♦ Predictable Cloud Computing

Sape J. Mullender

The standard tools for cloud computing—processor and network virtualization—make it difficult to achieve dependability, both in terms of real time operations and fault tolerance. Virtualization multiplexes virtual resources onto physical ones, typically by time division or statistical multiplexing. Time, in the virtual machine, is therefore as virtual as the machine itself. And fault tolerance is difficult to achieve when redundancy and independent failure in the virtual environment do not necessarily map to those properties in the physical environment. Virtualization adds a level of indirection that creates overhead, and makes it all but impossible to achieve predictable performance. Osprey uses an alternative to virtualization that achieves the same goals of scalability and flexibility but carries neither the overhead of virtualization, nor the restrictions on dependability. The result is a programming environment that achieves most of the compatibility offered by traditional virtualization efforts and provides much better and much more predictable performance. One technique we use is called Library OS, which stems from high-performance computing. The technique consists of linking applications with a library that implements most services normally provided by the operating system, creating an application that can run practically stand alone, or at least with a very minimal operating system. The Library OS approach moves the boundary between application and operating system down to a level where interactions with the operating system consist of sending/receiving messages (e.g., network packets) and scheduling resources (processor, memory, network bandwidth, and device access). These interactions, as we demonstrate, form a relatively weak bond between an application and the particular instance of the operating system on which it runs—one that can be broken and re-established elsewhere. In fact, we make sure this is the case. Legacy applications that cannot be recompiled or relinked can make use of a Library OS server that runs as a tandem process along with the legacy application processes. System calls from the legacy process are catapulted into the Library OS server which executes them. Applications can still migrate, taking their server process along with them. © 2012 Alcatel-Lucent.

Bell Labs Technical Journal 17(2), 25–40 (2012) © 2012 Alcatel-Lucent. Published by Wiley Periodicals, Inc.
Published online in Wiley Online Library (wileyonlinelibrary.com) • DOI: 10.1002/bltj.21542

Panel 1. Abbreviations, Acronyms, and Terms

3G—Third generation cellular network
AMD—Advanced Micro Devices
AoE—ATA over Ethernet
ATA—Advanced Technology Attachment
CPU—Central processing unit
DNS—Domain name system
DSP—Digital signal processor
ELF—Executable and linking format
ETH—Swiss Federal Institute of Technology
FOX—Fault-Oblivious eXascale
FPGA—Field programmable gate array
HA—Home agent
HARE—Holistic Aggregate Resource Environment
HP—Hewlett-Packard
HPC—High-performance computing
I/O—Input/output
IP—Internet Protocol
JVM—Java virtual machine
KVM—Kernel-based virtual machine
MIT—Massachusetts Institute of Technology
MN—Mobile node
NUMA—Non-uniform memory architectures
OS—Operating system
PC—Personal computer
PD—Process descriptor
SoC—System on a chip
SRAM—Static random access memory
TCP—Transmission Control Protocol
UART—Universal asynchronous receiver/transmitter
VPN—Virtual private network

Introduction

Distributed computing is the name of a branch of computer science that studies how exploiting independent failure in a network of computers can be used to build applications that are more reliable than their constituent components. Most of the groundwork for distributed computing was laid in the 1980s, just as personal computers (PCs) started becoming popular. The PC provided a reliable personal computing platform on the desktop. Combined with servers for file sharing and printing, most users saw little need for the complexity of distributed computing, and the field started languishing in the 1990s.

But the recurring phenomenon of technology moving in waves manifests itself again: once more, there are reasons to revisit distributed systems. One big reason is that, now, there are some truly globe-spanning distributed applications: Google* Search, Facebook*, Twitter*, and Skype*. Additionally, the advent of relatively resource-poor personal communication devices (the smartphone, netbook, and tablet computer) that depend on services in the network (map, nearby friends, YouTube*, Kindle*) are also driving research to re-examine how services can be delivered *from* the network.

Cloud computing is the term currently used for leveraging large numbers of servers in data centers for applications that share these servers, increasing and decreasing the demand they place on the server pool. In the past, we did something similar by running a divergent set of applications on one very large mainframe computer. We called it *time sharing*.

Interestingly, but nicely demonstrating the cyclic nature of systems engineering, most of the issues that arose in the time-sharing environments of the 1970s and 1980s have returned in cloud computing.

Fault Tolerance

The cloud will become a place in which extremely long-running applications are likely to find a home. Servers typically run for weeks and months on end, but users may also run applications in the cloud to which they connect from sundry places.

Long-running applications are more exposed to failures: power failures, system crashes, and network outages, and therefore they need better protection. Failures can be *masked*; that is, their consequences can be made invisible by having backup processes, network connections, or storage servers step into the breach. This can only be done if failures can be *contained*—a

failure in one component should not bring others down with it—and if there is *redundancy*. There have to be resources available to step into that breach when necessary.

Usually, the redundant resources have to be identified before any failure occurs. Essentially, there is an algorithm that decides which resources will take over in a given failure scenario.

Identifying such resources can be difficult and requires a vertical scrutiny of services, the platforms they run on, the hardware that supports those platforms, and the energy source for those platforms. All processes that are ultimately supported by a single power supply will fail if that power fails, so they cannot be backups for each other in the case of power failure. Similarly, redundant network links that share a fiber link somewhere can't be backups for one another if the light goes out.

Failure recovery requires distribution of system state. If a component crashes without a description somewhere else of what it was supposed to do, it becomes impossible to recover from that failure. Interestingly, state replication is in itself a candidate for failures: Can the state be updated if a copy is unreachable or unresponsive? What happens if the network fails in a way that leaves two sets of systems rendered incommunicado?

The latter failure actually causes a fundamental impossibility in a fault tolerant system: two servers that cannot communicate cannot know whether it is because their peer has crashed or their peer is "merely" unreachable. In the former case, the remaining server could continue to provide service and brief the crashed server when it comes back up. In the latter case, the servers, by continuing to work, may end up with inconsistent states, which they'll discover once the network is repaired. In other words, in a network that can be partitioned, you can obtain perfect availability or perfect consistency, but not at the same time. Think of this as the uncertainty principle of fault tolerant computing [9, 18].

