An Operating System for Multicore and Clouds: Mechanisms and Implementation

A Seminar Report
Submitted in partial fulfilment of
the requirements for the award of the degree of

Master of Technology
in
Computer Science and Engineering

by

Smitha Vas P
M105117

Department of Computer Science & Engineering
College of Engineering Trivandrum
Kerala - 695016
2010-11

College of Engineering Trivandrum
Department of Computer Science & Engineering

Certified that this Seminar Report entitled

An Operating System for Multicore and Clouds: Mechanisms and Implementation

is a bonafide record of the seminar presented by

Smitha Vas P
M105117

in partial fulfilment of
the requirements for the award of the degree of
Master of Technology
in
Computer Science & Engineering

Mr. Rameez Mohammed A
Guide
Faculty
Dept.of Computer Science & Engineering

Dr. M.S Rajasree
Professor and
Head
Dept.of Computer Science & Engineering

ACKNOWLEDGEMENT

I would like to express my sincere gratitude and heartful indebtedness to my Guide and Seminar Coordinator **Mr. Rameez Mohammed A**, Lecturer, Department of Computer Science and Engineering for his valuable guidance and encouragement in pursuing this seminar .

I am thankful to **Dr. M S Rajasree**, Head of the Department, **Mr. Shine S**, P.G Coordinator, **Mrs. Shreelekshmi R**, Asst. Professor and **Mr. Anwar A**, Asst. Professor and Project Coordinator, Department of Computer Science and Engineering for their help and support.

I also acknowledge my gratitude to other members of faculty in the Department of Computer Science and Engineering and all my friends for their whole hearted cooperation and encouragement.

Above all I am thankful to the God Almighty.

Smitha Vas P

CONTENTS

1 INTRODUCTION **5**
1.1 Challenges with Current Cloud Systems 5
1.2 Benefits of a Single System Image 6

2 MULTICORE AND CLOUD OPERATING SYSTEM CHALLENGES **8**
2.1 Scalability 8
2.2 Variability of Demand 8
2.3 Faults 9
2.4 Programming Challenges 9

3 ARCHITECTURE **10**
3.1 Microkernel 11
3.2 Messaging 11
3.3 Naming 12
3.4 OS Services 12
3.4.1 *fos Server Model* 13
3.4.2 *Parallel Data Structures* 14

4 CASE STUDIES **15**
4.1 File System 15
4.2 Spawning Servers 16
4.3 Elastic Fleet 18

5 CONCLUSION **19**

ABSTRACT

Cloud computers and multicore processors are two emerging classes of computational hardware that have the potential to provide unprecedented compute capacity to the average user. In order for the user to effectively exploit all of this computational power, operating systems (OSes) for these new hardware platforms are needed.Existing multicore operating systems do not scale to large numbers of cores, and do not support clouds. Consequently, current day cloud systems push much complexity onto the user, requiring the user to manage individual Virtual Machines (VMs) and deal with many system-level concerns.fos is a single system image operating system across both multicore and Infrastructure as a Service (IaaS) cloud systems. fos tackles OS scalability challenges by factoring the OS into its component system services. Each system service is further factored into a collection of Internet-inspired servers which communicate via messaging. This paper focuses on how to build an operating system which can service both cloud and multicore computers.

1 INTRODUCTION

Users have progressed from using mainframes to minicomputers to personal computers to laptops, and most recently, to multicore and cloud computers.In the past, new operating systems have been written for each new class of computer hardware to facilitate resource allocation, manage devices, and take advantage of the hardware's increased computational capacity. The newest classes of computational hardware,multicore and cloud computers, need new operating systems to take advantage of the increased computational capacity and to simplify users access to elastic hardware resources.

