two implementations of the messaging system within a multicore. These delivery mechanisms are transparently multiplexed behind the messaging API, so an application need not concern itself with the physical placement of services with which it communicates. For performance reasons, however, this may be important. Therefore, a key focus of fos is proper allocation of

[2]For performance, the system actually uses hashes of the textual name, termed an *alias*.

cores to OS services and placement of services to minimize communication costs.

The default delivery mechanism uses the microkernel to deliver messages from the sender directly into the receiver's mailbox. This mechanism traps into the microkernel, which checks the capabilities by reading from the destination's address space, and delivers the message by copying the data into the mailbox's receive buffer.

For shared memory machines, an optimized path exists that avoids the overhead of trapping into the microkernel. In this mechanism, a page is shared between the sending and receiving processes, called a *channel*. This page is treated as a circular queue, and messages are sent by copying data into the page. Messages are received by copying data out of the queue into a buffer. The second copy is necessary to free space for further messages and guarantee that the message isn't modified by the sender after delivery. This mechanism requires a channel to be established by the microkernel. Upon creation, appropriate capability checks are performed. If the capabilities of the mailbox are modified, then all channels must be unmapped to give proper behavior. Although creating the channel is a fairly expensive operation, the steady-state performance of messaging is greatly improved.

A third delivery mechanism exists for multi-machine fos systems. This uses a separate proxy service and network connections (TCP/IP) to deliver messages between machines. By bridging messaging across machines, fos can transparently span multiple machines under a single OS.[3] This aspect of fos is not the focus of this thesis, however, but is discussed further in [27].

2.3 Fleets

An OS service in fos is constructed as a fleet: a cooperative set of processes which, in aggregate, provide a single service. For example, within a single fos system, there will be a name service fleet, a page allocation fleet, a file system fleet, etc.. Because of the message-passing architecture of fos, services are effectively implemented as distributed systems. Fleets are the main contribution of fos over previous microkernel operating

[3]So long as applications use messaging and do not rely upon shared memory.

system designs, and we believe they are the key to scaling services to hundreds or thousands of cores. These ideas are discussed at length in [26], but this section briefly discuss the main ideas. This discussion describes the high-level goals, and the next chapter demonstrates how the name service enables each concept in this section.

In order for this design to function well in a multicore environment, the right services must be available when and where they are needed. There must be mechanisms to change fleet size dynamically to meet changing demand in the system. In fos, this is termed *elasticity*. Furthermore, once cores are divided among fleets, they must be spatially scheduled in the best manner to minimize communication overheads.

Because of the scale of future multicores and the cloud, it is necessary to parallelize within each OS service; factoring the OS into separate services alone is insufficient. Many important workloads in the cloud (*e.g.*, `memcached`, Apache) spend the majority of their time in the OS even on present multicores [7]. As core count increases, the demand on the service will increase proportionally.

Therefore fleets are designed with scalability foremost in mind. Services must scale from relatively few cores to hundreds or thousands. This is necessary in order to span differently-sized multicores and the cloud, where essentially arbitrarily large amounts of resources can be under management. As such, all services are designed from square one to be parallel as well as scalable. This often leads to fundamentally different design decisions than a simple sequential approach to the service.

In order to simplify handling simultaneous transactions, fleets are written using a light-weight cooperative threading model. This threading model provides a dispatcher that integrates with the messaging system to provide threads for each transaction processed by the service, and the ability to sleep until a response message arrives for remote procedure call (RPC) semantics. This model also enables lock-less design, as unlike kernel-level threading, threads are never preempted by other threads. fos also provides an RPC generation tool that parses standard C header files and generates marshalling routines to enable function-oriented code. This model allows non-conflicting transactions to be pipelined in the service to maximize utilization.

Fleets are also *self-aware*, meaning they monitor their own behavior and the envi-

ronment, and adapt their behavior accordingly. Although some external actions will control behavior, such as the allocation of cores to the service and the spatial layout of processes, fleets have many actions they can perform internally to adapt behavior. For example, active transactions can be migrated to under-utilized members or members that would have lesser communication cost. Additionally, if the scheduler has allocated a fleet fewer cores than it has members (*i.e.*, some members are being time-multiplexed on a single core), the fleet can shrink its size in order to avoid context switches.

2.3.1 Elasticity

In addition to handling peak demand, OS services must scale to match changing demand. Demand on the OS is rarely constant. Changes are due to phases in application behavior, new applications being run, or changes in dynamic workload (*e.g.*a web server).

Fleets are elastic, meaning they grow and shrink in order to match demand in the environment. This requires support from fleets themselves in order to add and remove members dynamically, including migrating state of active transactions.

Elasticity is closely related to resource allocation, specifically the allocation of cores to services and applications. A scheduling service makes decisions based on performance metrics (service utilization, message patterns, application heartbeats [16], etc.) on how to allocate cores in the system. Fleets then grow or shrink to meet their allocation.

2.3.2 Scheduling

Closely related to elasticity is spatial scheduling. The scheduling service in fos decides the spatial layout of processes in order to maximize utilization and minimize communication costs. Due to the increasing heterogeneity in communication costs within a multicore and the disparity of intra- and inter-machine communication, placing processes in a way that maximizes locality is important to performance. Early results

have shown that a modest heterogeneity of 48% can lead to end-to-end performance difference of 20% for some OS benchmarks [26]. Future multicores and the cloud have much higher heterogeneity, and therefore good scheduling is extremely important.

The scheduling problem in fos relates to spatial layout of processes, but fos *does* support basic time multiplexing. The microkernel on each core provides a simple process scheduler. This is useful for placing low-utilization, non-performance-critical services on a single core to maximize core availability. It is also much less expensive to migrate processes onto a single core than to transfer state between processes. Therefore, a "cheap" way to scale fleet size is to change the number of cores allocated to the service (*i.e.*, forcing time multiplexing) without changing the number of processes in the fleet. Depending on the granularity with which layout is updated, this could be an important tool to prevent thrashing fleet size while still controlling the layout effectively. It is also possible that some services will require a member to stay active (say, because of a pending transaction) and will not be able to shrink on demand.

Scheduling can be divided into two (possibly interdependent) problems: allocation and layout. The allocation is problem is *how many cores to allocate each service/application*. The layout problem is *how to place processes in the system to optimize performance, power, etc.*. This is currently an active area of research in fos.

2.4 Implementation

An implementation of fos is under development and currently boots on x86 multicores. The implementation runs as a paravirtualized OS under Xen [4] in order to facilitate its goals as a cloud OS, as the cloud management platforms Eucalyptus [20] and Amazon EC2 [13] only support Xen virtual machine images. Running under Xen has the additional advantage of limiting the need to develop hardware drivers, which is a challenge for any research OS.

Several services have been implemented: network stack, file system, process management, page allocator, proxy service, etc.. Of these, the network stack, name service, and page allocator are implemented as proper fleets that demonstrate good

scalability. A read-only file system fleet has been implemented to prototype ideas in fleet design and gather initial numbers.

The current implementation contains a port of libc from OpenBSD, with support for file descriptors, socket interface, etc.. Some full-featured applications have been run, in particular a video transcoder that uses `ffmpeg` and exercises the file system and network stack. The present limiting factor on applications is support for kernel-level multi-threading, on which many applications rely. This is on the road map to be implemented in the near future.

Chapter 3

Motivation for a Name Service

This section discusses the name service and demonstrates the foundational role it plays in the fos ecosystem.

Messaging is the main communication medium in fos, and the name service plays a crucial role in the messaging system. For this reason alone, the name service is a central component of fos. The name service finds the closest fleet member for a given service when the service is requested. It does this by routing messages from their symbolic name to a destination mailbox. Using a level of indirection for mailboxes provides an important convenience to users of the messaging system. This allows the high-level purpose of the mailbox (*i.e.*, the associated service) to be exposed and the location to reflect the dynamic placement of the mailbox.

The name service plays many other roles in fos that heighten its importance. It provides a mechanism for registering fleets in the name space, whereby multiple fleet members can be registered under a single name. To take a simple example, a file system fleet with two members could register the names `/sys/fs/1` and `/sys/fs/2` for the individual members, and each of these names would be registered under `/sys/fs`, the "global" file system name. When a message is sent to `/sys/fs`, the name service will return one of the fleet members, say `/sys/fs/1`, and the message will be sent accordingly (see example in Figure 1-1). Mappings from one name to another (*e.g.*, `/sys/fs` to `/sys/fs/1`) are termed *indirect*, as the name mapping resolves to another name rather than a mailbox. Likewise, name-to-mailbox mappings are termed *direct*.

This functionality is important to disguise from applications the precise number of servers active in a fleet. This allows for transparent growing and shrinking of fleets, enabling elasticity.

In servicing requests to these names, the name service effectively load balances the fleet. The name service *implements* load balancing policy, but it does not *determine* policy. Load balancing policy is specified by a service providing a mailbox which responds to requests for load balancing information. Alternately, the name service provides a default load balancing policy that favors names with lower communication cost.

This should not be taken to claim that indirect aliases offer a completely generic load balancing framework. Later in this thesis, I discuss instances where this policy must be circumvented, and fos's approach to these problems. However, in many cases this simple form of load balancing is sufficient, and furthermore gives equivalent performance to static mapping that is aware of each fleet member [26]. At worst, name service load balancing provides a good way to balance the initial request to a service among members before the final load balancing decision takes place.

The name service also plays an important role in scheduling (layout). The primary mechanism for scheduling is process migration, whereby processes are migrated to minimize global communication cost. This can result in changing the location of a mailbox, particularly when processes are migrated between machines.[1] Consequently, the name mapping must change to reflect the new location. The name service provides explicit support for migration by allowing a name to become "inactive" during the period of migration, so that messages sent to the name do not result in a fatal error. The name service therefore plays a crucial role in scheduling, by providing the essential level of indirection to enable process migration.

The name service plays a key role in many areas of fos, including the main active research areas in general OS design: resource allocation (elasticity) and scheduling (spatial layout).

[1]When mailbox locations must change depends on the scheme used for mailbox locations. On shared memory machines, intra-machine migration need not change the mailbox location. More discussion follows later in this thesis.

Chapter 4

Design

This section discusses the design of the name service and messaging system as it relates to the name service. It begins with a high-level discussion of the messaging interface and the different expected usage scenarios. Next, the design of the messaging system is discussed imagining the name service as a black box. This section concludes with the internals of the name service, and in particular a detailed discussion of the distributed data store which stores the name space itself.

4.1 Application Interface

The main customer of the name service is the messaging system within fos, and some of the design choices in the name service affect the usage of the messaging system itself. The primary function of the messaging system is allowing applications and services to send and receive messages. This is done by a *libfos* component that is present in all processes. To send messages, *libfos* contacts the name service to resolve the destination mailbox. *libfos* must also contact the name service to register and unregister mailboxes, as well as a few other operations that are useful in particular cases.

Table 4.1 shows the relevant portion of the messaging API. The API includes routines which explicitly modify the name space by registering or unregistering mailboxes, as well as operations to read state from the name service and resolve mappings.

Routine	Description
`GetCanonicalAlias`	Returns a "canonical" alias for the mailbox. This is a meaningless name that is unique to the mailbox. If no canonical alias has been created, then one is requested from the name service. This is intended to avoid unnecessary clutter in the name space.
`AliasRegisterDirect`	Register a direct mapping (name-to-mailbox) with the name service. Requires direct access (pointer) the mailbox itself, as well as appropriate capabilities.
`AliasRegisterIndirect`	Register an indirect mapping (name-to-name) with the name service. Requires capabilities to modify the name space, but does *not* require direct access to a mailbox.
`ReserveNamespace`	Reserve a sub-domain of the global name space. See discussion on capabilities.

`UnregisterDirect`
`UnregisterIndirect`
`ReleaseNamespace`
	The inverse of the corresponding registration operations.