Real Time

Real time service is a concern for the delivery of media and communication services. For telecom operators this is a familiar issue. Telephone networks have always operated in real time, and after convergence, real time service has become important in Internet Protocol (IP) networks too.

With the advent of media content delivery and gaming over IP networks, real time service is affecting a range of services, many of which are, or can be, delivered from the cloud, so real time computing in the cloud is important.

Real time processing requires a concerted effort of many real time components to get real time results: the ultimate requirement for the smooth presentation of a movie is that an image is presented on the screen every, say, 30 ms; but to make this possible the network must deliver the next batch of video data in time, the server must get chunks of movie from the storage server, and the disks have to be scheduled to deliver all the flowing media streams on time, simultaneously [5].

Providing real time services is even harder because of the possibility of faults. The timing budget must take into account the extra time needed for failover to appropriate standby components. Note that for achieving fault tolerance, failures must be considered as part of the specification: the types of failure (failures by crashing, failures by timing, failures by producing incorrect results, and failure through packet loss) as well as the maximum number of failures allowed.

Mobility

Cloud computing has to consider mobility in two ways. First, users accessing the cloud are becoming increasingly mobile. In fact, mobile devices, given their size and their reliance on batteries, can make very good use of services that run in the cloud. Such services, therefore, will have to be prepared to deal with client address changes as well as less than completely reliable communication.

Second, in the cloud itself, considerations for load balancing, moving computations to the data, and scheduled maintenance will force applications and services to relocate. Relocation may cause short service disruptions, but afterwards, things must work normally again, albeit from a different address.

The protocol underlying practically all Internet communication is the Internet Protocol [6, 22]. In the IP specification, Postel [22] paraphrases Shoch [25]:

> The name of a resource indicates *what* we seek
> an address indicates *where* it is, and
> a route tells us *how to get there*.

In practice however, an IP *address* has now become much more of a *name*. The slow-changing nature of the Domain Name System (DNS) doesn't accommodate changes in mapping from names to rapidly changing IP addresses very well, and higher-level protocol connections, e.g., the Transmission Control Protocol (TCP), cannot deal with midstream IP address changes.

A few applications cope reasonably well with address changes, however. Browsers and mailers are good examples. One can carry a laptop around and, wherever it gets plugged in, it can browse and email. For everything else, Mobile IP appears to be the model of choice, but as we shall show in the next section, it is far from perfect.

Virtualization Considered Harmful

The title of this section pays homage to Rubin [24] and Moore et al. [14], who cautioned that "the use of 'considered harmful' is sometimes considered harmful."

Consider that the data center, in a sense, is the modern equivalent of the time-sharing mainframe of yore. Its computers could run an operating system, possibly with some load-sharing facility, and applications could run in the data center just like they used to run on a time-sharing system.

Unfortunately, it's not that simple any more. Two things happened. One is that the data center is available to all and sundry and, therefore, has to be much better protected against malicious users and applications. The other is that operating systems have become so much more complicated that they can no longer be trusted to protect applications from attack, nor can they always prevent applications from successfully stealing operating system resources.

In the style of "All problems in computer science can be solved by another level of indirection," the current solution of record is processor virtualization. (This quote that has been attributed to Professor David Wheeler of Cambridge University. He is also said to have added "... except for the problem of too many layers of indirection.")

The data center's computers run an operating system that implements virtual machines with an operating system interface that mimics the hardware interface with sufficient fidelity to be able to run a conventional operating system on it *as an application*; that operating system becomes a *guest operating system*. Several such guest operating systems can run on a single computer and they need not be the same. Windows* and Linux* can happily share a machine in this way. The best known example of virtualizing a host operating system is Xen [2], but there are now many others, including kernel-based virtual machine (KVM), VMWare*, and VirtualPC.

The reason processor virtualization has caught on in cloud computing is that the virtual machine implemented by the host operating system is very simple—it is the computer's hardware. This simplicity allows the host to realize very good isolation from one guest to another. It also makes it feasible to pick up a guest, and move it to a different machine: the state of a guest is represented by its memory contents and a set of registers, and these are easy to capture and copy.

But that very same interface, by the nature of it being fixed by the hardware it emulates, hides information the guest could use from the host and vice versa. In particular, the guest has no control over (and no way to express to the host) when it gets scheduled, or what fraction of the host resources it can consume.

In other words, one can't schedule a real time process in a guest operating system and have any realistic hope it will run in "physical" real time. One could add interfaces for resource scheduling to the host operating system's interface (and in so-called *para*-virtualization systems, additional "system calls" have already been defined), but adding—necessarily complicated—interfaces for scheduling places virtualization on the slippery slope of adding precisely the interface complexity that virtualization seeks to eliminate.

In cloud systems, IP addresses are usually *virtualized* by the extra level of creating a virtual private network (VPN) that maintains a dynamic mapping between virtual (and migratable) addresses and physical ones and also between virtual network connections and physical ones.

Again, the virtualization hides information essential to achieving real time operations and fault tolerance: a

virtual link has a capacity that changes as virtual machines are relocated and disjoint virtual paths may share a physical path creating failure dependencies unobservable in the virtual world. Again, one could add management interfaces to virtualized networks, allowing the specification of fault tolerance properties and minimum throughput and latency requirements, but these would precisely eliminate the transparency virtualized networks aim to create.

In addition to these fundamental problems in achieving real time performance and fault tolerance, virtualization also adds significant overhead, first by duplicating a lot of mechanisms: routing in the VPN *as well as* the underlying network, process scheduling in the guest *and* in the host operating system, memory protection, and paging in the guest *and* in the host operating system as just a few examples. It just doesn't make sense to do load balancing by migrating a gigabyte operating system for the purpose of relocating a hundred megabyte application.

Mobile IP is not usually considered as a network-virtualization technique, but it is: A Mobile IP address isn't an address at all, it is the name of a mobile node (MN). Packets are delivered to the MN by sending them to a different node, the home agent (HA), which then forwards them to the MN by putting them inside other packets—a tunnel. The MN informs the HA of its whereabouts by registering a forwarding address with the HA every time it moves (and every time the registration is about to expire). Mobile IP achieves mobility by adding a level of indirection to packet routing.