Cloud computing and Infrastructure as a Service (IaaS) promises a vision of boundless computation which can be tailored to exactly meet a user's need, even as that need grows or shrinks rapidly. Thus, through IaaS systems, users should be able to purchase just the right amount of computing, memory, I/O, and storage to meet their needs at any given time. Unfortunately,current IaaS systems lack system wide operating systems, requiring users to explicitly manage resources and machine boundaries. Making operating systems scale, designing scalable internal OS data structures, and managing these growing resources will be a tremendous challenge. Contemporary OSes designed to run on a small number of reliable cores are not equipped to scale up to thousands of cores or tolerate frequent errors. The challenges of designing an operating system for future multicore and manycore processors include scalability, managing elasticity of demand, managing faults, and the challenge of large system programming.

One solution is to provide a single system image OS, making IaaS systems as easy to use as multiprocessor systems and allowing the above challenges to be addressed in the OS. A factored operating system (fos) provides a single system image OS on multicore processors as well as cloud computers. fos does this in two steps. First, fos factors system services of a full-featured OS by service. Second, fos further factors and parallelizes each system service into an Internet-style collection, or fleet, of cooperating servers that are distributed among the underlying cores and machines. All of the system services within fos, along with the fleet of servers implementing each service, communicate via message passing, which maps transparently across multicore computer chips and across cloud computers via networking. For efficiency, when fos runs on shared-memory multicores, the messaging abstraction is implemented using shared memory.

1.1 Challenges with Current Cloud Systems

Current IaaS systems present a fractured and non-uniform view of resources to the programmer. IaaS systems such as Amazon's EC2 [3] provision resources in units of virtual machines (VM). Using virtual machines as a provisioning unit reduces the complexity for the cloud manager, but without a suitable abstraction layer, this division introduces complexity for the system user. The user of a IaaS system has to worry about constructing their application, system concerns such as configuring and managing communicating operating systems. Addressing the system issues requires a completely new skill set than those for application development.

The fractured nature of the current IaaS model extends beyond communication mechanisms to scheduling and load balancing, system administration, I/O devices, and fault tolerance. For system administration, the user of an IaaS cloud system needs to manage a set of different computers,manage user accounts within a machine versus externally via NIS or Kerberos,

manage processes between the machines and keep configuration files and updates synchronized between machines (cfengine) versus within one machine.Last,faults are accentuated in a VM environment because the user has to manage cases where a whole VM crashes as a separate case from a process which has crashed.

Scheduling and load balancing differs substantially within and between machines as well. Existing operating systems handle scheduing within a machine, but the user must often build or buy server load balancers for scheduling across machines. Cloud aggregators and middleware such as RightScale [4] and Amazon's Cloud Watch Auto Scaling [3] provide automatic cloud management and load balancing tools, but they are typically application-specific and tuned to web application serving.

1.2 Benefits of a Single System Image

fos proposes to provide a single system image across multicores and the cloud as shown in Figure 1.

Figure 1: fos provides a single system image across all the cloud nodes

This abstraction can be built on top of VMs which are provided by an IaaS service or directly on top of a cluster of machines. A single system image has the following advantages over the ad-hoc approach of managing VMs each running distinct operating system instances:

- ***Ease of administration*** Administration of a single OS is easier than many machines. Specifically, OS update, configuration, and user management are simpler.
- ***Transparent sharing*** Devices can be transparently shared across the cloud. Similarly, memory and disk on one physical machine can transparently be used on another physical machine (e.g., paging across the cloud)
- ***Informed optimizations*** An OS has local, low-level knowledge, thereby allowing it to make better, finer-grained optimizations than middleware systems.
- ***Consistency*** An OS has a consistent, global view of process management and resource allocation. Intrinsic load balancing across the system is possible, and so is easy process migration between machines based on load, which is challenging with middleware systems.

A consistent view also enables seamless scaling, since application throughput can be scaled up as easily as executing new processes. Similarly, applications have a consistent communication and programming model whether the application resides inside of one machine or spans multiple physical machines. Furthermore, debugging tools are uniform across the system, which facilitates debugging multi-VM applications.

- ***Fault tolerance*** Due to global knowledge, the OS can take corrective actions on faults.

fos prototype provides a single system image across multicores and clouds, and includes a microkernel, messaging layer, naming layer, protected memory management, a local and remote process spawning interface, a file system server, a block device driver server, a message proxy network server, a basic shell, a webserver, and a network stack.