`AliasResolve`	Query the name service to resolve (dereference) a mapping and return the result. If the input name is a direct mapping, this will return the input parameter, but for indirect mapping this returns one of the names to which the name is mapped.
`Send`	Sends a message to a given name using a given capability. The messaging library implicitly queries the name service for the location of the referenced mailbox, and does appropriate checks on capabilities.

Table 4.1: Selected subset of the fos messaging API. These operations either explicitly or implicitly involve the name system.

Other operations implicitly involve the name service, for example sending a message, which performs an implicit lookup through the name service. Note that in the API, names are referred to as "aliases," because the messaging system actually deals with hashes of mailbox names.

The bulk of the operations in Table 4.1 involve modifications to the name space, which are guarded by capabilities. The name space itself is split into sub-domains, just as a UNIX file system is split into folders. For example, the name `/sys/fs/1` has sub-domains of `/sys/fs/1/*`, `/sys/fs/*`, `/sys/*`, and `/*`.[1] The last name is referred to as the *root* of the name space. In order to modify a portion of the name space,

[1] `/path/to/name/*` is the way that sub-domains are represented within fos.

one must have the capability to modify at least one of its sub-domains. Therefore a process in possession of the root capability has complete ownership of the name space. These operations are essentially forwarded to the name service directly, and bookkeeping is done within the mailbox structure itself if the operation succeeds (next section).

The operations that explicitly modify the name space are the registration and unregistration operations. These are used to register services in the name space, and there are two versions: `RegisterDirect` and `RegisterIndirect`. `RegisterDirect` creates a direct mapping, and is used to register individual fleet members or services that only have a single member. It requires that the process that owns the mailbox makes the request. This is done in order to guarantee permissions to create the mapping and to bootstrap the registration process.[2] `RegisterIndirect` registers an indirect mapping, and is used to register the global name for a service fleet. It is a versatile mechanism that can also be used to assist in building some services, as will be shown later. It does not require any special capabilities for the destination name. This does not introduce a security problem because capabilities are required to modify the name space (so an adversary can not modify sub-domains of the name space that it does not own), and the registration process does not create additional capabilities to send messages. To clarify, one must draw a distinction between capabilities that allow modifications to the name space and capabilities that allow one to send messages to a mailbox. `RegisterIndirect` does not create any message-sending capabilities, so it does not allow any communication to take place that could not happen regardless. Finally, `ReserveNamespace` creates a capability for a sub-domain of the name space and claims the sub-domain in the name service.

The remaining operation, `GetCanonicalAlias`, makes implicit modifications to the name space. `GetCanonicalAlias` provides an arbitrary, "canonical" name for the mailbox. This is useful for local mailboxes that are owned by *libfos* and used to receive responses from services. These mailboxes don't represent a meaningful

[2] An alternative would be to use the mailbox location to bootstrap, but the mailbox location structure is never exposed in the messaging API, and this approach would require additional capabilities.

service, so they don't need a meaningful name; they just need a way to be addressed by the messaging system so that a response can be sent. If the canonical alias has not been created, then the name service is queried to generate a new name. This name is cached in the mailbox structure and subsequent operations incur no cost.

The last two operations involve reading the name space, and it is expected (and indeed observed) that these constitute the majority of traffic to the name service. These operations are `AliasResolve` and `Send`. The purpose of `Send` and its relation to naming are rather obvious: in order to deliver a message, the messaging system must resolve the name to a mailbox. When the destination name is a direct mapping, then the messaging system simply resolves the name to its mailbox location and sends the message. When an indirect mapping is passed in, the messaging system will resolve the name to one of the mailboxes that it eventually points to, but there are no guarantees as to which mailbox the message will go to. In particular, there is no guarantee that the same mailbox will receive consecutive messages, nor that different mailboxes will receive consecutive messages. At a higher level, this means that `Send` may send requests to different servers, which can interrupt stateful transactions. Additionally, consecutive requests may go to the same server, so `Send` alone does not provide fine-grain load balancing. This behavior allows maximum flexibility in the name service and messaging system, but can cause problems for certain types of services.

In order to illustrate this issue, consider a file system. fos's current prototype file system implements a parallel read-only fleet. This fleet is parallelized in the most naive fashion possible — simply by spawning multiple copies of the service and registering them under a single name. There is no sharing of state between the fleet members, and therefore if a request to a file that is opened on one member arrives at a different member, then the request will fail. There must be some way to guarantee that subsequent messages arrive at the same fleet member. This is an example where name server load balancing, although desirable for the initial `open()` request, must be circumvented for the remainder of the transaction.

This is where `AliasResolve` comes in. This routine resolves (dereferences) a map-

ping so that the final, direct mapping can be cached by the application for subsequent message sends. The *libfos* file system component performs an `AliasResolve` on the global file system name, and caches the result for future requests with that file. In the example of the prototype file system, the cached result is refreshed only if no files are open, although this could be optimized to have separate caches for each open file.[3]

The name service can also be used to track distributed state and greatly simplify the implementation of some services. This is in some sense related to the prior example, but in this case the state is being cached within the name space itself. In this model, each transaction registers a new name to track which fleet member is responsible for it. The prime example of this behavior is the network stack service, which registers a new indirect mapping in the name space for each active connection (*e.g.*, `/sys/netstack/connections/12345678`). The final portion of the name is a unique identifier for the transaction — in this case, a hash of the source IP, source port, destination IP, and destination port. This mapping points to the mailbox of the fleet member who is responsible for the connection.

This behavior is desirable in order to simplify the implementation of the network interface service, the service which receives raw data packets from the wire. The network interface service is responsible for routing packets to the correct network stack, but is not part of the network stack fleet proper. Consequently, it does not immediately know which member of the network stack fleet was assigned a connection when the connection is established. It can, however, easily compute the name that is associated with the connection by inspecting the packet.

Prior to using the name service, the network interface service had to maintain a shared, distributed structure with the network stack fleet to track the active connections. This task alone greatly complicated the network interface service, especially since there was no way to predict which member of the network interface service would service the next packet from the wire. By moving the complexity of shared state into the name service, the network interface service can focus on its assigned

[3]The file system name cache is distinct from the messaging name cache, although both reside in *libfos*. The file system cache only deals with open files, and which file system server is responsible for each. It implemented completely independently of the messaging system.

task, and its implementation is greatly simplified.

These examples demonstrate two contrasting uses of the name service. The key difference is that in the file system, the originator of requests is a member of the fos ecosystem and can cache name service lookups directly, whereas the network interface service receives messages in an opaque manner from the wire. One can imagine the file system using the same method of registering names for each open file, however this is unnecessarily complex because *libfos* can cache the name lookup itself. Significantly, the method employed by the network interface service changes the expected usage of the name service to favor more registrations and unregistrations. There are likely as-yet-undiscovered opportunities to offload distributed state into the name service that may exacerbate this issue. For a service that is optimized for read-only workloads (as the name service currently is), this poses challenges. However, there are some promising directions of investigation to mitigate or eliminate this problem (Chapter 7).

4.2 Messaging

As far as naming is concerned, the messaging system consists of four parts: the mailbox structure, the name service external API, the user-space name cache within *libfos*, and the message delivery mechanisms. This section discusses the responsibilities assigned to each component in order to create a functional, high-performance messaging implementation.

4.2.1 Mailbox Structure

The mailbox structure is discussed first, as this structure is used by the remaining components and can be discussed in isolation. Mailboxes are generally created in order to provide a service, as in a file system registering its mailbox as `/sys/fs`. Mailboxes can also be created for internal fleet traffic, as the name server does to manage updates to the name space (discussed later in this chapter). Lastly, as mentioned in the discussion of `GetCanonicalAlias`, mailboxes can be created within *libfos* as

Operation	Description
`Lookup`	Read the information stored for a given name.
`Register`	Register a new name. Requires capabilities to modify the name space, information to be registered, an optional load balancer, and some flags. Returns a capability to unregister this name.
`Unregister`	Unregister a name. Requires capability for that name returned by `Register`.
`RegisterTemporary`	Allocate a meaningless, unique name and register the given destination under this name. Returns the name and associated capabilities.
`Reserve`	Reserve a sub-domain of the name space. Requires capabilities for a parent domain and returns a new capability for the sub-domain.

Table 4.2: The public interface of the name service.

essentially anonymous structures that are used to receive messages from services.

A new mailbox structure is instantiated for each mailbox created, and this structure is always registered with the microkernel. The mailbox structure contains buffers for different message delivery mechanisms and the location of the mailbox. In order to validate message sends, it also contains a list of valid capabilities for the mailbox and a list of valid names for the mailbox. The delivery buffers are specific to each of the intra-machine delivery mechanisms: microkernel and user-space channels. These fields are platform-specific, and on x86 they are implemented as shared memory regions. For microkernel messaging, the message queue is allocated upon mailbox creation and guarded by locks. An overview of user-space channels was given previously, and details are outside the scope of this thesis.

When a mailbox is created, it is registered with the microkernel. The microkernel keeps a table of all mailboxes in the system in order to validate message sends and channel creations. The microkernel fills in the location field when the mailbox is registered and inserts the mailbox into its table.

4.2.2 Name Service

In the context of this discussion, the name service is treated as a black box and plays a simple role. The public interface of the name service is given in Table 4.2. `Lookup` queries the name service for information that is stored for a given name. This is used by `Send` and `AliasResolve` to read the global name space. If `Lookup` succeeds, it returns a structure that contains: the type of mapping (direct or indirect), the destination mapped to (mailbox location or another name), and an expiration time for the result. The expiration time is included so that the user-space name cache will refresh the name at regular intervals. This is good practice in order to load balance more effectively and to preemptively detect stale mappings for infrequently-used names that may have changed. The expiration interval is set to be fairly large in order to incur minimal overhead.

`Register` writes a new name to the name space. This collapses the direct and indirect operations in the messaging API. This operation either registers a new name, if it has not been previously registered, or updates a previous registration, say in the case of adding a new destination to an indirect mapping. If the name has been registered for a direct mapping, then `Register` will fail. It requires capabilities for modifying the name space, and if a name is being updated then the capability for the name must be presented as well. For example, imagine one tries to register `/sys/fs` to point to `/sys/fs/2`, and it already points to `/sys/fs/1`. Then one must present capabilities for both `/sys/fs/*` (or a parent sub-domain) and `/sys/fs`.

`Register` also accepts an optional load balancer parameter. This parameter is a name and capability for a mailbox that will respond to queries for load balancing names. If this load balancer is not provided, then the name service defaults to a load balancing policy that serves names in approximate inverse proportion to the communication costs.[4]

`Unregister` is the inverse operation of `Register` and deletes a specific name-to-destination mapping. In the case of an indirect mapping with multiple destinations,

[4]Some heuristics are used to avoid crossing discrete jumps in communication cost, so that local fleet members are favored.

a single `Unregister` operation will not delete the name entirely.

`Register` and `Unregister` also accept flags that can activate or de-activate an existing registration. This is useful during process migration to temporarily disable a registration without causing fatal errors in the messaging system, as the name will re-appear eventually and the `Send` should not fail.

`RegisterTemporary` is used in the registration of canonical names, and simply registers a unique but meaningless name to a given destination and returns the name.

Finally, `Reserve` is the corresponding operation to `ReserveNamespace` in the messaging API, and simply claims a sub-domain of the name space. It is listed separately from `Register` because it requires far fewer parameters. `Unregister` is also responsible for removing sub-domain reservations.

This API is given presently for descriptive purposes in the discussion of the messaging system. The next section (Section 4.3) discusses the design behind this interface.