Here, Wheeler's observation hits home: adding levels of indirection leaves one with the problem of too many such levels. Virtualization not only doesn't help real time and fault-tolerant processing, in many cases it actually makes it impossible to do.

Operating System

For the reasons given in the previous section, we have explored different ways of achieving the benefits of virtualization without the problems.

At the basis of our design is a microkernel operating system called Osprey*, which assumes responsibility for the management of shared resources: the memory, processor cycles, and outgoing network bandwidth. Each application receives a *budget* for these resources which gives it the right to consume the resources in the budget and guarantees that those resources are available.

The specification of the budget can be in terms of real time performance or best effort performance, and applications can request statistical or absolute guarantees for the availability of those resources at run time.

The other main responsibility of the Osprey kernel is security, particularly protecting applications from the unwanted attention of other applications; address-space protection is an obvious example.

The system-call interface is kept as small as possible. When applications attempt to leave the confines of their protection envelope (for example, to address unmapped memory or execute instructions that trap), a server process designated—and trusted—by the application can be invoked to attend to the event. This is an important mechanism for emulating mainstream operating system interfaces.

In general, however, we expect applications to be linked with a library that translates conventional system calls into a combination of Osprey system calls and server interactions. The library forms the replacement operating system for (legacy) applications.

Related Research

For background on related research, we will provide details below on the Nemesis multimedia operating system, the Barrelfish operating system, and Library operating systems in general before providing an in-depth description of the Osprey system.

Nemesis

With funding from an Esprit Basic Research Programme, the systems groups at the Universities of Cambridge and Twente developed the Nemesis multimedia operating system in the 1990s [11, 20].

Nemesis introduced a scheduling discipline that was an interesting mix of real time and best effort. The notion was that multimedia applications need a consistent share of the processing resources, but that this share may change as the application mix changes. What should not happen in Nemesis, however, is that one application, by misbehaving in some way, can suddenly deprive other applications of their rightful share of resources.

Nemesis achieves this by two mechanisms. One is to make sure that all resources consumed either by or for an application are counted towards that application's resource budget. The other is a set of scheduling regimes that mix best effort and real time. The idea is that an application can have a guaranteed real time allocation with additional recurring best effort allocations. A multimedia application can, for example, use the real time allocation to guarantee an audio connection and use the unguaranteed allocation to do the best it can to add video.

Nemesis introduced the term *quality-of-service crosstalk* [23] for the effect that one process can have on another's access to processor resources, and it set out to eliminate that effect by carefully assigning every activity to the resource budget of the application that caused it to happen.

After doing this, schedulers can begin to calculate feasible budget allocations and keep applications within their own budgets.

Osprey uses the *quality-of-service crosstalk* lesson from Nemesis to assign all activities to resource budgets. At the moment, Osprey scheduling just distinguishes real time and best effort and counts on obtaining any kind of real time performance by using collaborating real time and best effort processes.

Barrelfish

The Barrelfish operating system [3] is being developed under a joint project of the ETH (Swiss Federal Institute of Technology) Systems Group in Zürich and Microsoft Research.

The project focuses on designing efficient operating systems for modern multicore architectures. The research team argues that as the number of cores increases with Moore's law, the efficiency of memory sharing is reduced: the cost of synchronization becomes a bottleneck.

Baumann et al. [4] present an operating system design in which each core runs its own microkernel, and communication between cores takes place exclusively by message passing. Although they say that message passing should be realized in a processor-dependent fashion, they have just one implementation, and on the x86 architecture they use cache-coherent shared memory for cache-line-sized message passing.

For the most part, Barrelfish uses *polling* for message reception; that is, the sending core places the message in the receiving core's queue where the receiver will eventually find it.

Although Osprey's architecture is significantly influenced by the Barrelfish design, our concern of guaranteeing real time deadlines made us reconsider this notion of using polling.

Library Operating Systems

A *Library operating system* provides an operating system interface to a program via a library that is linked with the program and emulates the system calls; some of the system calls are emulated locally, inside the library itself, others are emulated by interacting with a variety of servers and with other operating systems.

The term *Library operating system* appears to stem from the MIT Lab for Computer Science, where the idea was used in Exokernel [8], but it is in fact much older. The Amoeba distributed operating system used it as well. Mullender et al. [21] state:

> To bridge the gap with existing systems, Amoeba provides a Unix* emulation facility. This facility contains a library of Unix system call routines, each of which does its work by making calls to the various Amoeba server processes.

Today, there are reasons for using library operating systems other than merely getting a program written for one system to run on another. The Exokernel team already recognized this and used "libos" to give processes better performance by cutting down on the overhead of user/kernel transitions.

Libra* [1] does the same, in the context of high-performance computing.

As a special case of library operating systems, we must consider user space implementations of protocol stacks, especially those that were done as a performance-enhancing measure.

One of the earlier experiments was carried out at HP Labs, where Edwards and Muir [7] showed significant performance benefits by running the TCP stack in the user space.

A streaming protocol, such as TCP, creates a servo loop consisting of packets travelling in one direction

and feedback (acknowledgements) in the other. A kernel implementation of such a protocol terminates this servo loop in the kernel, but creates an additional, coupled one consisting of the packet traffic between the kernel packet buffers and user space buffers (via system calls such as send/receive).

Van Jacobson demonstrated [10] that such coupled servo loops slow things down and that a pure user space implementation of TCP can perform much better than a kernel space one.

Osprey

Osprey is not an operating system for cloud computing alone. It is also eminently applicable as a server operating system, an embedded platform that might even be adapted for portable devices. The project team collaborates closely with the Bell Labs team working on the Holistic Aggregate Resource Environment/Fault-Oblivious eXascale (HARE/FOX) high-performance computing project, making HPC another potential target for Osprey.

Moore's law states that the complexity of minimum component costs (on an integrated circuit) increases at a rate of roughly a factor of two per year [16]; he later corrected this to "doubling only every couple of years." (Apparently, Gordon Moore never said "every eighteen months" [15]). Until recently, Moore's Law was primarily expressed by the fact that memories grew larger (and wider) and processors became faster. Processors now typically run with clock rates of 3 GHz or so and the increase in clock rates has slowed down. Processors now have as many transistors as they need and the doubling of component counts is now the cause for the appearance of more processors on a chip: the *multicore* processor.