2 MULTICORE AND CLOUD OPERATING SYSTEM CHALLENGES

Cloud computing infrastructure and manycore processors present many common challenges with respect to the operating system.This section introduces the main problems OS designers will need to address in the next decade. fos, seeks to address these challenges in a solution that is suitable for both multicore and cloud computing.

2.1 Scalability

The number of transistors which fit onto a single chip microprocessor is exponentially increasing. In order to turn increasing transistor resources into exponentially increasing performance, microprocessor manufacturers have turned to integrating multiple processors onto a single die. Current OSes were designed for single processor or small number of processor systems. The current multicore revolution promises drastic changes in fundamental system architecture, primarily in the fact that the number of general-purpose schedulable processing elements is drastically increasing. Therefore multicore OSes need to embrace scalability and make it a first order design constraint.The scalability limitations of contemporary OS design include locks, locality aliasing, and reliance on shared memory[2].

Concurrent with the multicore revolution, cloud computing and IaaS systems have been gaining popularity. The number of computers being added by cloud computing providers has been growing at a vast rate, driven largely by user demand for hosted computing platforms. The resources available to a given cloud user are much higher than are available to the non-cloud user. Cloud resources are virtually unlimited for a given user, only restricted by monetary constraints. Thus, it is clear that scalability is a major concern for future OSes in both single machine and cloud systems.

2.2 Variability of Demand

Elasticity of resources can be defined as the aspect of a system where the available resources can be changed dynamically over time. Manycore systems provide a large number of general purpose, schedulable cores. Furthermore, the load on a manycore system translates into number of cores being used. Thus the system must manage the number of live cores to match the demand of the user. Therefore, multicore OSes need to manage the number of live cores which is in contrast to single core OSes which only have to manage whether a single core is active or idle.

In cloud systems, user demand can grow much larger than in the past. Additionally, this demand is often not known ahead of time by the cloud user. It is often the case that users wish to handle peak load without over-provisioning. In contrast to cluster systems where the number of cores is fixed, cloud computing makes more resources available on-demand than was ever conceivable in the past.

A major commonality between cloud computing and multicore systems is that the demand is not static. Furthermore, the variability of demand is much higher than in previous systems and the amount of available resources can be varied over a much broader range in contrast to single-core or fixed-sized cluster systems.

2.3 Faults

Managing software and hardware faults is another common challenge for future multicore and cloud systems. In multicore systems, hardware faults are becoming more common. As the hardware industry is continuously decreasing the size of transistors and increasing their count on a single chip, the chance of faults is rising. With hundreds or thousands of cores per chip, system software components must gracefully support dying cores and bit flips. In this regard, fault tolerance in modern OSes designed for multicore is becoming an essential requirement.

In addition, faults in large-scale cloud systems are common. Cloud applications usually share cloud resources with other users and applications in the cloud. Although each user's application is encapsulated in a virtual container (for example, a virtual machine in an EC2 model), performance interference from other cloud users and applications can potentially impact the quality of service provided to the application.

Programming for massive systems is likely to introduce software faults. Due to the inherent difficulty of writing multithreaded and multiprocess applications, the likelihood of software faults in those applications is high. Furthermore, the lack of tools to debug and analyze large software systems makes software faults hard to understand and challenging to fix. In this respect, dealing with software faults is another common challenge that OS programming for multicore and cloud systems share.

2.4 Programming Challenges

Contemporary OSes which execute on multiprocessor systems have evolved from uniprocessor OSes. This evolution was achieved by adding locks to the OS data structures. There are many problems with locks, such as choosing correct lock granularity for performance,reasoning about correctness, and deadlock prevention [2]. Ultimately, programming efficient large-scale lock-based OS code is difficult and error prone.