4.2.3 User-space Name Cache

Every fos process contains a user-space name cache in *libfos*. This cache is private to the process and accelerates message delivery by eliminating a round-trip delay to the name service in order to send messages. Some caching of name lookups is necessary for messaging performance, and having a per-process cache offers advantages over a global cache. A per-process (or per-thread) cache improves load balancing over a global cache by having multiple, different responses cached for a given name. It is also potentially faster than a global cache by keeping all data within the process's hardware cache. This comes at the cost at missing more frequently in the name cache, as names are not shared between processes. However, since the name cache does not need to evict names due to capacity or conflicts, the only misses occur because invalid entries or names that have never been accessed. Because most applications have a well-defined, limited set of names they access, this overhead is negligible.

The name cache stores responses to `Lookup`s from the name service. Additionally, the name cache stores whether any channels have been created for a particular name. This latter function necessarily occurs in the address space of the process, as the sole

purpose of user-space channels is to avoid overhead of trapping into the microkernel.

The name cache is filled on `Send` and `AliasResolve` as needed, or whose entries have expired. When a `Send` occurs on an indirect mapping, then multiple `Lookups` to the name service may be required to resolve the name to a final mailbox location.

The main design principle is that the name cache only pulls from the name service. Stated another way, the name service *never* pushes updates or invalidations to the name cache. This greatly simplifies and optimizes the implementation of the name service, as it limits the distributed state under management by the name service to the name service fleet itself.

The approach taken by the messaging system is to optimistically assume that all cache entries are valid. What this requires for correctness is that the message delivery mechanisms provide error detection and recovery mechanisms. The delivery mechanism must detect when a cache entry becomes invalid and, if possible, successfully deliver the message.

4.2.4 Message Delivery

This section discusses the error detection and recovery for microkernel and user-space message delivery. Error detection is tailored to each delivery mechanism, but each mechanism share common themes. The basic idea is that the system must guarantee whenever a message is sent that (i) the destination mailbox is registered under the name to which the message was sent, and (ii) the capability is valid for the mailbox. Error recovery is provided by the messaging library in a delivery-agnostic manner. Complexities arise from the distributed nature of the name caches and the semantics of indirect mappings. To illustrate this, consider the following scenarios:

An application wishes to make a request to the file system. It has previously interacted with the file system, and at that time `/sys/fs` resolved to `/sys/fs/2`, and this result is cached by the application in *libfos*. However, the fleet member registered as `/sys/fs/2` still exists but has unregistered the indirect mapping, so that in the global name space `/sys/fs` no longer maps to `/sys/fs/2`. When the application goes to make a request to `/sys/fs`, it will resolve in its private name

cache to `/sys/fs/2`. When the message is sent, then `/sys/fs/2` points to a valid mailbox which *is* registered under this name, and the application has a valid capability for said mailbox, so the delivery succeeds. This is an error, because `/sys/fs` is no longer a valid name for the mailbox, and probably with good cause (fault detected, member is over-saturated, it is being migrated, etc.).

Another scenario involves user-space channels. Suppose an application sends messages to a mailbox and a user-space channel is created between the application and this mailbox. The channel is cached in the name cache of the application under the name of the mailbox. At some point, the owner of the mailbox invalidates the capability that the application is using to send messages. Without error detection, the application can continue to deliver messages to the mailbox, even though this has circumvented the security mechanisms.

These are examples of invalid operations that could plausibly succeed in implementations of the messaging system (indeed, they would have succeeded in previous incarnations of fos messaging). The former represents a semantic error, and the latter a security flaw. The former is not a security flaw, because as the mailbox is still registered as `/sys/fs/2`, an application *could* send to this name if they so desired. The solution in the first case is that the messaging system in the application must state its destination as `/sys/fs`, not `/sys/fs/2`, and the delivery mechanism must check that this is a valid name for the mailbox. The solution to the latter is that either capabilities must be checked when the message is received (the sender does not have reasonable access to the valid capabilities), or the channel must be destroyed when the capability is invalidated.

These solutions are the reasons that the mailbox structure contains a list of valid names and capabilities. Because the microkernel contains a table of all mailboxes, it has the wherewithal to validate names and capabilities, and it provides the routines to do so. Next, I discuss the specific error detection schemes for each delivery mechanism.

Microkernel. The interface to microkernel messaging is a single system call, `UkSend`. This function takes the destination name, destination mailbox location, capability,

and the data to send. This system call performs all error checks by confirming that the name and capability are valid for the mailbox at the given location. The messaging library must pass the original, intended destination as the destination name. That is, in the first example above, the messaging library must call `UkSend` with `/sys/fs` as the destination, not `/sys/fs/2`.

User-space. user-space messaging performs name and capability checks upon channel creation. Channel creation occurs through system calls, and these routines perform the same checks as does `UkSend`. When a capability or name is removed from a mailbox, then another system call allows a channel to be shot down. This guarantees that any time a channel is active, the name and capability it was created under are valid for the mailbox. This addresses the second example, because the channel is destroyed when the capability is removed. Finally, when a channel is created it must be stored in the name cache with the intended destination (*e.g.*, `/sys/fs`), not the resolved mailbox (*e.g.*, `/sys/fs/2`).

These methods provide error detection. The messaging library provides error recovery by refreshing the name cache via the name service. If the `Lookup` fails, then the name has been unregistered and the `Send` fails. Otherwise, the name cache is updated, and another send is attempted. If this second attempt fails, then the `Send` fails because subsequent refreshes of the name cache will result in the same behavior.

4.3 Name Service

The name service is divided into two components: the logic of the naming system, and the distributed data store that manages the global name table. The distributed data store consumed the bulk of the design effort, and for that reason merits its own section. This section discusses the design of the naming system logic, and the support required from the microkernel. In addition to the API given in Table 4.2, the name service supports management routines that are given in Table 4.3.

These four operations manage the name service fleet and are hopefully self-explanatory.

Operation	Description
`Start`	Starts the name service by spawning a name server. Only succeeds if it has not already been called. Returns the capability for root (`/*`) of the name space and a capability for protected operations within the name service.
`Stop`	Stops the name service. Requires the capability returned from `Start`.
`GrowFleet`	Increases the size of the name service fleet. Requires the capability returned from `Start`.
`ShrinkFleet`	Decreases the size of the name service fleet (but not below a single member). Requires the capability returned from `Start`.

Table 4.3: The management interface for the name service.

`Start` initializes the name service and returns capabilities for the root of the name space and a capability that protects other management tasks. `Start` will only succeed the first time it is called, and therefore it (and the other management operations) are not used by *libfos* within a typical application. Rather, `Start` is called at system boot as the first service started because messaging forms the foundation for all services in fos. Similarly, `Stop` is called on system shutdown. `GrowFleet` and `ShrinkFleet` are currently called in an ad hoc manner within the name service or in stress tests, but ultimately these will be called by a separate elasticity service that determines core allocation.

The name service is implemented in a completely distributed manner. No member of the name service is distinguished in any way, except for initialization that is performed by the first member spawned. In order to implement the API used by the messaging system (Table 4.2), the name service interacts with a distributed data store that contains the global name space. Its interface is given in Table 4.4. The data store runs as an independent entity within the name service, and registers its own mailbox to handle updates. Updates are processed in separate threads, as the name service runs on top of the dispatcher.

All of the operations in Table 4.4 accept a name (key) and an entry (value). For `Insert` and `Update`, the value is the new data that should be in the store. For `Delete`, the value is required to confirm the version of the data that the name service was

Operation	Description
`Insert`	Inserts a new entry into the data store.
`Update`	Updates an existing entry with new data.
`Delete`	Deletes an entry from the data store.
`Get`	Retrieves data from the data store.

Table 4.4: Interface to the distributed data store that manages the global name space.

operating on. `Get` accepts a pointer to a value structure to fill in. Entries in the data store contain: the entry type (direct, indirect, or a reservation[5]); the capability that protects modifications to the entry; and lastly, the destination mapped to the alias (a mailbox location, or a list of names).

Next, the implementation of operations in Table 4.2 are discussed, including the support provided by the microkernel.

`Lookup.` This is the simplest of the operations. It performs a `Get` on the data store, and checks the type of the returned entry. If the entry is a direct mapping, then it returns the mailbox location; if it is an indirect mapping, it randomly selects one of the destination names and returns it; if it is a reservation then the name service treats it as though the entry was not present (messages can't be sent to sub-domains). For indirect aliases, random selection occurs using weights specified by the load balancer (if present for this name) or a default scheme that favors local names.

All of the remaining operations modify the name space in some fashion. Because of the distributed nature of the data store, it can return an error code that signifies an inconsistent state. In these cases, the name service must restart the transaction. This is accomplished by rewinding all changes in the current operation and recursing.

`Register.` Registration modifies the name space by adding new entries or modifying existing ones. This corresponds to `Insert` and `Update` operations on the data store. The registration process begins by checking the data store (via `Get`) to see if the name exists. If so, then it checks the type of the entry and flags that were passed.

[5]For reservations of sub-domains such as `/sys/*`.

Operation	Description
`UkAddAlias`	Adds a name to a mailbox. Takes the location of the mailbox to modify, as well as the name to add. This is protected by a microkernel capability (different from previously discussed capabilities).
`UkRemoveAlias`	Removes a name from a mailbox. Takes same parameters as `UkAddAlias`.

Table 4.5: Microkernel support for indirect mappings.

For direct mappings, `Register` can update the entry by re-activating it if appropriate flags are passed. For indirect mappings, `Register` updates the entry by adding a new destination name. Otherwise, a new entry is created and inserted into the data store.

In order to allow for proper error detection, the name service must update a mailbox when a new name is mapped for it. For example, if `/sys/fs` is registered to point to `/sys/fs/2`, then `/sys/fs/2`'s mailbox must add `/sys/fs` as a valid name. This is complicated by the fact that the mailbox does not lie in the address space of the name service, and potentially not in the address space of the requesting process either. So whereas direct mappings can simply add the name to the mailbox structure in *libfos* of the requesting process, indirect mappings require another mechanism.

This support is provided by the microkernel, in an API given in Table 4.5. These system calls provide direct access to the name list for every mailbox. For security, these operations are protected by a unique capability that is passed to the name service when it registers with the microkernel (see Chapter 5).

I should briefly note that with indirect mappings it is possible to form arbitrary mapping graphs[6], and in particular one can form cycles. This can not be allowed, and the name service must ensure that the name space is always a directed acyclic graph. A complete implementation can check this using standard graph algorithms, *e.g.*breadth-first search, to avoid cycles. A similar procedure must be followed to add names to each mailbox that is affected by a registration, and likewise for unregistration. However, the current implementation avoids this issue by placing a restriction on indirect mappings that they must always point to a direct mapping (*i.e.*, the name

[6]Treating names as vertices and indirect mappings as edges.

graph contains two levels only). This restriction has so far posed no limitations on the use of the naming system, and it can be removed as necessary.

The remaining operations are straightforward.

`RegisterTemporary.` This operation is built atop `Register`, and simply loops through a set of meaningless names (currently in the `/tmp/*` sub-domain) until one is successfully registered.

`Reserve.` This is a simpler form of `Register` that deals with sub-domain registrations only. It is a separate operation because its input parameters are few and its logic much simpler: it attempts to register the sub-domain as a new entry, and fails if the name is already registered.

`Unregister.` The discussion of `Register` applies equally here, except instead of inserting new entries into the data store, `Unregister` deletes them. `Unregister` is the inverse operation of both `Register` and `Reserve`.

4.4 Distributed State

The distributed data store is the most complicated component of the name service, and it consumed the bulk of the design and implementation effort. The interface was given in Table 4.4, and additional management operations are given in Table 4.6. Except for bootstrapping issues which will be discussed at length in the next chapter, this data store is generic and could be used to store any key-value mapping. This section discusses the design of the data store, starting with the framework for reaching consensus, with a specific operation given as an example, and concluding with management operations.