This trend started with the so-called *system on a chip* (SoC), which combined a processor and a number of peripheral devices (e.g., Ethernet, universal asynchronous receiver/transmitter (UART), static random access memory (SRAM)) on a chip. Now SoCs are used a lot in mobile devices where they routinely combine processor, graphics processor, Wi-Fi, and 3G radio.

Today, "pure" multicore processors behave as symmetrical multiprocessors: all processors are the same and every processor has the same access to memory. As the number of cores grows (eight is common today; 64 will be common in a few years' time), things will become less symmetrical: we may expect different types of processors on a chip (a combination, as it were, of multicore and SoC) and see general purpose processors; digital signal processors (DSPs), as, for example, in graphics processors; field programmable gate arrays (FPGAs); and highly integrated peripherals. We'll see non-uniform memory architectures (NUMA) because otherwise memory will become a bottleneck.

If a processor has different core types and processors cannot access all of the memory, running a single operating system image with shared data structures stops making sense. In fact, on today's symmetrical multiprocessors, it already makes sense to run separate copies of the operating system kernel on each core. This leaves processes assigned to a particular core, but most operating systems already attempt to run processes on the same core as much as possible for reasons of *cache affinity* [12]. An operating system application can be used to perform medium term load balancing by transferring processes from one core to another (or by assigning new processes to the lightest-loaded one).

Each core, thus, schedules its own set of processes. It does, however, expose information about the set of processes it owns to the other cores in a way that allows another core to make an informed decision about whether it should interrupt the kernel when a process needs to be run more urgently than the process currently running.

Messages from processes on one core to a process on another are routed via a central mailbox on the receiving core so the scheduler on the receiving core becomes aware of any process wakeups that may be needed.

Kernel Structure

Osprey cores run more-or-less independent copies of the operating system. Each core, therefore, schedules the user and kernel processes assigned to it and manages a portion of physical memory.

On each core, we run *tasks*. Tasks are kernel processes. Some of these tasks may carry responsibility for a process in the user space. Each user process is managed by a dedicated task. The task is scheduled so the user process may run and, when an interrupt

interrupts a user process, its task saves the process state before passing control to the scheduler.

User processes can be multithreaded, but thread management takes place completely in the user space and the thread model is, therefore, not dictated by Osprey.

All interprocess (and interthread) communication takes place by means of *messages*. Messages are data structures whose size is a cache line (typically 32 or 64 bytes). They are transmitted by copying them and delivering them in message *queues*. Messages may contain pointers to, for example, buffers or network packets. When a message is copied, the pointers are also copied, but not the data they point to. It is the responsibility of threads, processes, and tasks to make sure that pointer values make sense as they are interpreted in different address spaces.

Traditional *system calls* are largely replaced by message exchanges between a process and the operating system. Each process is created with a queue segment (usually just one or two small pages) at a well known address. This contains at least two queues, one, $U \rightarrow K$, for messages from the user space to the kernel and one, $K \rightarrow U$, for messages from the kernel to user space. A system call is performed by placing a request on $U \rightarrow K$ and waiting for a reply on $K \rightarrow U$. Processes are allowed to send multiple system call requests before waiting for system call replies.

There are a few *real* system calls: one, for instance, that effectively says "I'm waiting for a reply" and one that says "I'm done, exit."

Given these asynchronous system calls, it becomes relatively straightforward to implement many thread models in user space.

As the numbers of cores on processors grow, synchronization across all cores will become relatively more costly. In our design, we follow the trend set by the Barrelfish operating system [4] and treat the processor as an interconnected network of cores. We don't follow it quite as far however, and in contrast to Barrelfish, we make use of interrupt-based signaling between cores that help guarantee deadlines while, paradoxically, reducing numbers of interrupts.

This is best explained by taking an extreme example. Suppose a core has two processes, one, *R*, scheduled as a *sporadic* real time process, the other, *B*, as a best effort process. *R*, we assume, is waiting for a message that will release it and then it will have to be scheduled to meet its deadline. *B* is running; let's say it's trying to factor the product of two ten-thousand-bit primes (which should keep it busy for the foreseeable future). The scheduler can leave *B* running until a message for *R* arrives.

When that message does arrive, there are two ways of making sure *R* will meet its deadline: One is to make the scheduler check for incoming messages often enough for it to schedule *R* in time to meet its deadline. The other is to make sure *R* is alerted by the arrival of the message. If *R* has deadline d_R and cost c_R, the scheduler, in the first scenario, must react within $d_R - c_R$ seconds after the arrival of a message for *R*; that means a frequency of $1/(d_R - c_R)$. In the other scenario, the scheduler need not set any timers at all, while the core or device that delivers the message must interrupt the receiving core (i.e., scheduler) within $d_R - c_R$ seconds after delivery of the message.

But things are a tad more complex than that. If the message were for a best effort process, or a real time process with a very distant deadline, and a real time process with a very near deadline is running, then the interrupt generated by the arrival of the message may cause a deadline to be missed, rather than helping to meet deadlines. For this reason, a scheduler exports information to the other cores about the deadlines of processes for which messages may arrive, as well as the setting of its timer. When delivering a message to a core, the delivering task can now check whether or not the receiving core should be interrupted.

The mechanisms for doing this are embedded in Osprey's message passing and delivery system. In general, a message from a user space thread on one core is delivered to a user space thread on another by routing the message to the sender's kernel, from there, to the receiving core's process scheduler, then to the receiving process' thread scheduler and, finally, to the thread itself. In other words, messages to an active entity, such as a kernel task, a user process, or a user thread are always routed via their schedulers, allowing those schedulers to make appropriate scheduling decisions.

Messages, as stated, are delivered to *queues*. If a queue is an active entity's input queue, messages are

routed there as shown. More generally, messages are always routed to queues and, if a queue is directly addressable, the delivery path is direct (the *sender* deposits the message in the receiver's queue).

All queues are constructed to allow reception to be independent of delivery (no critical sections exist for sender and receiver). However, if there are concurrent senders, locking is needed to prevent one sender from stepping on another. The same applies to concurrent receivers, but to date, we have not found a need for having them.