Developing cloud applications composed of several components deployed across many machines is a difficult task. The prime reason for this is that current IaaS cloud systems impose an extra layer of indirection through the use of virtual machines. Whereas on multiprocessor systems the OS manages resources and scheduling,on cloud systems much of this complexity is pushed into the application by fragmenting the application's view of the resource pool.Furthermore, there is not a uniform programming model for communicating within a single multicore machine and between machines.The current programming model requires a cloud programmer to write a threaded application to use intra-machine resources while socket programming is used to communicate with components of the application executing on different machines.

In addition to the difficulty of programming these large-scale hierarchical systems, managing and load-balancing these systems is proving to be a daunting task as well. Ad-hoc solutions such as hardware load-balancers have been employed in the past to solve such issues. These solutions are often limited to a single level of the hierarchy (at the VM level). In the context of fos, however, this load balancing can be done inside the system, in a generic manner (i.e. one that works on all messaging instead of only TCP/IP traffic) and on a finer granularity than at the VM or single machine level. Furthermore, with this design, the application developer need not be aware of such load balancing.

3 ARCHITECTURE

fos is an operating system which takes scalability and adaptability as the first order design constraints. fos ventures to develop techniques and paradigms for OS services which scale from a few to thousands of cores. In order to achieve the goal of scaling over multiple orders of magnitude in core count, fos uses the following design principles:

- ***Space multiplexing replaces time multiplexing*** Due to the growing bounty of cores, there will soon be a time where the number of cores in the system exceeds the number of active processes. At this point scheduling becomes a layout problem, not a time-multiplexing problem. The operating system will run on distinct cores from the application. This gives spatially partitioned working sets; the OS does not interfere with the application's cache.

- ***OS is factored into function-specific services, where each is implemented as a parallel, distributed service*** In fos, services collaborate and communicate only via messages, although applications can use shared memory if it is supported. Services are bound to a core, improving cache locality. Through a library layer, libfos, applications communicate to services via messages.

- ***OS adapts resource utilization to changing system needs*** The utilization of active services is measured, and highly loaded services are provisioned more cores (or other resources). The OS closely manages how resources are used.

- ***Faults are detected and handled by OS*** OS services are monitored by watchdog process.If a service fails, a new instance is spawned to meet demand, and the naming service reassigns communication channels.

The following sections highlight key aspects of the fos architecture,shown in Figure 2.

Figure 2: An overview of the fos server architecture, highlighting the cross-machine interaction between servers in a manner transparent to the application. In scenario (a), the application is requesting services from "fos Server a" which happens to be local to the application. In scenario (b), the application is requesting a service which is located on another machine.

In the figure, fos runs on an IaaS system on top of a hypervisor. A small microkernel runs on every core, providing messaging between applications and servers. The global name mapping is maintained by a distributed set of proxy-network servers that also handle inter-machine messaging. A small portion of this global namespace is cached on-demand by each microkernel. Applications communicate with services through a library layer (*libfos*), which abstracts messaging and interfaces with system services.

3.1 Microkernel

fos is a microkernel operating system. The fos microkernel executes on every core in the system. fos uses a minimal microkernel OS design where the microkernel only provides a protected messaging layer, a name cache to accelerate message delivery, basic time multiplexing of cores, and an Application Programming Interface (API) to allow the modification of address spaces and thread creation. All other OS functionality and applications execute in user space. OS system services execute as userland processes, but may possess capabilities to communicate with other system services which user processes do not.

Capabilities are extensively used to restrict access into the protected microkernel. The memory modification API is designed to allow a process on one core to modify the memory and address space on another core if appropriate capabilities are held. This approach allows fos to move significant memory management and scheduling logic into userland space.

3.2 Messaging

fos provides a simple process-to-process messaging API for inter-process communication and synchronization. There are several key advantages to using messaging for this mechanism. One advantage is the fact that messaging can be implemented on top of shared memory, or provided by hardware, thus allowing this mechanism to be used for a variety of architectures. Another advantage is that the sharing of data becomes much more explicit in the programming model, thus allowing the programmer to think more carefully about the amount of shared data between communicating processes. By reducing this communication, better encapsulation as well as scalability is achieved, both are desirable traits for a scalable cloud or multicore operating system.

fos allows conventional multithreaded applications with shared memory. This is in order to support legacy code as well as a variety of programming models. However, operating system services are implemented strictly using messages. This is done to force careful thought about which data are shared to improve scalability.