The expected usage of the name service heavily favors reads. It is difficult to make precise statements, as fos cannot currently run a full suite of large-scale applications, but intuitively it is clear that most applications and services will register relatively

Operation	Description
`Init` `Register`	Initialization operations. Initialization is separated into two operations for bootstrapping reasons.
`Shutdown` `Unregister`	Inverse of initialization operations.
`ReserveIndex`	Allocate the next available index for a member of the name service fleet.
`Join`	An initialization routine that instructs the data store to query an active fleet member and "join" the distributed data store. Takes the name and capability to message in order to join.
`Leave`	The inverse operation to `Join`.

Table 4.6: Management interface to the distributed data store.

few names during initialization, and run in steady state with rare updates. As previously discussed, the network service is a possible exception, but even in this case many messages will be sent using a given name (*i.e.*, TCP connection) before it is unregistered.

Therefore the data store was designed as a fully replicated data structure, with a consensus protocol used to arbitrate modifications. This design optimizes `Get` operations, as they can always proceed locally and guarantee that the latest copy of entry is available. Modifications are expensive, however, and require responses from every fleet member to complete.

The challenge was developing the general framework for reaching consensus on decisions. The unique parts of each operation are relatively uninteresting, as each ultimately amount to performing an operation on a local hash table. Therefore this discussion focuses on the general ideas used in the design, and the later detailed example will show how a particular operation is performed.

4.4.1 Programming Model

In order to contextualize the discussion, the programming model used for fos services must be discussed in greater detail. fos services are written on top of a dispatcher, which is a cooperative thread scheduler that spawns new threads to handle incoming requests. Threads can sleep themselves and wait for responses to arrive, which are

identified by unique tokens. This means that a single thread of execution has full, uncontended control of the service, but when it makes requests to the other services, it will sleep for an unspecified time and will return to a system with modified state. The threading library also provides condition variables that can be used to sleep and wake threads, bypassing the dispatcher.

The dispatcher allows multiple mailboxes to be registered, with a static priority scheme for each mailbox. This allows one to implement different classes of requests, but more importantly for present discussion, it allows libraries to register their own mailboxes without conflicting message types with the application.

Within the name service, the data store uses this feature to register an internal mailbox. All updates to the data store go to these internal mailboxes, and “data store threads” are spawned which compete with “name service threads”. This allows all distributed complexity to be pushed into the data store, and the name service works as though accessing a local hash table.[7]

Another important issue when writing a distributed system is the guarantees provided by the messaging system with respect to delivery, re-ordering, and so on. fos guarantees successful message delivery when `Send` returns a success code, so this is a non-issue. The messaging system also guarantees *single-stream* ordering, but there are no guarantees on global ordering. A stream is a sending process (*i.e.*, an application) and a destination mailbox (*i.e.*, a fleet member). Therefore, requests to a service from a single process will not be re-ordered, but collaboration between multiple servers may be sensitive to re-ordering. This can simplify the logic of services and isolate complications arising from re-ordering to the distributed data structures. Indeed, this is the case within the name service.

[7]In prototyping this service, a sequential, non-distributed version of the data store was implemented. The code implementing the main name service API was unchanged when moving to the distributed data store. (Some name service management routines were modified.)

```
1  err = requestCommitLocally(in_entry)
2
3  if err != SUCCESS:
4      return BAD_OPERATION_ERROR
5
6  broadcastRequest()
7
8  err = collectResponsesAndCommit()
9
10 if err == SUCCESS:
11     commitLocally(in_entry)
12 else:
13     rollbackLocally(in_entry)
14
15 return SUCCESS
```

(a) Local

```
1  err = requestCommitLocally(in_entry)
2
3  if err == SUCCESS:
4     sendResponse(SUCCESS)
5  else:
6     sendResponse(CONFLICT_ERROR)
7     return
8
9  err = waitForConfirmation()
10
11 if err == SUCCESS:
12    commitLocally(in_entry)
13 else:
14    rollbackLocally(in_entry)
```

(b) Remote

Figure 4-1: Two-phase commit code flow for a modification to the data store.

4.4.2 Code Flow

The following discussion gradually builds the framework for making modifications to the data store. In each listing, the code is divided into *local* and *remote* portions, where the local portion is the code that initiates the request and the remote portion is the code that handles the update on other fleet members. The code listings assume that each routine takes two parameters, `in_alias` and `in_entry`. Modifications to the local table are made by passing `in_entry`, and `in_alias` is used to reference which name is being changed.

Conflict detection. The distributed data store uses a two-phase commit protocol to detect conflicts. Figure 4-1 shows the pseudocode for this version. The idea of this protocol is to split committing a transaction into two phases: request and complete. The request phase checks that no other transactions are pending, and marks the entry as pending. Completing a transaction involves updating the local table and removing the pending marker. In order for a transaction to finish, it must receive confirmation from all other fleet members.[8]

In this version, the local code requests a commit (lines 1-4), and aborts if it fails.

[8]This can be optimized to wait for fewer responses, but it is asymptotically equivalent and complexities arise from needing to terminate the threads that are spawned to handle the initial request.

Then the request is broadcasted as before (line 6), and the local thread goes to sleep until all other fleet members have responded (line 8).

Control is now transferred to the remote handlers. They being by attempting the same reservation as the local side, and if this succeeds then they send a message to the requester indicating so (lines 1-4). If this request fails, then the thread sends a error code indicating conflict and terminates the handler (lines 5-7). At this point, the remote side goes to sleep waiting for a response (line 9) and control is returned to the local side.

The local side resumes when all responses have been collected (line 8). Before returning from `collectResponsesAndCommit`, this checks the responses and confirms that all were `SUCCESS`. If so, then the routine responds to each handler telling it to complete the transaction. Otherwise, it responds to those handlers that responded successfully (the others have terminated), telling them to abort the transaction. The transactions now complete in parallel. Either the transaction completes, if it is successful, or the request is rolled back, if unsuccessful.

This avoids inconsistency in the data store because one error response will be received, at a minimum, if there are conflicting transactions. In particular, the original requesters for the conflicting transactions will respond with an error, as they have already requested a transaction to the same entry (lines 6-7 remote). Furthermore, any conflicting operations must arise from simultaneous transactions, because the local code will terminate a request if another transaction is pending (lines 3-4 local).

Conflict resolution. Two-phase commit provides a way to avoid inconsistent states, but it does not provide a good way to guarantee progress. Because all operations are rolled back when a conflict is detected, they will likely conflict again in the near future.

One solution is random back-off, where each conflicting operation sleeps for a random period of time before restarting. This is a probabilistic method that will eventually lead to progress, but since updates can involve large communication cost (particularly in multi-machine instances) and there can be many fleet members, the

```
1  arbiter = getArbiter(in_name)
2
3  if arbiter.isPending():
4      return CONFLICT_ERROR
5  else:
6      arbiter.claim()
7
8  err = requestCommitLocally(in_entry)
9
10 if err != SUCCESS:
11     arbiter.clear()
12     return BAD_OPERATION_ERROR
13
14 broadcastRequest()
15
16 err = collectResponsesAndCommit()
17
18 if err == SUCCESS:
19     commitLocally(in_entry)
20 else:
21     if arbiter.isOwnedBy(me):
22         rollbackLocally(in_entry)
23
24 arbiter.clear()
25
26 return SUCCESS
```

(a) Local

```
1  arbiter = getArbiter(in_name)
2
3  if arbiter.hasPriority(in_originator):
4      arbiter.claim()
5  else:
6      sendResponse(CONFLICT_ERROR)
7      return
8
9  err = requestCommitLocally(in_entry)
10
11 if err != SUCCESS:
12     forceCommitLocally(in_entry)
13
14 sendResponse(SUCCESS)
15
16 err = waitForConfirmation()
17
18 if arbiter.isOwnedBy(in_originator):
19
20     if err == SUCCESS:
21         commitLocally(in_entry)
22     else:
23         rollbackLocally(in_entry)
24
25     arbiter.clear()
```

(b) Remote

Figure 4-2: Use of arbiters to protect modifications to an entry in the data store.

sleep time would have to be quite large to work well.

A better solution is to provide a conflict resolution scheme that guarantees one of the conflicting operations will complete. This is done via a distributed arbiter that controls access to entries. Each entry in the table is guarded by a different arbiter, and arbiters are distributed among fleet members so that one arbiter resides with each copy of the entry. In order for an operation to complete, all arbiters for that entry must agree that the operation has priority and should complete. Depending on the ordering of messages during conflicting operations, it is possible that a some of the arbiters will validate an operation that should not complete, but this is harmless so long as at least one arbiter aborts it.

In order to modify an entry, the originator of the operation must "own" the arbiter. A server owns an arbiter when the arbiter grants priority to that server's request, but ownership may be revoked before the operation completes if a subsequent request has higher priority. Ownership is granted via a `claim` operation. One can test if an oper-

ation is pending via `isPending`. In order to arbitrate among conflicting operations, the arbiter provides `hasPriority`, which takes the identifier for the originator of the operation and determines if that operation has priority over the pending one.

The idea is that the arbiters will reach a consensus on which pending request has priority, and that request will preempt the others even if they have already requested a commit. The requests that are aborted remotely will return an error code to the originator. This will lead to a consistent state, as the same request (and only that request) will complete on each node.

Pseudocode is given in Figure 4-2. As before, discussion begins with the local code. First, the arbiter for the entry is found and it is queried to see if any requests are pending. In order to maintain the invariant that at least one error code is returned, requests cannot be preempted on the local node. To illustrate the problem, consider this example: the data store has two active members, A and B. Assume that requests from B always have priority. A makes a request, and B confirms it. Now B begins a conflicting transaction. If B preempts the pending transaction, then it will succeed because B has priority. The request from A should fail, but A will return `SUCCESS` because it received a confirmation from B. This leads to a state where A's request returned `SUCCESS`, but it was actually preempted by B's request in the data store.

The problem arises because B has preempted a request that it already confirmed with another fleet member. The solution is that a request cannot be originated on a node that has a pending operation, because the pending operation cannot be preempted safely. The local code implements this by checking the arbiter to see if any requests are pending (lines 3-6). If so, then a conflict is detected and the operation aborts.

Next, local code proceeds as in the two-phase commit protocol to broadcast the request and collect responses (lines 8-14). The only change is that the arbiter is always cleared when the operation terminates (line 11).

Control transfers to the remote code, which begins by querying the appropriate arbiter to see if this request has priority (lines 1-7). `hasPriority` first checks if any requests are pending — if not, then the request automatically has priority. If there is a

pending request, then a simple scheme based on the identifiers (*i.e.*, mailbox names) of the request originators is used. Currently, the request with the lexigraphically lowest name is given priority. If the request is given priority, then it claims the arbiter and proceeds. Otherwise, it sends an error code response and terminates.

Next, the remote code attempts to request a commit locally (line 9). The difference between previous code listings is that if this request fails, then the pending request is preempted. Because the current operation still has priority, it must force the request (lines 11-12). In some cases nothing needs to be done — the details depend on the operation. The remote code sends `SUCCESS` to the originator and waits for a confirmation, as before (lines 14-16).

The local code completes in the same manner as before. Local code resumes when `collectResponsesAndCommit` returns, and if all responses were `SUCCESS` then it completes the operation. Otherwise, it has been preempted and it rolls back the request. However, it first checks if the arbiter is still owned by the thread to prevent interfering with the preempting operation.

When the remote threads resume, they also check that the arbiter is still owned by the originator of the request. In this case, the check must guard both completion and roll back. This is because there is no global ordering, it is possible for a later request to complete, and the earlier request should not overwrite the result. An example requires three members, say A, B, and C. Suppose A initiates a modification, and all members respond successfully. A and B complete the operation, but C does not receive the complete message (yet). B then initiates a subsequent modification and A and C confirm *and* complete the operation. This is possible because the messaging system does not enforce ordering of messages from different sources, so the messages from B arrive at C before those from A. Now, when C receives the complete message from A, it should not modify its local copy.