For efficient intercore message passing, we currently use a queue for each source/destination pair of cores (i.e., n cores have n^2 queues). This arrangement will, however, be architecture dependent. We expect that future multicore processors will no longer necessarily have a memory architecture that allows each pair of cores to share memory directly. We may, however, see hardware mechanisms for efficient intercore communication that Osprey can use. For this reason, we do specify the message-passing mechanism in terms of interface, but not in terms of implementation.

Networking

Three issues influence the way in which networking is implemented in Osprey.

1. Applications must be able to process network data at network speeds of at least 10 Gb/s. Data copying should be avoided and context switches should be kept to a minimum.
2. Mobility must be supported as much as possible. This implies that when processes relocate, any protocol state should relocate as well (and protocol implementation should be mobility-aware).
3. The network interface is shared and this implies two types of protection: a) protection against one process intercepting or generating data for another's connection, and b) protection against one process using network resources reserved by another.

The central component of the network infrastructure in Osprey is a packet multiplexer. The multiplexer is controlled by a network resource table that also defines the network resource budgets of the applications.

An entry in the network resource table is depicted in **Table I**. For each supported protocol, the protocol

Table I. Sample entry in a network resource table.

Network	Ethernet Source MAC address Destination MAC address			
Protocol	IPv4 TCP/UDP	IPv6 TCP/UDP	AoE	πP Client/Server
Protocol parameters	Source IP Destination IP Source port Destination port		LUN	Server ID Client ID
Resource	Size Max MTU		Real time/best effort Period Packets Bytes	

AoE—ATA over Ethernet
ATA—Advanced Technology Attachment
IP—Internet Protocol
LUN—Logical unit number
MAC—Medium access control
Max—Maximum
MTU—Maximum transmission unit
TCP—Transmission Control Protocol
UDP—User Datagram Protocol

data determines whether and what packets may be sent/received by the process associated with the entry. The resource section of the table determines the packet size and scheduling priority for outgoing traffic. (Incoming traffic cannot be scheduled).

Besides IP-based traffic, Osprey supports special protocols such as ATA over Ethernet (AoE) for accessing disks over Ethernet and the πP file access protocol. Typically, separate ring buffers are used for these special protocols in which jumbo packet payloads are aligned on page boundaries for copy-less delivery to receiving processes.

The packet multiplexer is designed to deliver incoming packet data directly to the receiving process, via no more than a single wakeup. The receiving process must copy the data (if it needs to be copied) and restock the multiplexer's receive buffer supply.

Outgoing packets are queued for the multiplexer to send based on the priorities determined by the network resource table. The multiplexer also checks if the packet header matches the network resource table entry so that rogue processes cannot intrude on other processes' communication.

Setting up the network resource table requires coordination with already existing entries. New entries may not intrude on reservations for existing entries, both in terms of bandwidth resources and protocol/address resources. The network resource table is, of course, closely related to other types of resources and should thus be viewed as part of the general resource budgets allocated to applications. These are the subjects of the next section.

The packet multiplexer performs no protocol processing whatsoever; that is left to applications. Two ways of protocol handling appear obvious: One is to use protocol processing servers—a TCP/IP server, for example—and the other is to let each process do its own protocol processing.

Although both are possible in Osprey, we advocate the latter—for several reasons:

1. When communicating processes contain their own protocol stack, mobility and process migration become much easier to handle. It makes processes more self contained and thus simpler to detach from one host Osprey and reattach to another.
2. Faults can be handled by the application: when connections break, a user space stack gives the application control over restoration of the connection and retransmitting any unacknowledged data in a way that can be hidden from the rest of the (possibly legacy) application.
3. Protocol processing in one place—be it in the kernel or in a user space protocol server—brings with it the difficulty of charging the cycles consumed for a particular application to that application's resource budget. By making the applications do their own protocol processing, any processing cost is automatically charged to the correct budget.
4. As shown in the section on Library operating systems, user space protocol implementations can outperform kernel-based ones because of the elimination of coupled servo loops. This is yet another reason for placing the protocol implementations in user space.

Resource Budgets

Resource budgets specify what applications can use in terms of processors, memory, and input/output (I/O). The exact specification of these budgets is still under investigation. One goal is to come up with a way to allow a meaningful combination of budgets for the sub-applications in a large distributed application running on multiple machines.

As an example, video conferencing applications running on participating hosts, as well as the network connecting these hosts, need their budgets specified in a way that gives the video conference the intended bandwidth and latency and allows any video/audio processing to be done in real time.

At the moment, the processor budget consists of a per-core real time or best effort allocation. The best effort allocation is simply in terms of a base priority. A process' priority is calculated by subtracting from the base priority a value that expresses recent consumption of central processing unit (CPU) cycles. The best effort process with the highest priority is run first. The offset based on recent CPU consumption avoids starvation.

Real time computing relies on process scheduling with an awareness of the deadlines processes have to meet. Those deadlines are usually recurring: a real

