From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud

Nikos Vasilakis, Ben Karel, Jonathan M. Smith
The University of Pennsylvania

Abstract

Unix took a rich smorgasbord of operating system features from its predecessors and pared it down to a small but powerful set of abstractions: files, processes, pipes, and the shell to glue the system together. In the intervening forty years, the common-case computational substrate has evolved from a lone PDP-11 minicomputer to vast clouds of virtualized computational resources. Contemporary distributed systems are being built by adding layer upon layer atop the foundation established by Unix's chosen abstractions. Unfortunately, the resulting mess has lost the "simplicity, elegance, and ease of use" that was a hallmark of the original Unix design [24]. To cope with distribution at astronomic scale, we must take our operating systems back to the drawing board. We are living in a new world, and it is time to be brave.

1 Introduction

The June 1986 edition of Jon Bentley's Programming Pearls column featured two contrasting solutions to a simple word-counting problem [3]. With the goal of illustrating Literate Programming style, Don Knuth presents a Pascal program on the order of 100 lines, carefully crafted "from scratch". Doug McIlroy's review notes that the problem could have been solved with a simple Unix pipeline:

```
tr -cs A-Za-z'\n' | tr A-Z a-z | sort
     | uniq -c | sort -rn | sed ${1}q
```

In McIlroy's view, it was no coincidence that Unix was up for the job: the utilities at hand were simple, modular building blocks, designed for composition, which captured useful steps extracted from real problems.

The modern incarnation of Bentley's problem would involve orchestrating a fleet of machines to process a corpus that exceeds any single machine's capacity. What tools can the contemporary programmer bring to bear?

Our first candidate would be a distributed processing framework, such as Hadoop. But setting up virtual

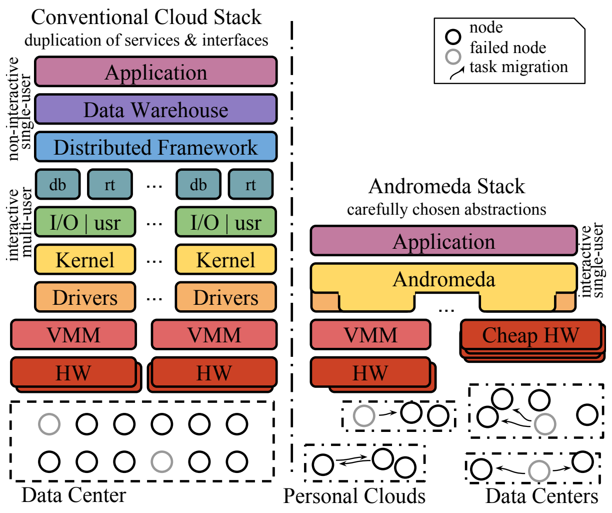

Figure 1: High level differences between conventional cloud workload setups (left) and Andromeda (right).

machines, configuring Hadoop on them, expressing the computation in terms of the particular primitives (*i.e.*, map, reduce), configuring the job, taking care of file systems are all excessively tedious. Complexities of this nature are not inherent to Hadoop itself (for instance, more recent frameworks such as Spark [32], DryadLINQ [31] or Naiad [21] have similar difficulties). High scalability today comes at steep cost, because *complete, transparent de-centralization was never a concious design decision or inherent feature of present systems* (Fig.1 – left).

What we sorely lack is a distributed *foundation* upon which to build a solution as concise and elegant as McIlroy's pipeline. To change that state of affairs, we sketch our vision for a massively distributed operating system, designed afresh for the commodity cloud. As a parallel to the scale of distribution the system is targeting, we name our project *Andromeda* (M^{31} for short), after the Messier 31 galaxy. Andromeda is designed *de novo* to match the computational environment of the not-too-distant future, taking advantage of many promising and underexplored ideas from the past, morphing them synergistically with some of its own. Contrary to current systems, in which developers are required to retrofit new models to existing software architectures, it assumes distribution, mobility, fault-proneness, extreme bandwidth, low but non-negligible latencies, and dynamic provisioning. It takes extensive care, however, in making their implications transparent to the application layer.