One last optimization maximizes useful work done by the name service. This is motivated by the observation that if a request is aborted, then it should not be immediately restarted because this will lead to another conflict and abortion. This is avoided in the arbiters using condition variables. Threads that do not have priority

```
1  arbiter = getArbiter(in_name)
2
3  if arbiter.isPending():
4      return CONFLICT_ERROR
5  else:
6      arbiter.claim()
7
8  # mark entry invalid
9  in_entry.version = -1
10
11 # request
12 err = localInsert(in_name, in_entry)
13
14 if err != SUCCESS:
15     arbiter.clear()
16     return BAD_OPERATION_ERROR
17
18 broadcastRequest()
19
20 err = collectResponsesAndCommit()
21
22 if err == SUCCESS:
23     # complete
24     in_entry.version = 0
25 else:
26     if arbiter.isOwnedBy(me):
27         # roll back
28         localDelete(in_name)
29
30 arbiter.clear()
31
32 return SUCCESS
```

(a) Local

```
1  arbiter = getArbiter(in_name)
2
3  if arbiter.hasPriority(in_originator):
4      arbiter.claim()
5  else:
6      sendResponse(CONFLICT_ERROR)
7      return
8
9  # mark entry invalid
10 in_entry.version = -1
11
12 # request
13 err = localInsert(in_name, in_entry)
14
15 if err != SUCCESS:
16     # force
17     localUpdate(in_name, in_entry)
18
19 sendResponse(SUCCESS)
20
21 err = waitForConfirmation()
22
23 if arbiter.isOwnedBy(in_originator):
24
25     if err == SUCCESS:
26         # complete
27         in_entry.version = 0
28     else:
29         # roll back
30         localDelete(in_name)
31
32     arbiter.clear()
```

(b) Remote

Figure 4-3: Code for performing a distributed `Insert` in the data store.

are put to sleep in `isPending` and `hasPriority`, and these operations only return when the arbiter is `clear`ed and becomes available.

4.4.3 Concrete Example

`Insert` provides a concrete example of a distributed modification operation. Figure 4-3 shows pseudocode for this operation, showing how each of the general request, force, or complete operations are implemented.

The data store keeps a monotonically-increasing version number associated with each entry. Each modification to the data store increments the version number. Previously, this was used to track conflicting operations, but the arbiters subsumed this responsibility and provide other benefits (conflict resolution, condition variables,

etc.). A negative version number indicates an invalid entry in the local table, and `Get` will treat such entries as though they were not present.

Thus, to request an insertion in the table, one inserts an entry with a negative version number (line 9 local, line 10 remote). Similarly, in order to complete an insertion, one sets the version number to zero (line 24 local, line 27 remote). Roll back is done by deleting the entry from the local table (line 28 local, line 30 remote). Finally, forcing an update (preempting a pending `Insert`) means overwriting the pending entry (line 17 remote).

It may seem that this method of forcing a request could overwrite a valid entry in the local table. But consider the state of the data store when a forced request takes place. This can only happen when an `Insert` is pending on the thread performing the forced request. If a prior `Insert` had successfully completed, then the originator of the pending `Insert` would have had to confirm it. This means that, at a minimum, an invalid entry would exist in that member's local table, and the pending `Insert` would fail locally (line 12 local).

4.4.4 Management

Management of the data store consists of initialization and shutdown of the data store, and changing the number of active members in the data store. The interface was given in Table 4.6. Initialization and shutdown are fairly uninteresting. Initialization empties the local hash table, creates the internal mailbox, registers this mailbox with the name service, and registers request handlers with the dispatcher. This involves bootstrapping that is the subject of the next chapter. Shutdown simply does the inverse. Consequently, the remainder of the section discusses managing the members of the data store.

The number of members in the data store are managed via `Join` and `Leave`. When a new member joins the data store, state must be transferred to the member. This involves downloading the contents of the data store from an existing member. A single member is designated to do this. Additionally, all existing members must be notified of the existence of the new member. `Leave` is simpler because the data store

Operation	Description
`modifyLock`	Grab the modify lock when a new modification begins. Increases the count of pending modifications. Will sleep while a thread holds the blocking lock.
`modifyRelease`	Release the modify lock when a modification completes. Decreases the count of pending modifications, and may grant the blocking lock.
`blockingLock`	Grab the blocking lock, which prevents anyone from grabbing the modify lock. Only returns when all pending modifications have completed.
`blockingRelease`	Release the blocking lock, waking all threads that are waiting on the modify lock.

Table 4.7: The interface to control modifications to the data while a `Join` is in progress.

is fully replicated on each member, so no state must be transferred. Instead, the member notifies its peers that it is leaving, and it can safely terminate.

`Join` complications arise when handling new or pending requests during the download process. It would be easy for an inconsistent state to be reached if modifications occurred while state was transferred. Therefore, the data store currently blocks all modifications during a `Join`.

This choice is not fundamental to the data store design, and could be optimized to allow modifications to take place during `Join`. However, it is expected that `Join`s will be rare and fairly short-lived. This decision appears to create no practical limitations in actual use of the name service.

Blocking is done by requiring all modifications to acquire a "modify lock" when they begin, and release this lock when they finish. The interface is given in Table 4.7. `Join` acquires this lock in a "blocking" mode when it wishes to begin state transfer. In order for `Join` to proceed, all pending modifications must complete. `blockingLock` sets a flag that blocks new modifications, but does not grant the lock until all pending modifications have completed. Likewise, only the local code portion acquires the modify lock, because remote portions must execute in order to complete pending transactions. Stated another way, local code represents new transactions, whereas remote code represents pending transactions that must complete. When a thread

attempts to acquire the modify lock when the blocking flag is set, it is put to sleep. When the blocking lock is released, these threads are woken, and normal operation resumes.

To summarize in concrete terms, `Join` changes the code flow from Figure 4-2 so that all modifications begin locally with `modifyLock()`. This can potentially put them to sleep if a `Join` is in progress until it completes. Local code calls `modifyRelease()` when it returns. Remote code is unchanged.

4.4.5 Summary

The data store implements a general-purpose, fully-replicated key-value store. It isolates the complexity of distribution to an encapsulated component of the name service. As a result, the code for the name service reads as though it were using a local table for storage.

One of fos's research goals is to explore programming models for distributed services and applications. We recognize distributed state as a major issue, and therefore we want to provide a library of distributed data structures. This library should isolate distributed complexity from the user and give the illusion of local access for distributed data. Although far from proof, the name service gives encouraging evidence that this goal can be achieved. However, this is but one example, and it remains to be shown that a general-purpose library can accommodate a variety of usage patterns with good performance.

Chapter 5

Implementation

The implementation of the distributed name service was a major effort that touched many areas of fos. It exposed issues in several support libraries and the messaging system itself. It is also the first service to require a distributed key-value store. Finally, because the name service plays a foundational role in the messaging system, it exposed numerous bootstrapping issues. This section discusses these themes and the experiences gained.

5.1 Scope

Before this effort began, naming was performed within the microkernel as part of the messaging system. There was no separate name service fleet or user-space name cache. This greatly simplified the implementation for a number of reasons: (i) the name space could be stored in a single shared-memory table within the microkernel, eliminating issues arising from distribution; (ii) all bootstrapping problems were avoided because the name service was accessed via system calls instead of messages; (iii) naming and messaging were contained in a single component, so it was possible to avoid some issues such as imperfect load balancing due to private name caches; and (iv) because of the lack of name caches throughout the system, there was no need to consider error detection or recovery for each delivery mechanism. The name service presented in this thesis began with a simple, straightforward design, but as each of the above

problems became apparent, the design expanded and became more complicated.

There were problems with the microkernel implementation. This implementation is highly architecture-specific and relies on global shared memory, something fos explicitly wants to avoid. One could argue that such a central task as naming could be optimized for the architecture, but it also posed real functional issues. In a cloud computer, there is no global shared memory. Therefore, the name space was fragmented among the microkernels on each machine. This was "solved" by having the proxy service, which sends messages between machines, maintain a routing table of which machine each name lived on. Then, if a name did not exist within the microkernel's name table, the message was forwarded to the proxy service for delivery.

This wasn't a real solution, however. It incurred performance overhead by attempting to send locally before sending to the proxy service, and created unnecessary traffic on the proxy service for `Send`s to invalid names. Furthermore, the proxy's routing table had no conflict detection or recovery mechanism. Most importantly, it violated the global consistency of the name space by creating a two-level hierarchy of names: those within the machine were preferentially served by the name service. fos claims to provide a single system image in the cloud, and this naming implementation did not meet this promise. For example, the network interface services on each machine were *all* registered as `/sys/netif` with a direct mapping, which is invalid. Thus, if a process migrated to a new machine, it would begin communicating with the wrong network interface service.

These problems motivated an improved implementation of the name service. In particular, we were beginning to discuss scheduling (layout) and needed to implement process migration. As the above example shows, the name service needed to be improved to allow this. Rather than patch the existing implementation, I decided to test fleet design concepts by implementing a proper name service fleet.

Most parts of this design were implemented from scratch. The microkernel messaging interface was reworked: mailbox locations were added to `Send`, and the majority of the messaging system calls were removed and handled in user-space through the name service fleet. Several new system calls were added, such as `UkAddAlias` and

those discussed in the next section. The microkernel mailbox table was written from scratch; in some ways, this replaced the shared-memory name table, but in a smaller, more well-defined role.

The messaging library was heavily modified in order to properly use the name service and provide necessary error detection and recovery. The name cache was written from scratch to implement naming outside of the microkernel and correctly interface with the name service. Another name cache existed under simultaneous development, which cached user-space message channels. These implementations were merged to provide the final name cache.

Other support libraries were significantly modified. In particular, the threading and dispatch libraries were modified to provide new APIs, improve existing implementation, or fix bugs. Numerous bugs in the RPC generation tool were found and corrected, and countless minor bugs were found in various places in the system.

Most significantly, the name service itself was written from scratch with very little shared code from the microkernel implementation. In summary, this thesis represents a large effort that affected many of the building blocks of fos.

5.2 Bootstrapping

Moving the name service out of the microkernel exposed numerous bootstrapping issues. For example, how can one communicate with the name service if messaging relies on the name service? How does a new process register its first mailbox (who can the name service respond to)? These problems interfere in many small ways with normal operation. This section contains a few examples, and I begin with one that was elegantly avoided by the RPC generation tool.

Name service on *libfos*. The name service was written using the tools and libraries for other services. This involved a dependency cycle between the messaging library and the name service, but it was necessary because re-implementing the libraries would be prohibitively difficult. The dependency cycle was that the messaging

library messages the name service to resolve names. Thus, when the name service sent a message, it would message itself to resolve the name, which resulted in another message to itself to resolve the name, and so on *ad nauseum*.

The cycle was broken using the RPC generation tool and C linkage semantics. This tool takes a library function, say `int foo(char * a)` and generates a remote procedure call. Most importantly for this discussion, the RPC has the same signature as the original function, say `int rpc_foo(char * a)`. Therefore the interface between the messaging library and the name service is actually a set of RPC routines. Most applications and services link with the library generated by the RPC tool that marshalls parameters and messages the name service. However, the name service links with a separate library that is a thin wrapper for the local routines. Thus, where most services would message the name service, the name service itself will call into its local routines. This lets the name service use the messaging library exactly as a regular application would.

Curiously, this means that even though the name service has a full copy of the name space in its local memory, it maintains its own name cache within the messaging library that may be inconsistent with the global name space.[1]

Messaging the name service. How does an application communicate with the name service? To illustrate the problem, consider sending a message with an empty name cache. First, the application will miss in its cache and attempt to message the name service to fill the entry. Then it will miss in its name cache when looking for `/sys/name` (the name service mailbox name), and enter into an infinite loop of refreshing its cache.