Using messaging is also beneficial in that the abstraction works across several different layers without concern from the application developer. When one process wishes to communicate with another process it uses the same mechanism for this communication regardless of whether the second process is on the same machine or not. Existing solutions typically use a hierarchical organization where intra-machine communication uses one mechanism while inter-machine communication uses another, often forcing the application developer to choose a-priori how they will organize their application around this hierarchy. By abstracting this communication mechanism, fos applications can simply focus on the application and communication patterns on a flat communication medium, allowing the operating system to decide whether or

not the two processes should live on the same VM or not. Additionally, existing software systems which rely on shared memory are also relying on the consistency model and performance provided by the underlying hardware.

fos messaging works intra-machine and across the cloud, but uses differing transport mechanisms to provide the same interface. On a shared memory multicore processor, fos uses message passing over shared memory. When messages are sent across the cloud,messages are sent via shared memory to the local proxy server which then uses the network (e.g., Ethernet) to communicate with a remote proxy server which then delivers the message via shared memory on the remote node. Each process has a number of mailboxes that other processes may deliver messages to provided they have the credentials. fos presents an API that allows the application to manipulate these mailboxes and their properties. An application starts by creating a mailbox. Once the mailbox has been created, capabilities are created which consist of keys that may be given to other servers allowing them to write to the mailbox. In addition to mailbox creation and access control, processes within fos are also able to register a mailbox under a given name. Other processes can then communicate with this process by sending a message to that name and providing the proper capability. The fos microkernel and proxy server assume the responsibility of routing and delivering messages regardless of whether or not a message crosses machine boundaries.

3.3 Naming

One unique approach to the organization of multiple communicating processes that fos takes is the use of a naming and lookup scheme. Processes are able to register a particular name for a mailbox. This namespace is a hierarchical URI much like a web address or filename. This abstraction provides great flexibility in load balancing and locality to the operating system.

The basic organization for many of fos's servers is to divide the service into several independent processes (running on different cores) all capable of handling the given request. As a result, when an application messages a particular service, the nameserver will provide a member of the fleet that is best suited for handling the request. To accomplish this, all of the servers within the fleet register under a given name. When a message is sent, the nameserver will provide the server that is optimal based on the load of all of the servers as well as the latency between the requesting process and each server within the fleet.

When multiple servers want to provide the same service, they can share a name. One solution to route the message to the correct server, is to have a few fixed policies such as round robin or closest server. Alternatively, custom policies could be set via a callback mechanism or complex load balancer. Meta-data such as message queue lengths can be used to determine the best server to send a message to. The advantage to this design is that much of the complexity dealing with separate forms of inter-process communication in traditional cloud solutions is abstracted behind the naming and messaging API. Each process simply needs to know the name of the other processes it wishes to communicate with, fos assumes the responsibility of efficiently delivering the message to the best suited server within the fleet providing the given service.

3.4 OS Services

A primary challenge in both cloud computing and multicore is the unprecedented scale of demand on resources, as well as the extreme variability in the demand. System services must be

both scalable and *elastic*, or dynamically adaptable to changing demand. This requires resources to shift between different system services as load changes. fos addresses these challenges by parallelizing each system service into a *fleet* of spatially-distributed, cooperating servers. Each service is implemented as a set of processes that, in aggregate, provide a particular service. Fleet members can execute on separate machines as well as separate cores within a machine. This improves scalability as more processes are available for a given service and improves performance by exploiting locality. Fleets communicate internally via messages to coordinate state and balance load. There are multiple fleets active in the system: e.g., a file system fleet, a naming fleet, a scheduling fleet, a paging fleet, a process management fleet, etc.