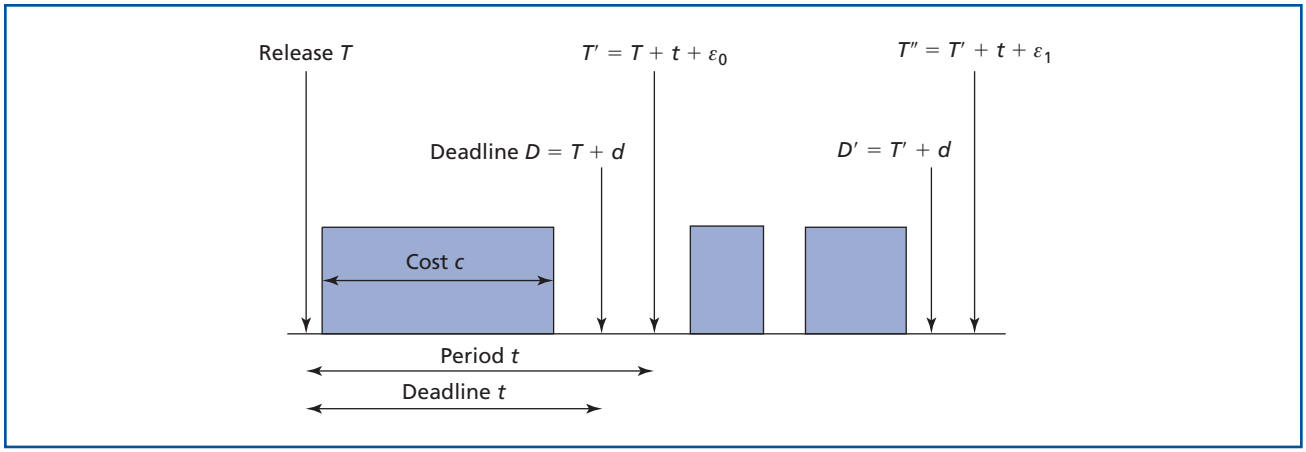

Figure 1.
Parameters characterizing a real-time process; note that intervals are indicated by lower-case letters, while times are indicated using capitals. Periodic process are scheduled precisely every t seconds ($\epsilon_j = 0$); sporadiac process are scheduled at least t seconds apart ($\epsilon_j \geq 0$).

time process, for instance, must process a video frame precisely every 30 ms. They are typically characterized by a period t, a deadline d, and a cost c, as illustrated in **Figure 1**. The process is released (at most) every t seconds and may then consume up to c seconds of CPU time before the deadline, d seconds later ($c \leq d \leq t$). For n real time processes on a given CPU, $\sum_{i=0}^{n} c_i/t_i < 1$. In fact, if d = t and if processes can be preempted, this is a sufficient condition for schedulability [13].

Guaranteeing that processes meet their deadlines when there are additional constraints (e.g., processes must wait for others to finish something) is a complex art and the subject of much research [19]. What is of concern in this paper is that it requires precise control of a clock to interrupt the normal flow of execution when real time processes must take over, as well as knowledge of the number of processor cycles available per second.

Osprey keeps track of clock time in nanoseconds. A 64-bit signed value records nanoseconds since the *epoch*, 00:00, January 1, 2000. This allows expressing times anywhere between Thursday, 22 September 1707 and Monday, 11 April 2292. The accuracy of the Osprey scheduler is around 5 μs on a typical modern processor.

Real time processes may be scheduled *periodically* or *sporadically*; the former means that processes are released at precise intervals equal to their specified period, *t;* the latter means that processes wait for, and are released by, external events and, if such an event occurs less than t seconds after the previous, the release is postponed until t seconds have elapsed. An application that processes incoming data packets in real time would normally be scheduled as a sporadic real time process.

Adding a real time processing budget to a resource budget requires permission from the operating system, and this permission will only be granted if there are sufficient resources left to allow the real time guarantees to be given. An *admission* test is performed to verify this.

An application's memory resources are specified in *segments*. Each segment is composed of *pages* and has a size that is a multiple of the page size chosen for the segment. The range of page sizes is dictated by the processor architecture; on the Intel/AMD k8 architecture, the page sizes are 4 KB, 2 MB, or 1 GB.

Not all the pages in a segment have to be allocated a priori. Segments have a type that dictates whether, where, and how pages can be added to a segment. Segments, however, can never grow beyond their initially specified size, and thus the set of segments for an application specifies the memory budget of that application.

The processes that form the application may map these segments into their address space; segments in an address space may not overlap.

Segment sharing is heavily used in Osprey for efficient interprocess communication. Control over sharing is provided to a segment's owner (i.e., the owner of the budget from which the segment was allocated), via an access control list, as if the segment were a file. Segments, in fact, are usable as files, so this is no surprise.

For the time being, the I/O bandwidth budgets are just network bandwidth budgets. They were discussed in the previous section. We are still discovering what the best expression of network bandwidth reservations must be, and at the moment, we are using *best effort* and *real time* reservations. The real time reservations specify a *period* and the maximum number of packets and/or bytes for which a guarantee to send can be given. This implies that with a given bandwidth in bytes, one cannot expect a guarantee to send the bytes one byte per packet; or with a reservation of a large number of packets with a moderate number of bytes, one cannot expect to be able to send the maximum number of maximum-size packets.

Longer term, we are looking for ways to express *CPU* and network reservations in a way that makes them *composable* in the sense that a number of collaborating machines can each get reservations that match their needs and that the combined reservations form a working one for the distributed application as a whole.

An issue that hasn't been addressed at all yet is formulation of resource reservations suitable for weathering failures in real time. This is a subject for current study.

Library OS

At the heart of Osprey's cloud support is the notion of the Library OS. The Library OS idea is not new. As we reported earlier, a Library OS has been used in the Exokernel and for supporting Java* virtual machines (JVMs). Osprey uses a Library OS to relieve the operating system kernel from maintaining user process state. In fact, a Library OS achieves several important goals for Osprey:

1. Per-process operating system state is minimized.
2. Different operating system interfaces can be supported on a single Osprey platform.
3. Library OS support tailored to the operating system interface can provide much better performance than matching one full-fledged OS interface to another, as is the case for Wine* [26].

These three points make Osprey eminently suitable for cloud computing. The overhead of a virtualized operating system's extra layers of indirection— two levels of memory management and process scheduling—significantly affect performance.

Naturally, the Library OS needs a system interface that allows it to do its job. Osprey cannot be completely devoid of system calls. The system calls present are related to process creation and destruction, to memory management, to interprocess communication, and to resource budgets. We'll discuss these categories one at a time.

Processes are created and managed with so-called *process descriptors* (PDs). Process descriptors were inspired by the Amoeba Distributed System [17] which also used them for process management. A PD is depicted in **Figure 2**.

The figure depicts a data structure that contains all of a process' state except the contents of its memory. For each memory segment in a process' address space, the PD contains the type and size, and where it's mapped. The *content reference* behaves like a file name and can be used by the receiver of a PD to read the contents in memory like a file.

Before a program can be run, its representation is converted (from, e.g., executable and linking format (ELF) format) into a PD. Content references, for convenience, may contain a name plus an offset allowing a PD plus the initial content of a program's text and data segment to be stored in one file.