This dependency cycle is broken through a system call, `UkNameServiceLookup`. The microkernel maintains a list of the active name servers on the local machine, and each microkernel (*i.e.*, machine) in a fos system must have a name service registered. This system call takes no parameters and returns the mailbox location for the name service, bootstrapping the lookup process. There are corresponding operations

[1]This could avoided by using the same tricks to link in a custom name cache that accessed the global store directly, but currently the name cache is encapsulated within the messaging library.

that are performed during name service initialization, `UkNameServiceRegister` and `UkNameServiceUnregister`.

Curiously, this means that `/sys/name` is never actually registered in the distributed data store, as each name server is registered with the microkernel and lookups to `/sys/name` are redirected to `UkNameServiceLookup`.

Sending first message. In order for a process to send its first message, it must be able to perform a lookup in the name service because its name cache will be empty. The previous solution allows the process to message the name service, but how can the name service respond? In order to respond via the messaging library, the process must have a mailbox with a registered name. But registration requires a message to the name service, so we have a dependency cycle.

This dependency is broken by having the messaging library register an internal mailbox first, and by adding a system call, `UkUncheckedSend`. This system call uses microkernel messaging to send a message directly to a mailbox, bypassing the name and capability checks. This allows the name service to message a mailbox that does not have a name. This is obviously a security hole, so this system call is guarded by a special capability that is only given to the name service. The name service receives this capability when its first member registers, and it passes the capability to new members as they are spawned. Because the name service must be started first in any interesting instantiation of fos, this gives good security.

This problem also plays into process migration. Because `UkUncheckedSend` uses a mailbox location directly, and there is no error recovery scheme, a process cannot be migrated while its first registration is pending. Whether this creates any practical limitation on scheduling remains to be seen.

Starting the name service. Earlier it was mentioned that the distributed data store's initialization is split into two routines, `Initialize` and `Register`. This is because of a bootstrapping problem in starting the name service. Until the name service starts, obviously there is no name service running. In order for the name

service to start, it must initialize the distributed data store, and the distributed data store must to initialize its internal mailbox to receive updates. But the initialization of this mailbox involves registering it with the name service, and we have a dependency cycle.

This is solved by splitting initialization of the data store. `Initialize` empties the local hash table and readies it for entries, along with other tasks *except* for registering its internal mailbox. The name service then claims root `/*` and performs its remaining initialization, calling `Register` when the name service is operational.

Along a similar vein, when the name service grows by adding a member, this member must communicate with the existing members to join the fleet. Because the name service uses its local hash table to fill its name cache, it is essentially an isolated name space until it joins the main fleet. Therefore, it must be passed the name, capability, and location for the fleet member that will help it join the fleet so that it can message this member.

Although these problems may not seem disastrous, in fact these problems required significant re-workings of the design to solve. In the author's biased opinion, the current design solves each issue elegantly, but this was the result of several crises and not the "natural" design that was originally conceived.

5.3 Development Issues

The name service stresses the fos system in unprecedented ways, and is the first service sensitive to global message ordering. In terms of messages alone, every modification to the distributed data store requires communication to and from each fleet member, creating complicated message patterns and extensive traffic. Given this, it is no surprise that several unexpected problems were encountered during development. This section discusses a few issues to give a flavor for the development experience.

Bear in mind that when pursuing these bugs, limited tools were available. fos is a research OS, and full debugger support is not available as in Linux. To make matters

worse, these bugs are sensitive to timing, so using the console to debug would often perturb the test enough to avoid the bug, or expose a new failure mode. And on top of that, some issues would cause the machine to terminate abruptly, and output buffers would not be flushed. This would lead to misleading traces of program behavior and many wasted hours.

The traces themselves, even when completely accurate, were difficult to parse. Messages from a single server correspond to different threads processing different transactions, and multiple servers are active simultaneously. This created a very difficult debugging environment, and given the subtlety of some bugs, it was at times a nightmare.

Send deadlock. While running tests that would perform many simultaneous registrations, a pair of name service fleet members would suddenly "disappear" and cease responding to messages. This bug could be solved easily because fos has the capability to give a back trace of all active processes, which can be activated interactively by the user. Therefore deadlocks can usually be solved quickly, because one can tell where a process is stuck. (Of course, any debugger in a commercial OS could give the same information.)

The deadlock was occurring because the user-space messaging code would block if a message was partially sent and the channel became full. (If the channel was full before the message is sent, then it would return an error code, and the system behaved correctly.) If two name servers sent each other messages simultaneously, it was possible for them to block. The channels would never drain, and no progress would be made. This is a classic example of send deadlock in a system with arbitrary communication and finite buffers.

The solution to this problem was to return an error code if no space was available in the channel, but this will not work for large messages. A complete solution would send as much of the message as possible and return after a period of time if no progress is being made. Before returning, the channel would be marked as "claimed" so that other messages to do not interleave with the partially-sent message.

Single stream re-ordering. During similar tests, the distributed data store would often arrive at an inconsistent state. By examining several traces, it was determined that messages within a single stream had been re-ordered. An inspection of the messaging system provided no hints as to how this could occur. Eventually, it became clear that the dispatch library was responsible.

The problem was that the dispatch library deferred the processing of responses, but immediately handled new requests. Explanation of this behavior requires an understanding of the internals of the dispatch library. When no threads were runnable, the dispatch library spawned a new thread to process pending messages. If a response was received, then it was buffered and all waiting threads were signalled. The message-processing thread continued *without yielding to the woken threads* to process further messages. However, if a request was received, the message-processing thread called the appropriate handler *immediately*.

Thus, if a response and request were pending in the message queue, then the request would always be handled before the response. If the response arrived before the request, this violated the expected behavior, and led to an apparent message re-ordering.

Two solutions were possible: to defer handling of new requests, or to immediately handle responses. It was simpler to implement deferring of new requests, but this could quickly lead to an explosion of waiting threads if the service was over-saturated. It is better to immediately handle the message in order to apply appropriate back pressure through the system. This required additional support from the threading library, and significant modification of the dispatch library itself.

Global re-ordering. Message ordering guarantees were first explicitly discussed as a result of the name service fleet. Microkernel messaging guarantees global ordering in a single-machine fos instance, as all messages go to a single buffer that is guarded by a lock. User-space messaging is a relatively new addition, and previous services were insensitive to global ordering regardless, so the issue of re-ordering had not been problematic for any services. When stress testing the name service fleet with con-

tending registrations, global re-ordering was observed that led to inconsistent states in the data store.

Global re-ordering occurs when using user-space messaging. Each process sends messages on a separate channel, and the messaging library receives messages one channel at a time. This guarantees single-stream ordering, but messages can arrive re-ordered globally when the channels happen to be checked in the "wrong" order.

The solution to this required more liberal use of arbiters. Previously, arbiters had been used only when conflict was detected (using version numbers). It was possible that a valid modification could complete on some fleet members before a preceding modification, which led to inconsistencies when the preceding modification completed (see example in Section 4.4.2). In order to detect these kinds of behavior, arbiters were used to protect even uncontended operations.

5.4 Experiences

This thesis serves as an important proof-of-concept for fleet design. Previously, the microkernel had implemented naming via a global, shared-memory table. If fleets are indeed the correct approach for scalable OS services, then it seems inappropriate for the most basic service (*viz.*, naming) to be implemented in a traditional, monolithic fashion. Furthermore, implementing the name service as a fleet should not require herculean effort or result in significant performance degradation. As the results show (next chapter), the new naming and messaging system actually improved performance. The verdict is less clear in terms of development effort, however.

It is not necessarily a goal of fos to provide the easiest implementation experience compared with other OS designs,[2] but developing an OS service should not be prohibitively difficult. The name service fleet required several orders of magnitude more effort than the shared memory, "monolithic" implementation. This effort was largely spent in solving bootstrapping issues, implementing the distributed data store, and

[2]Although we *are* interested in making this experience as pleasant as possible through programming models and libraries.

painful debugging.

The bootstrapping issues are almost certainly unique to the name service, and one would not expect other services to encounter them. Although each service will have its share of complications, they will not need to bootstrap the underlying messaging system.

Every service will, however, need to manage its distributed state. The difficulty this presents is highly dependent on the service in question. Some services, *e.g.*page allocator, can split distributed state among members with modest sharing. Others, *e.g.*device drivers, may not be distributed at all. For still others, *e.g.*network stack, state can be owned by a single fleet member, but a distributed table is needed to lookup which member is the owner. And finally some will, like the name service, have highly shared state that requires global consistency. It is unclear what patterns will emerge in shared state. fos would like to provide a library of data structures that solves the common case, but more experience is needed to see if this vision will come to fruition.

Debugging is a challenge that all distributed services will need to solve. Hopefully the distributed data structure library will isolate issues such as message re-ordering, but when bugs inevitably occur, debugging will be required. Debugging of both distributed systems and research OSes is hard, and two wrongs definitely do not make a right. The good news is that because fos runs under Xen, tools are available to pause the entire machine and inspect its state. Unfortunately, these tools were not available during development of the name service. Furthermore, the most painful bugs encountered were those in the support libraries. These bugs are solved, so other development will not be affected.

This suggests that while development of the name service was difficult, future fleets will forge an easier path.

Chapter 6

Results

This section presents results for the name service fleet. This includes benchmarks for particular operations in the name service, and end-to-end performance numbers from messaging and file system benchmarks. Discussion begins with methodology and then covers results from each experiment in turn.

6.1 Methodology

These results were gathered from fos instances running as Xen 4.0 paravirtualized machines. Two implementations are compared: the implementation described in this thesis, and the prior microkernel, "monolithic" implementation. Both implementations contain the same basic messaging delivery mechanisms, but there are significant differences in the code base (Section 5.1). Instances were run on a 48-core machine with quad 12-core AMD 6168 processors and 64 GB of RAM. Due to Xen configuration issues, fos instances are currently limited to 32 cores.

Results show overall system throughput (requests / time) for different system configurations. The number of clients making requests to the service are shown on the x-axis, and system throughput is shown on the y-axis. Different plots are shown for each configuration, usually different fleet sizes.

6.2 Micro-benchmarks

These results examine the performance and scaling of individual operations of the name service. The name service was designed envisioning a read-mostly workload, and thus used a fully-replicated data store. These results reflect this decision, as `Lookup` has the best raw performance and scaling.

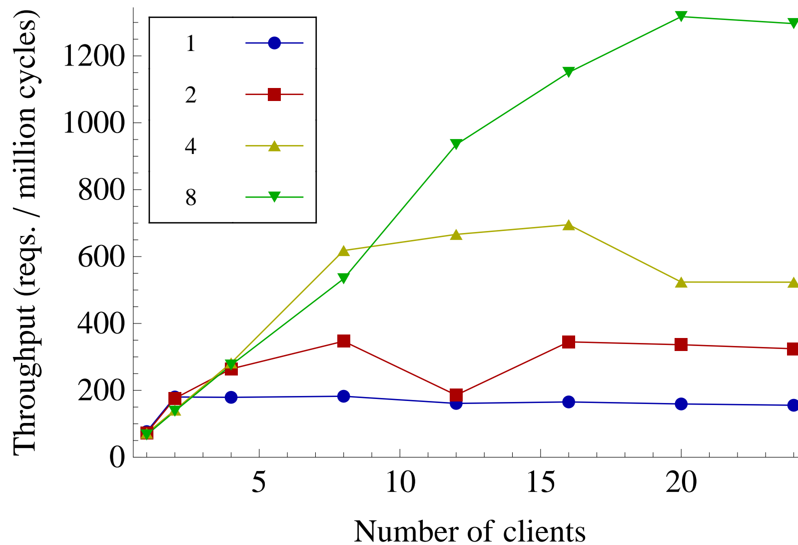

Figure 6-1: `Lookup` performance in the name service fleet. Legend indicates size of name service fleet. Performance scales well with number of clients until fleet saturates. Peak service rate scales well with fleet size.