When demand for a service outstrips its capabilities, new members of the fleet are added to meet demand. This is done by starting a new process and having it handshake with existing members of the fleet. In some cases, clients assigned to a particular server may be reassigned when a new server joins a fleet. This can reduce communication overheads or lower demand on local resources (e.g., disk or memory bandwidth). Similarly, when demand is low, processes can be eliminated from the fleet and resources returned to the system. This can be triggered by the fleet itself or an external watchdog service that manages the size of the fleet.

fos provides (i) a cooperative multithreaded programming model; (ii) easy-to-use remote procedure call (RPC) and serialization facilities; and (iii) data structures for common patterns of data sharing.

3.4.1 *fos Server Model*

fos provides a server model with cooperative multithreading and RPC semantics. The goal of the model is to abstract calls to independent, parallel servers to make them appear as local libraries, and to mitigate the complexities of parallel programming. The model provides two important conveniences: the server programmer can write simple straight-line code to handle messages, and the interface to the server is simple function calls.

Servers are event-driven programs, where the events are messages. Messages arrive on one of three inbound mailboxes: the external (public) mailbox, the internal (fleet) mailbox, and the response mailbox for pending requests. To avoid deadlock, messages are serviced in reverse priority of the above list.

New requests arrive on the external mailbox. The thread that receives the message is now associated with the request and will not execute any other code. The request may require communication with other servers (fleet members or other services) to be completed. Meanwhile, the server must continue to service pending requests or new requests. The request is processed until completion or a RPC to another service occurs. In the former case, the thread terminates. In the latter, the thread yields to the cooperative scheduler, which spawns a new thread to wait for new messages to arrive.

Requests internal to the fleet arrive on the internal mailbox. These deal with maintaining data consistency within the fleet, load balancing,or growing/shrinking of the fleet. Otherwise,they are handled identically to requests on the external mailbox. They are kept separate to prevent others from spoofing internal messages and compromising the internal state of the server.

Requests on the response mailbox deal with pending requests. Upon the receipt of such a message, the thread that initiated the associated request is resumed.

The interface to the server is a simple function call. The desired interface is specified by the programmer in a header file, and code is generated to serialize these parameters into a message to the server. Likewise, on the receiving end, code is generated to deserialize the parameters and pass them to the implementation of the routine that runs in the server. On the caller side, the thread that initiates the call yields to the cooperate scheduler. When a response arrives from the server, the cooperative scheduler will resume the thread. The cooperative scheduler runs whenever a thread yields. If there are threads ready to run (e.g., from locking), then they are scheduled. If no thread is ready, then a new thread is spawned that waits on messages. If threads are sleeping for too long, then they are resumed with a time out error status.

3.4.2 *Parallel Data Structures*

One key aspect to parallelizing operating system services is managing state associated with a particular service amongst the members of the fleet.The idea is to provide a common container interface, which abstracts several implementations that provide different consistency,replication, and performance properties.In this solution, the operating system and support libraries provide an interface for storing and retrieving data. On the back-end, each particular server stores some of the data (acting as a cache) and communicates with other members of the fleet to access state information not homed locally. Special care needs to be taken to handle joining and removing servers from a fleet. By using a library provided by the operating system and support libraries, the code to manage this distributed state can be tested and optimized, alleviating the application developer from concerning themselves with the implementation of distributed data structures.

4 CASE STUDIES

The key components of fos are illustrated here.It also describes how fos works and how fos solves key challenges in the cloud.

4.1 File System

Figure 3: Anatomy of a File System Access

An example of the interaction between the different servers in fos is the fos file server. Figure 3 depicts an anatomy of a file system access in fos. In this figure, the application client, the fos file system server and the block device driver server are all executing on distinct cores to diminish the cache and performance interferences among themselves. Since the communication between the application client and systems servers, and amongst the system servers, is via the fos messaging infrastructure, proper authentication and credential verification for each operation is performed by the messaging layer in the microkernel. All services are assumed to be on the same machine, however the multi-machine case is a logical extension to this example, with a proxy server bridging the messaging abstraction between the two machines.

fos intercepts the POSIX file system calls in order to support compatibility with legacy POSIX applications. It bundles the POSIX calls into a message and sends it to the file system server. The microkernel determines the destination server of the message and verifies that the client application possesses the requisite capabilities to communicate with the server. It, then looks up the destination server in its name cache and determines which core it is executing on. If the server is a local server (i.e. executing on the same machine as the application), the

microkernel forwards the message to the destination application.
In Figure 3, fos intercepts the application File system access in step 1. In step 2, system call is bundled in a message to be sent via the messaging layer. In step 3,since the destination server for this message is the file system server, fos queries the name cache and sends the message to the destination core.