Before delving into the details of the system, it is worth noting the constraints and factors that have guided its design. First, the forthcoming *data cataclysm*. Our position is that the current model is broken more deeply than may appear at first blush. Simply adding to the existing pile of incremental changes or doing minor tweaks at the highest levels is not enough for even the most conservative forecasts. Second, the increasing reliance of cloud computing infrastructure on *commodity* hardware: inexpensive, unreliable nodes, that trade local guarantees for global properties. These cause a shift of assurance from the hardware to the software layer, requiring redesign of the software stack under different assumptions. This leads to a third development, the rise of the *personal cloud*, opposing the old idea that users will own only dumb terminals connected to a datacenter – they will own multiple micro-clouds. And fourth, the shift of data processing from experts to ordinary people in all fields (*e.g.*, biologists, economists, anthropologists) using it for their everyday tasks. Systems of the future will need to take care of all the gory details and expose the cleanest possible abstractions. Assuming these trends, the question becomes, what kind of mechanisms should tomorrow's ideal operating system provide, to allow a solution of similar simplicity to the one presented three decades ago?

2 Nucleosynthesis of Andromeda

Here we sketch the elements composing Andromeda, as selected by our intuition, and reflecting three themes that characterize the Andromeda vision: subsumption of OS mechanisms with linguistic abstractions, transparent distribution, and a focus on interactive programming.

Fully-Transparent Object Distribution As is usually the case in a distributed operating system, Andromeda takes care of distributing and replicating state across nodes. It does so *transparently* and *homogeneously*. Transparency means that the distribution and replication mechanisms remain invisible to the user, who has only a unified view of the underlying resources, as if they were part of a sole, centralized, single-core machine. Homogeneity means that all interconnected machines forming a constellation export the same interfaces (even down to particular versions), and this is reflected in the M^{31} substrate. When a user connects into an Andromeda instance, there is no distinction between local and remote data – everything feels local.

Of course, there exist workloads that are better suited for a particular type of machine, and in our model these machines form a dedicated constellation. For instance, GPU-intensive workloads would be located within a constellation augmented with GPUs, and could be relocated to a uniform constellation right after completion. To allow the properties mentioned above, there is an implicit split between *nucleus*, the low-level, inner, per-node kernel, and *mantle*, the high-level, spatially continuous, global substrate with which the user interacts.

Pico-kernel The nucleus comprises of the absolute minimum core, the language runtime, required to service the mantle layer. It is minimized beyond traditional microkernels, in which the OS takes care of process management and scheduling, by enabling shared address spaces, lightweight processing primitives and cooperative scheduling. While it is closer to exokernels in the sense that it avoids kernel mediation as much as possible, contrary to them, it does not expose hardware resources nor does it allow for applications to define their own interfaces on top of it – upper layers can only make use of a set of well-defined, carefully-chosen abstractions. It is little more than a JIT compiler, receiving code and data through active packets from other nodes, and, combining with local libraries and data, translating down to machine code, and executing them.

Low-Overhead Execution Primitives Current cloud workloads require provisioning of hundreds or even thousands of execution primitives (*i.e.*, processes, threads) in a very short amount of time. Some of them are short-lived while others keep processing continuously. Andromeda provides *fibrils*, carefully-chosen primitives of execution that are thin and lightweight. Fibrils are control units with their own private, dynamically allocated stack, and private data . They are cooperatively scheduled at user level, allowing traditional interrupt and resume without incurring context switches, at an overhead comparable to a function call, and communicate through messages. Pipe- or rendez-vous-like synchronization comes at a much lower cost, without incurring conventional kernel overheads (*e.g.*, setting up FIFOs, buffering and producer/consumer synchronization).

Fibril Migration In a galaxy of interconnected machines, one increases fault tolerance and reduces access latency by migrating a processing primitive and the data it processes closer together. Since moving data around can easily saturate network links in a data center (and, with certainty, in the case of a personal cloud!), Andromeda offers transparent, non-preemptive *fibril migration* across machines or processors. Fibrils are much cheaper to relocate or clone than whole datasets, and can safely move along with all their associated state into an entirely different node for execution. Co-location improves data locality, thus increasing overall performance, and replication improves robustness in the case of failing nodes. Aforementioned node homogeneity and data privacy make state migration easier and safer. Naturally, migration policies are not set in stone by the nucleus-but are subject to user requirements; for instance, when needed, fibrils are allowed to anchor to a specific node or processor.