Lookup. Figure 6-1 shows results for a `Lookup`-focused micro-benchmark. This benchmark spawns a name service fleet (size indicated in legend) and different numbers of clients. Each client performs `Lookup`s as quickly as possible. These `Lookup`s are done using the name service RPC routines directly to avoid the user-space name cache. As such, this load is *much* higher than the name service would experience in actual operation.

Results demonstrate excellent scaling of name service performance as the fleet size increases and the number of clients increases. For this benchmark, results are not shown for the microkernel naming implementation, because this has no notion of a `Lookup` separate from a `Send`.

Comparing these results to messaging results (below), it seems that `Lookup` performance follows the same trends. This is expected because `Lookup` always resolves locally, and therefore only involves one message round-trip. This indicates that `Lookup` performance is limited by messaging throughput.[1]

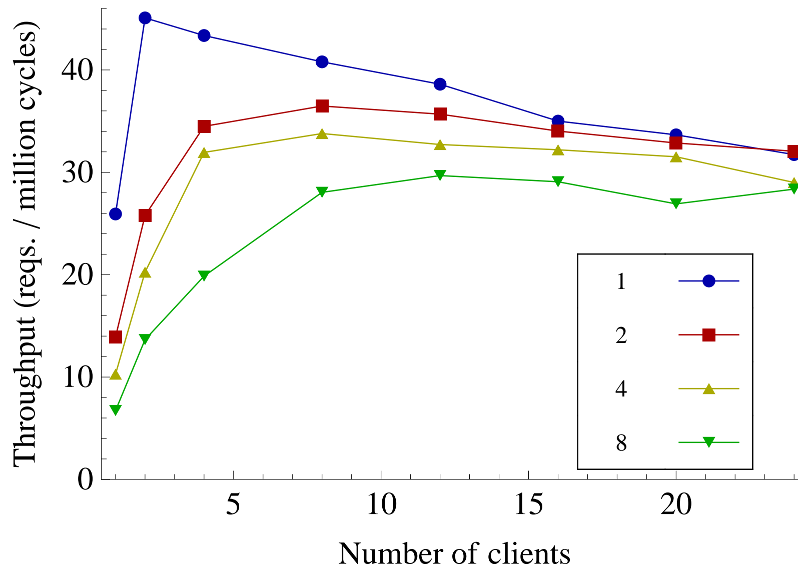

Figure 6-2: `Register` performance in the name service fleet. Legend indicates size of name service fleet. Performance converges for all fleet sizes as the number of clients increases. For few clients, larger fleets have worse performance. Throughput is far worse than for `Lookup`.

Register. Figure 6-2 shows results for a `Register`-focused micro-benchmark. This benchmark spawns a name service fleet (size indicated in legend) and different numbers of clients. Each client `Register`s and `Unregister`s a unique mailbox as quickly as possible. This test uses the messaging API to perform registration, so it includes overhead of updating the mailbox structure, etc.. Each client registers a separate name, so there is ample parallelism available.

Results show no scaling of throughput as the number of clients or fleet size increases. This is expected due to the implementation of the data store; each `Register` requires global consensus, so the work required to complete a registration scales with the size of the fleet. Likewise, the latency of `Register` increases as the fleet grows,

[1] The difference in absolute performance from messaging results is explained by the larger size of messages in a `Lookup`.

and this is indicated in the throughput with a single client, where performance is best for a single name server and decreases as servers are added. Throughput for all fleet sizes converges as the number of clients increases. This is because the service becomes fully saturated, and every fleet member is fully utilized. Work scales proportionally with fleet size, thus a fully saturated fleet maintains the same performance as a fleet with a single server.

Note that each request in this test corresponds to two updates to the name table. Therefore, to compare with Figure 6-1, performance should be scaled by a factor of two. When this is done, performance of registration is comparable to that of lookups with a single fleet member (roughly 70 for `Register` versus roughly 150 for `Lookup`). This comparison is valid because increasing fleet size does not improve throughput.

One might expect `Register` to exhibit worse performance in this case for several reasons. First, this benchmark uses the message APIs instead of the name service directly. This was done to give a honest assessment of messaging system performance, which is not possible to do for `Lookup`s because the name cache blocks repeated requests to the name service. However, the messaging APIs add extra overhead that is not present when accessing the name service directly. Secondly, `Register` involves larger messages than `Lookup`. So communication costs alone reduce throughput. Finally, the name service has to perform much more work to handle a `Register` than a `Lookup`. Given these considerations, the observed difference is reasonable.

For this benchmark, results are not shown for the microkernel naming implementation, because this does not support `Unregister`.[2]

Register (contended). Figure 6-3 shows results for a `Register`-focused microbenchmark with contention. This benchmark is identical to the previous, except now the registrations use the same name. Each client enters a tight loop that requests `Register` as quickly as possible until it succeeds, and then immediately `Unregister`s it. This is different from prior benchmarks because most requests detect contention and fail immediately. Because each client is in a tight loop, requests therefore come

[2]As noted previously, this was an incomplete implementation of the naming system.

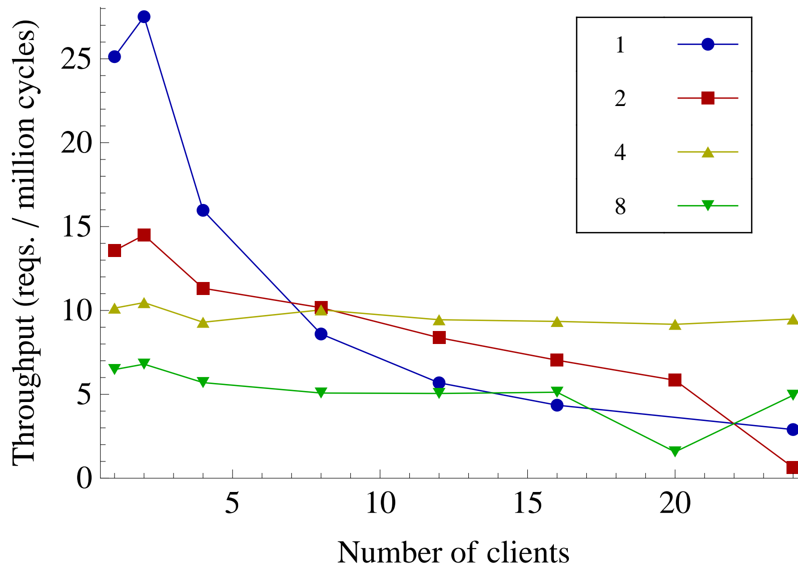

Figure 6-3: `Register` performance under contention in the name service fleet. Legend indicates size of name service fleet. Throughput at a single client is same as uncontended (as expected), but performance does not improve with more clients. For small fleets, performance degradation occurs as clients are added.

much more quickly. This benchmark stresses an unlikely usage scenario, as the primary purpose of conflict resolution is not high performance but correctness. Thus this benchmark shows two things: how the service performs under very high contention, and more importantly, that the service functions correctly and makes some progress when under contention.

Performance for a single client is identical to the previous case, but performance does not improve as clients are added, and for one- and two-member fleets, performance degrades severely. I believe this performance degradation is due to oversaturation of the messaging system, which triggers back off in `Send` that seriously degrades performance. This is unique to this micro-benchmark because of the behavior described above, where requests come in much more quickly than in uncontended registration. Thus degradation is most severe with a single-member fleet, in which the single mailbox suffers the most contention. Performance degradation is less for larger fleet sizes.

Another factor limiting performance is the semantics of contending registrations. Only one registration can succeed at a time, so there is no opportunity for pipelining

of requests. This means that, at best, throughput would be flat as the number of clients increased. This is observed for fleets with four and eight members, where messaging degradation is not observed.

6.3 End-to-End

This section shows results for larger performance tests. Because fos currently lacks the infrastructure to run standardized OS benchmarks, this section uses hand-coded benchmarks for fos's messaging and file systems. These are higher-level benchmarks than those presented in the last section. However, as will be seen, these results are difficult to attribute to the naming system directly.

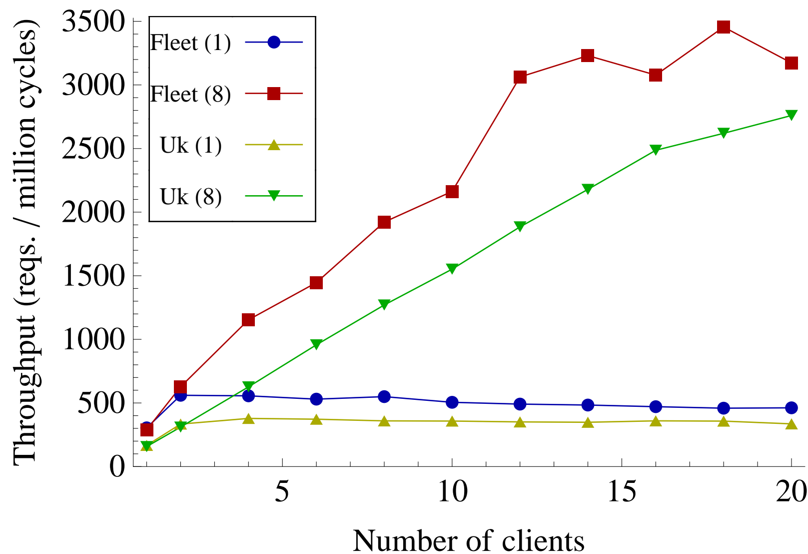

Figure 6-4: Messaging performance comparing the name service fleet design and the prior microkernel naming design. Results are shown for round-trips of small messages, with one and eight servers servicing requests. Legend indicates the design and number of servers (*e.g.*, "Uk (8)" is microkernel with 8 servers). Results demonstrate significantly improved messaging performance in the name service fleet design.

Messaging. Figure 6-4 shows results for the messaging system. This benchmark stresses the messaging system on small message round-trip throughput. Each client sends messages to a fleet of "echo servers," which immediately reply. The echo server fleet contains one or eight servers. Results are shown for the prior, microkernel

naming implementation and the name service fleet implementation. This benchmark is constructed to *avoid* the name service, because each client contacts the name service only during initialization to fill its name cache.

Results show consistently improved performance for the new messaging system. Both the microkernel and fleet implementations use user-space messaging, which employs some form of a user-space name cache in each implementations. Because the name service is not critical to performance in this benchmark, more analysis is needed to determine where the disparity comes from.

These results demonstrate two things. First, moving naming out of the microkernel does not reduce messaging performance. On the contrary, a significant improvement is observed. Second, because messaging performance alone is significantly different from the prior implementation, it is difficult to attribute performance differences in other benchmarks to the name service. This is discussed further at the end of the chapter.

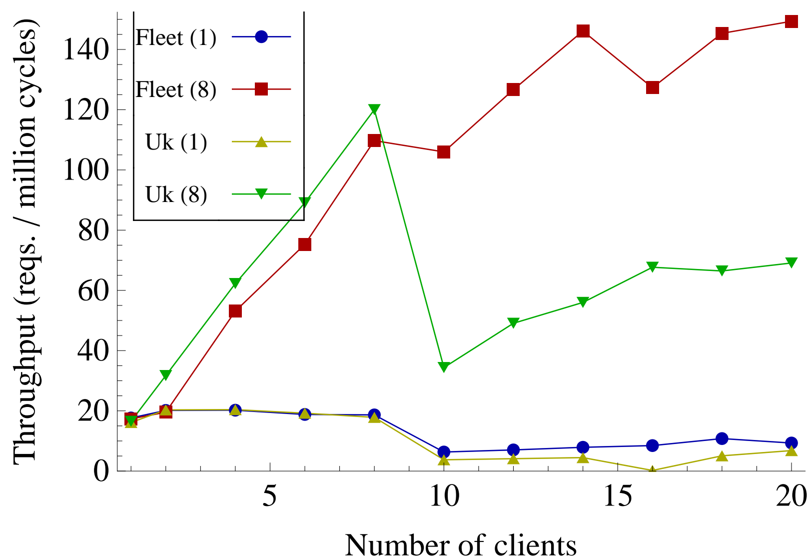

Figure 6-5: Results for the file system micro-benchmark.