Once the file system server receives a new message in its incoming mailbox queue, it services the request. If the data requested by the application is cached, the server bundles it into a message and sends it back to the requesting application. Otherwise, it fetches the needed sectors from disk through the block device driver server.
In the file system anatomy figure, step 5 represents the bundling of the sectors request into *block messages* while step 6 represents the look-up of the block device driver in the name cache. Once the server is located, the fos microkernel places the message in the incoming mailbox queue of the block device driver server as shown in step 6.

The block device driver server provides Disk I/O operations and access to the physical disk. In response to the incoming message, the block device driver server processes the request enclosed in the incoming message, fetches the sectors from disk as portrayed in steps 7, 8 and 9 respectively in the figure. Afterward, it encapsulates the fetched sectors in a message and sends it back to the file system server, as shown in steps 10, 11 and 12. In turn, the file server processes the acquired sectors from the incoming mailbox queue, encapsulates the required data into messages and sends them back to the client application. In the client application, libfos receives the data at its incoming mailbox queue and processes it in order to provide the file system access requested by the client application. These steps are all represented by steps 13 through 15 in the file system access anatomy in Figure 3.

Libfos provides several functions, including compatibility with POSIX interfaces. The user application can either send the file system requests directly through the fos messaging layer or through *libfos*. In addition, if the file system server is not running on the local machine (i.e. the name cache could not locate it), the message is forwarded to the proxy server. The proxy server has the name cache and location of all the remote servers. In turn, it determines the appropriate destination machine for the message, bundles it into a network message and sends it via the network stack to the designated machine.

4.2 Spawning Servers

To expand a fleet by adding a new server, one must first spawn the new server process. As shown in Figure 4, spawning a new process take into account the machine on which the process should be spawned. Spawning begins with a call to the spawnProcess() function; this arises through an intercepted exec syscall from the POSIX compatibility layer, or by directly calling the spawnProcess function by a fos-aware application. By directly calling the spawnProcess function, parent processes can exercise greater control over where their children are placed by specifying constraints on what machine to run on, what kinds of resources the child will need, and locality hints to the scheduler.

The spawnProcess function bundles the spawn arguments into a message, and sends that message to the spawn servers incoming request mailbox. The spawn server must first determine which machine is most suitable for hosting that process. It makes this decision by considering the available load and resources of running machines, as well as the constraints

given by the parent process in the spawnProcess call. The spawn server interacts with the scheduler to determine the best machine and core for the new process to start on. If the best machine for the process is the local machine,the spawn server sets up the address space for the new process and starts it. The spawn server then returns the PID to the process that called spawnProcess by responding with a message. If the scheduler determined that spawning on a remote machine is best, the spawn server forwards the spawn request to the spawn server on the remote machine, which then spawns the process.

Figure 4: Spawning a VM

If the local spawn server was unable to locate a suitable machine to spawn the process, it will initiate the procedure of spawning a new VM. To do this, it sends a message to the cloud interface server, describing what resources the new machine should have;when the cloud interface server receives this message, it picks the best type of VM to ask for. The cloud interface server then spawns the new VM by sending a request to the cloud manager via Internet requests (the server outside of fos which is integrated into the underlying cloud infrastructure eg. EC2). When the cloud manager returns the VM ID, the cloud interface server waits until the new VM acquires an IP address. At this point, the cloud interface server begins integration of the new VM into the fos single system image.