The PD is then sent to the process manager server on an Osprey core, where a process can then be created and initialized. The memory segments are created, and depending on the type of process or memory segments, the segments can be read in beforehand, or they can be *paged in* (or *faulted in*).

The process owner may *stop* a running process. Its hosting process manager will then produce a PD describing its state. The content references for immutable segments will be the original ones; those

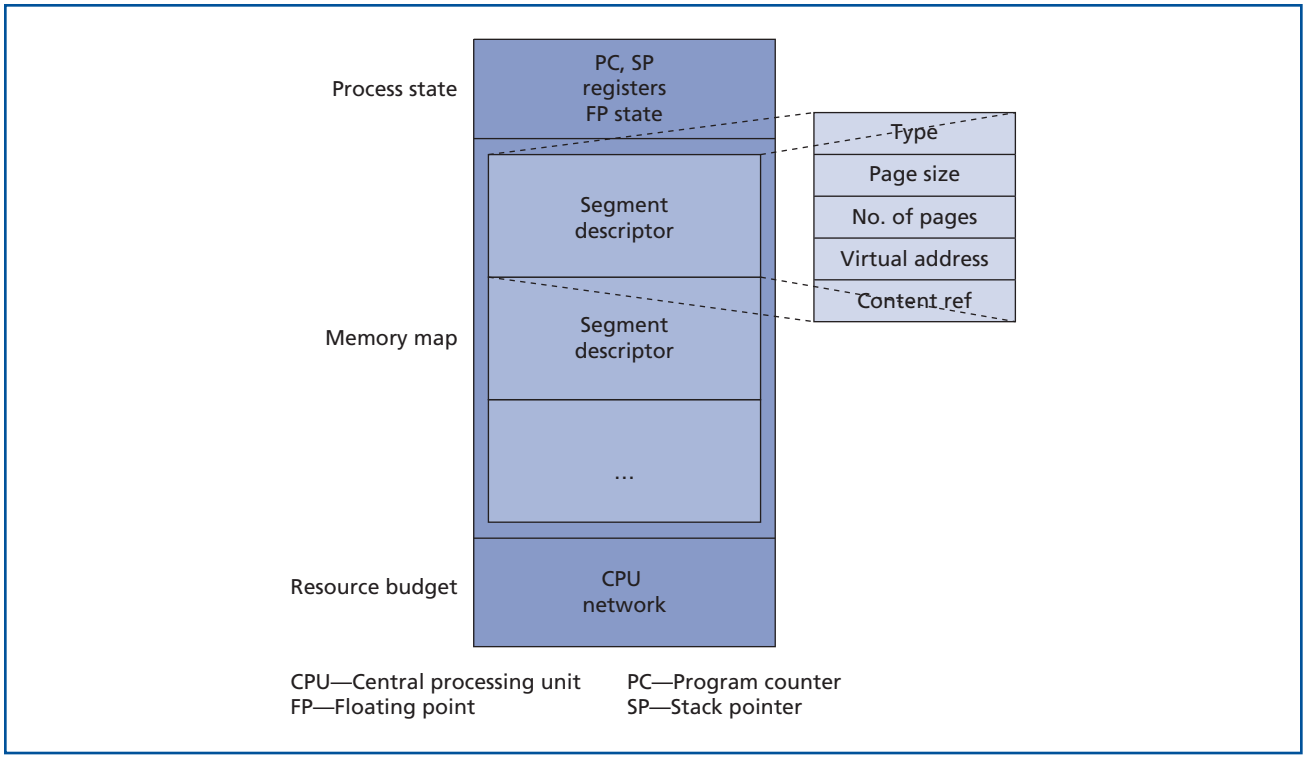

Figure 2.
Process descriptor.

for modified segments will be names served by the process manager as if they were files.

Segments are, in fact, treated as files: they have owners and access control lists and these are consulted to determine whether memory sharing will be allowed (or whether segments can be used as memory-mapped files).

A PD produced by a process manager behaves just like any PD and can be given to another process manager. If the original process is then deleted, the procedure amounts to process migration. If the original is allowed to continue, it amounts to *forking*.

The self-contained nature of processes in Osprey allows them to exert full control over their own fault tolerance. A process can choose the moment and then checkpoint itself by receiving its own PD.

Status and Conclusions

Osprey is still under development, but large parts of it are now up and working on Intel/AMD 32-bit and 64-bit platforms. Processes can run, send, and receive messages; schedule themselves in periodic or sporadic real time; and there is a completely untested Library OS for Plan 9® applications. On 2 GHz machines, the scheduling accuracy is under 10 μs and context-switching times are 1 μs for kernel tasks and a little more for user processes.

When it becomes stable, Osprey code will be made available under an Apache license.

This paper described the design of an operating system that can handle dynamically distributed workloads and operate in real time in a fault tolerant manner. We discussed the low-level operating system design, but have not touched upon the actual realization of fault tolerance, nor have we discussed security. These are active areas of research in the group and we will publish on these topics separately.

To date, we are very satisfied with a design that runs more or less separate copies of the operating system on each core. It significantly simplifies the kernel design—there is much less concurrency to worry about. The places where cross-core coordination is

needed are easily identified—e.g., free memory and address space shared cross-core, message routing and delivery, load balancing—and some of these can be delegated to user level operating system servers.

We are exploring the possibilities of sharing cores between Osprey and Plan 9. The idea started in the HARE/FOX high-performance computing project (where it was called *Nix*; Nix is described elsewhere in this issue), and, in our context, it boils down to this: An almost unmodified copy of Plan 9 boots up on one of the cores and then, perhaps, takes a few more cores for its own use. It then allocates memory for Osprey's use and boots up Osprey on the remaining cores. Osprey tasks and processes communicate in the usual way with processes on Plan 9: by sending and receiving messages. These messages can be used to access devices and files on the Plan 9 system. We are thinking of using this to give Osprey access to devices and file systems more quickly.

Acknowledgements

We gratefully acknowledge the contributions of the entire Osprey team: Utku Günay Acer, 安雪莉 (Xueli An), Tom Bostoen, Davide Cherubini, Noah Evans, Franck Franck, Eric Jul, Jim McKie, Jeff Napper, Fabio Pianese, Jan Sacha, Henning Schild, and Tom Wood.