File system. Figure 6-5 shows results for a file system micro-benchmark. This micro-benchmark contains a block device server, a read-only file system fleet, and several clients. Each client makes requests that open a unique file, read one kilobyte,

and close the file. Each time a file is opened, the name mapping is refreshed through the name service. These results show comparable performance for the file system up to eight clients. The fleet implementation shows slightly worse performance because it explicitly messages the name service to load balance, whereas the microkernel implementation uses system calls. After eight clients, the microkernel implementation suffers serious performance degradation. This degradation is not observed in the new name service fleet implementation. It is not clear where this comes from, but it is unlikely to be due to the name service itself, rather due to the messaging implementation, which hits a performance bottleneck when the number of clients exceeds the number of servers.

6.4 Discussion

These results demonstrate that the name service scales well under expected usage patterns. `Lookup` performance scales well, and closely follows the trends of the underlying messaging system. The messaging system itself has improved performance when compared to its prior implementation. However, because of the many changes in the system, it is difficult to pinpoint end-to-end differences to a particular source.

More importantly, it is unclear how the name service will affect system performance. As the messaging benchmark shows, best performance is achieved by avoiding the name service entirely. Indeed, this is the sole purpose of the name caches. So it may be the case that the name service is under-utilized in a real fos system, and plays largely a functional role. However, there are usage scenarios that would increase demand on the name service. If a service wants to refine its load balancing, then it may explicitly refresh its local cache to balance requests among multiple fleet members.

On the other hand, there are usage scenarios that would increase the number of registrations. One example that was previously discussed is the interface between the network stack and the network interface service. Because name caching reduces `Lookup` traffic, these registrations may constitute a significant portion of the name service's workload. Optimizations to the data store may be necessary to provide good

performance under these conditions.

Chapter 7

Future Work

There are many areas of research in the name service that merit more exploration. This chapter discusses a few that are under active consideration. Future development in fos will certainly expose new uses and challenges for the name service, and these will be addressed as needed.

7.1 Expectations versus Reality

The most critical issue was discussed at the end of the last chapter (Section 6.4). We must determine if the expected use of the name service matches its use in practice. The simple fact is that we currently do not know how the name service will be used. This will be determined only once fos runs a full set of benchmarks, with services that make intelligent use of the name service. Then it will be possible to determine the usage patterns in practice and whether the name service is a performance bottleneck.

7.2 Local Sub-domains

There is some reason to believe that the name service will not serve `Lookup`s in as high proportion originally believed. That is, performance of `Register` may be equally important in actual use. If experience bears this out, then the distributed data store will need modifications in order to perform well. In particular, `Register` should not

always require global consensus. There a few ways that this can be done.

One approach is to partition the name space among members as done in a traditional distributed hash table. This would optimize performance of `Register` at the cost of `Lookup` performance, as some (most) `Lookup`s would require messaging another fleet member to serve the request. It isn't clear that this is a good tradeoff.

The favored approach is to implement "software cache coherence," so that names that are widely used are cached by all fleet members, and those that are not are stored on a subset of fleet members. This optimizes `Lookup`s for widely-used names, and optimizes `Register` performance to avoid global operations for local names. This is beneficial because the use case that increases `Register` traffic is the network stack and network interface, where a new name is registered for each connection. In this example, the name is only used by the network component local to that machine, not globally.

In this approach, a sub-domain of the name space would be owned by an individual fleet member or a subset of the fleet. `Register`s of names in this sub-domain would only contact the owners. For example, network stack connections could all be registered under `/sys/netstack/connections/<machine id>/*`. This sub-domain would be owned by the name server on the machine, and modifications to the name space would involve a single message round-trip. To keep global consistency, this sub-domain would be registered globally and indicate which fleet members were the owners. An important area of research in this design would be when to change ownership of a sub-domain, and whether this should be managed implicitly by the name service or exposed via the messaging API.

7.3 Fairness in Conflict Resolution

The current implementation of conflict resolution uses a static priority scheme that is unfair. Contention is arbitrated using the name of the fleet member that originated the request. This means that the fleet member with the overall highest priority will tend to complete more requests than other members. This could be resolved using a

dynamic priority scheme, perhaps based on the version of the entry being modified.

7.4 Bootstrapping Cloud Computers

fos currently does not support correct bootstrapping of a cloud computer, *i.e.*an instance with multiple machines. The problem is that the current proxy implementation implements a two-level name space (Section 5.1). In order to properly add a new machine to an existing fos instance, the basic system services (naming, network stack, and proxy) must register with valid names in the *global* name space. Currently, all of these basic services use the same names on each machine.

The valid names will be dynamic, so they must be passed over a network connection after the machine has booted. The services must then re-register under the correct names, and then the machine would be ready to join the fos instance. Additionally, the responsibilities of the proxy service have been reduced, and a new proxy service implementation is needed that correctly uses the naming system and provides correct messaging guarantees.

Chapter 8

Related Work

This section discusses the related work for the fos system as a whole, and naming and distributed data in particular.

8.1 Full System

There are several classes of systems which have similarities to fos: traditional microkernels, distributed OSes, and cloud computing infrastructure.

Traditional microkernels include Mach [2] and L4 [18]. fos is designed as a microkernel and extends the microkernel design ideas. However, it is differentiated from previous microkernels in that instead of simply exploiting parallelism between servers which provide different functions, this work seeks to distribute and parallelize within a server for a single high-level function. fos also exploits the "spatial-ness" of massively multicore processors by spatially distributing servers which provide a common OS function.

Like Tornado [12] and K42 [3], fos explores how to parallelize microkernel-based OS data structures. They are differentiated from fos in that they require SMP and NUMA shared memory machines instead of loosely coupled single-chip massively multicore machines and clouds of multicores. Also, fos targets a much larger scale of machine than Tornado/K42. The recent Corey [7] OS shares the spatial awareness aspect of fos, but does not address parallelization within a system server and focuses on smaller

configuration systems. fos is tackling many of the same problems as Barrelfish [5] but fos is focusing more on how to parallelize the system servers as well as addresses the scalability on chip and in the cloud. Also, in this work we show the scalability of our system servers which was not demonstrated in previous Barrelfish [5] work.

The structure of how fos can proxy messages between different machines is similar to how Mach [2] implemented network proxies with the Network Message Server. Also, Helios's [19] notion of satellite kernels is similar to how fos can have one server make a function call to a server on a different machine.

Disco [8] and Cellular Disco [15] run multiple cooperating virtual machines on a single multiprocessor system. fos's spatial distribution of fleet resources is similar to the way that different VM system services communicate within Cellular Disco. Disco and Cellular Disco argue leveraging traditional OSes as an advantage, but this approach likely does not reach the highest level of scalability as a purpose built scalable OS such as fos will. Also, the fixed boundaries imposed by VM boundaries can impair dynamic resource allocation.

fos bears much similarity to distributed OSes such as Amoeba [25], Sprite [21], and Clouds [11]. One major difference is that fos communication costs are much lower when executing on a single massive multicore, and the communication reliability is much higher. Also, when fos is executing on the cloud, the trust model and fault model is different than previous distributed OSes where much of the computation took place on student's desktop machines.

The manner in which fos parallelizes system services into fleets of cooperating servers is inspired by distributed Internet services. For instance, load balancing is one technique taken from clustered web servers. fos also takes inspiration from distributed services such as distributed file systems such as AFS [23], OceanStore [17] and the Google File System [14].

fos differs from existing cloud computing solutions in several aspects. Cloud (*IaaS*) systems, such as Amazon's Elastic compute cloud (EC2) [1] and VMWare's VCloud, provide computing resources in the form of virtual machine (VM) instances and Linux kernel images. fos builds on top of these virtual machines to provide a single system

image across an IaaS system. With the traditional VM approach, applications have poor control over the co-location of the communicating applications/VMs. Furthermore, IaaS systems do not provide a uniform programming model for communication or allocation of resources. Cloud aggregators such as RightScale [22] provide automatic cloud management and load balancing tools, but they are application-specific, whereas fos provides these features in an application agnostic manner.

8.2 Naming and Distributed Data

The name service was inspired by universal resource identifiers (URIs) in the world wide web, and the domain name system (DNS) in particular. This is a hierarchical naming system that provides access to services, so functionally it is similar to fos's name service. Likewise, DNS can map a single name to multiple different locations (IP addresses), which is similar to fos's indirect mappings. Unlike fos, however, DNS does not need strong consistency and can lazily respond to additions and deletions. The authoritative name server in DNS is similar to the local sub-domain idea discussed in Section 7.2.

Some previous operating systems have included a name service. Helios [19] includes a network service that maps a global name space. The Helios microkernel manages name resolution and a name cache, which is similar to fos. fos has moved these components out of the microkernel, which involves a different security model. Additionally, usage is slightly different, as the Helios name space is divided into subdomains for each processor, whereas the fos name space exposes services on multiple processors under a single name, and is completely agnostic to the placement of the services within the system. Barrelfish includes a name service, but the details of its design and usage are not published.

The data store used in the name service is related to numerous prior projects. The idea of a distributed key-value store has been explored in distributed hash tables, such as Bit Torrent [10] and Chord [24]. The name space is not implemented in a partitioned DHT, however, and its design is more similar to that of Barrelfish's

replication. Two-phase commit is a common technique in database systems [6], also used in Barrelfish. The arbiter objects are similar conceptually to distributed lock managers [9] that synchronize access to shared resources, except in fos the arbiter guards completion of a specific operation and is not exposed via a lock interface.

Chapter 9

Conclusion

This thesis presented the design of the name service within fos. It showed the crucial role that the name service plays in central concepts of fos, namely fleets, elasticity, and spatial scheduling. The design for the naming and messaging system was discussed, showing how a complete system could be constructed with good performance and minimal complexity. Additionally, the detailed design of the distributed data store that holds the global name space was presented. The name service is the first service with complicated distributed state, and its development gave experience that can be applied to future services. Results were presented showing the excellent scalability of name service under expected usage and a significant improvement in messaging throughput in the new design. Finally, the success of this effort gives affirmative evidence of the viability of fleet design in general.

Bibliography

[1] Amazon Elastic Compute Cloud (Amazon EC2), 2009. http://aws.amazon.com/ec2/.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, June 1986.

[3] J. Appavoo, M. Auslander, M. Burtico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[7] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, December 2008.

[8] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 143–156, 1997.

[9] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[10] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[11] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. Applebe, J. M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C. J. Wileknloh. The design and implementation of the Clouds distributed operating system. *USENIX Computing Systems Journal*, 3(1):11–46, 1990.

[12] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 87–100, February 1999.

[13] Simson L. Garnkel. An evaluation of amazons grid computing services: Ec2, s3 and sqs. Technical Report TR-08-07, Center for Research on Computation and Society, School for Engineering and Applied Sciences, Harvard University, August 2007.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2003.

[15] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 154–169, 1999.

[16] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 347–348, New York, NY, USA, 2010. ACM.

[17] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, November 2000.

[18] Jochen Liedtke. On microkernel construction. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 237–250, December 1995.

[19] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[20] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Lamia Youseff, and Dmitri Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of 9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 09)*, Shanghai, China, 2009.

[21] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[22] Rightscale home page. http://www.rightscale.com/.

[23] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–18,20–21, May 1990.

[24] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.

[25] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 558–563, May 1986.

[26] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Harshad Kasture, Lamia Youseff, Jason Miller, and Anant Agarwal. Fleets: Scalable services in a factored operation system, 2010.

[27] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, June 2010.