The newly-booted VM starts in a bare state, waiting for the spawner VM to contact it. The cloud interface server notifies the local proxy server that there is a new VM at the given IP address that should be integrated into the system, and the proxy server then connects to the remote proxy server at that IP and initiates the proxy bootstrap process. During the bootstrap process, the proxy servers exchange current name mappings, and notify the rest of the machines that there is a new machine joining the system. When the local proxy server finishes this setup, it responds to the cloud interface server that the VM is fully integrated. The cloud interface server can then respond to the local spawn server to inform it that there is a new machine that is available to spawn new jobs, which then tells all the spawn servers in

the fleet that there is a new spawn server and a new machine available. The local spawn server finally forwards the original spawn call to the remote spawn server on the new VM.

In order to smooth the process of creating new VMs, the spawning service uses a pair of high- and low-water-marks, instead of spawning only when necessary. This allows the spawning service to mask VM startup time by preemptively spawning a new VM when the resources are low but not completely depleted. It also prevents the ping-ponging effect, where new VMs are spawned and destroyed unnecessarily when the load is near the new-VM threshold, and gives the spawn servers more time to communicate with each other and decide whether a new VM needs to be spawned.

4.3 Elastic Fleet

As key aspects of the design of fos include scalability and adaptability, a fleet grows to match demand as described here. If, while the system is running, the load changes, then the system should respond in a way that meets that demand if at all possible. In the context of a fos fleet, if the load become too high for the fleet to handle requests at the desirable rate, then a watchdog process for the fleet can grow the fleet. The watchdog does this by spawning a new member of the fleet and initiating the handshaking process that allows the new server to join the fleet. During the handshaking process, existing members of the fleet are notified of the new member, and state is shared with the new fleet member. Additionally, the scheduler may choose to spatially re-organize the fleet so as to reduce the latency between fleet members and those processes that the fleet is servicing.

If there are many servers on a single machine that are all requesting service look-ups from the nameserver, the watchdog process may notice that all of the queues are becoming full on each of the nameservers. It may then decide to spawn a new nameserver and allow the scheduler to determine which core to put this nameserver on so as to alleviate the higher load. By using the programming model provided for OS services as well as the parallel data structures for backing state, many servers can easily enjoy the benefit of being dynamically scalable to match demand.

While the mechanism for growing the fleet will be generic, there are several aspects of this particular procedure that will be service specific. One issue that arises is obtaining the meta-data required to make this decision and choosing the policy over that meta-data to define the decision boundary. To solve this issue, the actual policy can be provided by members of the fleet.

The fact that this decision is made by part of the operating system is a unique and advantageous difference fos has over existing solutions. In particular, the fleet expansion (and shrinking) can be a global decision based on the health and resources available in a global sense, taking into consideration the existing servers, their load and location (latency) as well as desired throughput or monetary concerns from the system owner. By taking all of this information into consideration when making the scaling scheduling decision, fos can make a much more informed decision than solutions that simply look at the cloud application at the granularity of VMs.

5 CONCLUSION

Cloud computing and multicores have created new classes of platforms for application development; however, they come with many challenges as well. New issues arise with the fractured resource pools in clouds. Also new OSes need to deal with with a dynamic underlying computing infrastructure due to varying application demand, faults, or energy constraints. fos, seeks to surmount these issues by presenting a single system interface to the user and by providing a programming model that allows OS system services to scale with demand. By placing key mechanisms for multicore and cloud management in a unified operating system, resource management and optimization can occur with a global view and at the granularity of processes instead of VMs. fos is scalable and adaptive, thereby allowing the application developer to focus on application-level problem-solving without distractions from the underlying system infrastructure.

References

[1] David Wentzlaff,Charles Gruenwald III,Nathan Beckmann and A. Agarwal. . "An Operating System for Multicore and Clouds: Mechanisms and Implementation" *SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing*

[2] D. Wentzlaff and A. Agarwal. . "Factored operating systems (fos): the case for a scalable operating system for multicores" *SIGOPS Oper. Syst. Rev., 43(2):76-85, 2009.*

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

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