*Trademarks

Facebook is a trademark of Facebook, Inc.
Google and YouTube are registered trademarks of Google, Inc.
Inferno is a registered trademark of Vita Nuova Holdings Limited.
Java is a trademark of Sun Microsystems Inc.
Kindle is a registered trademark of Amazon Technologies, Inc.
Libra is a registered trademark of Agilent Technologies, Inc.
Linux is a trademark of Linus Torvalds.
Skype is a registered trademark of Skype Limited.
Twitter is a registered trademark of Twitter, Inc.
Unix is a registered trademark of The X/Open Group.
VMWare is a registered trademark of VMWare, Inc.
Windows is a registered trademark of Microsoft Corporation.
Wine is a registered trademark of Software Freedom Conservancy.

References

[1] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski, "Libra: A Library Operating System for a JVM in a Virtualized Execution Environment," Proc. 3rd Internat. Conf. on Virtual Execution Environments (VEE '07) (San Diego, CA, 2007), pp. 44–54.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," Proc. 19th ACM Symp. on Operating Syst. Principles (SOSP '03) (Bolton Landing, NY, 2003), pp. 164–177.

[3] Barrelfish, <http://www.barrelfish.org>.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," Proc. 22nd ACM Symp. on Operating Syst. Principles (SOSP '09) (Big Sky, MT, 2009), pp. 29–44.

[5] P. Bosch and S. J. Mullender, "Real-Time Disk Scheduling in a Mixed-Media File System," Proc. 6th IEEE Real-Time Technol. and Applications Symp. (RTAS '00) (Washington, DC, 2000), pp. 23–33.

[6] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," IETF RFC 2460, Dec. 1998, <http://www.ietf.org/rfc/rfc2460.txt>.

[7] A. Edwards and S. Muir, "Experiences Implementing a High Performance TCP in User-Space," Proc. Conf. on Applications, Technol., Architectures, and Protocols for Comput. Commun. (SIGCOMM '95) (Cambridge, MA, 1995), pp. 196–205.

[8] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney, "Fast and Flexible Application-Level Networking on Exokernel Systems," ACM Trans. Comput. Syst., 20:1 (2002), 49–83.

[9] W. Heisenberg, "Über den Anschaulichen Inhalt der Quantentheoretischen Kinematik und Mechanik," Z. Phys., 43:3-4 (1927), 172–198.

[10] V. Jacobson and B. Felderman, "Speeding Up Networking," Linux.conf.au (LCA '06) (Dunedin, Nzl., 2006).

[11] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," IEEE J. Select. Areas Commun., 14:7 (1996), 1280–1297.

[12] H. Li and K. C. Sevcik, "Exploiting Cache Affinity in Software Cache Coherence," University of Toronto, Computer Systems Research Institute (CSRI), Tech. Report 299, Apr. 1994.

[13] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," J. ACM, 20:1 (1973), 46–61.
[14] D. Moore, C. Musciano, M. J. Liebhaber, S. F. Lott, and L. Starr, "'GOTO Considered Harmful' Considered Harmful" Considered Harmful?," Commun. ACM, 30:5 (1987), 351–355.
[15] G. Moore, "Excerpts from a Conversation with Gordon Moore: Moore's Law," Intel, Video Transcript, 2005, <http://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf>.
[16] G. E. Moore, "Cramming More Components onto Integrated Circuits," Electron., 38:8 (1965).
[17] S. J. Mullender, "Process Management in a Distributed Operating System," Experiences with Distributed Systems (J. Nehmer, ed.), LNCS 309, Springer-Verlag, Berlin, Heidelberg, New York, 1988, pp. 38–51.
[18] S. J. Mullender, "Interprocess Communication," Distributed Systems, 2nd ed. (S. J. Mullender, ed.), ACM Press/Addison-Wesley, Wokingham, Eng., Reading, MA, 1993, Chapter 9, pp. 217–250.
[19] S. J. Mullender and P. G. Jansen, "Real Time in a Real Operating System," Computer Systems: Theory, Technology, and Applications—a Tribute to Roger Needham (A. Herbert and K. Spärck Jones, eds.), Springer-Verlag, New York, 2004, pp. 213–222.
[20] S. J. Mullender, I. M. Leslie, and D. McAuley, "Operating-System Support for Distributed Multimedia," Proc. USENIX Summer Tech. Conf. (USTC '94) (Boston, MA, 1994), vol. 1.
[21] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba—A Distributed Operating System for the 1990s," IEEE Comput., 23:5 (1990), 44–53.
[22] J. Postel (ed.), "Internet Protocol," IETF RFC 791, Sept. 1981, <http://www.ietf.org/rfc/rfc791.txt>.
[23] T. Roscoe, The Structure of a Multi-Service Operating System, Ph.D. Dissertation, Queens' College, University of Cambridge, 1995.
[24] F. Rubin, "'GOTO Considered Harmful' Considered Harmful," Commun. ACM, 30:3 (1987), 195–196.
[25] J. F. Shoch, "Inter-Network Naming, Addressing, and Routing," Proc. 17th IEEE Conf. on Comput. Commun. Networks (COMPCON '78–Fall) (Washington, DC, 1978), pp. 72–79.
[26] WineHQ, "Wine," <http://www.winehq.org>.

(Manuscript approved February 2012)

SAPE J. MULLENDER is director of the Network Systems Lab at Bell Labs in Antwerp, Belgium. He has worked extensively in operating systems, multimedia systems, and, in recent years, wireless systems research. He was a principal designer of the Amoeba distributed system; he led the European Union's Pegasus project, which resulted in the design of the Nemesis multimedia operating system; and he made valuable contributions to work on the Plan 9® and Inferno® operating systems. He received a Ph.D. from the Vrije Universiteit in Amsterdam, The Netherlands, where he also was formerly a faculty member. He currently holds a chair part time in the Computer Science Department at the University of Twente. Dr. Mullender has published papers on file systems, high-performance remote procedure call (RPC) protocols, migratable object location in computer networks, and protection mechanisms, and he was involved in the organization of a series of advanced courses on distributed systems. ◆