Message Passing Shared memory suffers from many shortcomings in a massively-distributed environment: poor multi-core scalability due to lock contention [2, 13], and difficulty of reasoning from the programmer's perspective [1, 28]. M^{31} offers *channels*, one-way FIFO primitives that facilitate both local and remote communication. Since no structures are shared, channels safely allow scalability to thousands of nodes, and blur the distinction between local and remote execution. They come in both synchronous and asynchronous versions, and their endpoints are strongly typed – data exchanged have their types agreed upon beforehand, allowing error catching and runtime optimizations. Cloning endpoints, handled opaquely by the runtime, allows for paradigms of multiple senders or receivers. Moreover, enabling channel and fibril migration through channels themselves, similar to theoretical process calculi [29], enables powerful and well-studied programmatic expressiveness.

Scheduling In such a distributed setting, each node has to take into account global decisions, while at the same time conduct local optimizations (*e.g.*, masking hardware heterogeneity, I/O latencies). Andromeda utilizes *layered* (hierarchical) scheduling [11, 5], in which decisions are made using a combination of global, distributed resolution and local, cooperative judgement. Pushing global decisions to collaborating nodes with a holistic view of the constellation allows efficient and tunable high-level, mid-term decision-making while low-latency micro-management is left to the nucleus layer. For instance, fibril migration requires coordination at the global plane, but the coordination mechanism itself running on each node makes use of local, cooperative scheduling. Linguistic abstractions allow the kernel to resolve possible dependencies between tasks, and make it easy to alter scheduling policies on the fly based on global knowledge.

Flat, Labeled File System Most previous attempts on distributed operating systems assumed hierarchical file systems, with ACID guarantees often built on top of them; today's requirements, though, caused a shift towards flat file systems [27]. Indeed, to allow data scaling, query efficiency and fault tolerance, Andromeda offers a distributed, flat, *labeled* key-value store. Data are stored as typed objects (as opposed to, say, tables or text files), the same abstraction used for interchange formats and linguistic constructs. These are distributed and replicated across multiple nodes, and can be queried efficiently even on their secondary attributes. Objects can have unlimited labels attached, namely arbitrary meta-data similar to permissions and access times in traditional file systems, that follow data along (and, if needed, can emulate hierarchical structure). Apart from fast retrieval due to relaxed schemas, and convenience due to utmost freedom, flat labeling allows for easy migration and conditional replication. Andromeda borrows ideas from *hyperspace hashing* [9] (the so-called second generation key-value stores) to create a multidimensional object space, one per query-able dimension, with different types residing on different subspaces, but adds support for transparent object versioning and version branching.

Naming In a setting where system state is decentralized, virtual processors migrate and nodes come and go non-deterministically, resource naming –the way by which local, short handles reference remote, large pieces of data– poses a considerable challenge. Post-migrated fibrils, resources and channels should be accessible through the same identification mechanisms as before (think of process IDs in Unix-land) and the identifier should include enough information to verify that dereferences produce meaningful results. Some of the issues are alleviated by the transparently-distributed, attribute-based persistence store and the identical system image. On top of these, Andromeda leverages RESTful uniform resource identifiers (URIs) [10], for both internal (local and remote) and external naming. Many of these names are resolved into virtual resources or are handed off to the persistence layer (*e.g.*, `GET` /fs/log?since=01012014&fmt=JSON). The system makes opening public interfaces to share data or services with non-Andromeda users straightforward (see also Data Interchange) and allows for naming aliases that can follow resource migration.

Implementation Language Picking the right linguistic abstractions will influence both the performance and security of the system [17] as well as the ease with which astronauts and crew will be able to fly around. To hide much of the complexity that arises from systems of this nature and scale, we envision a small, multi-paradigm, systems programming language that follows the same paradigm as the M^{31} itself – small core, but trivially extensible semantics. Indeed, Andromeda as a whole is structured around the language in the same way Unix is structured around C [13]. It emphasizes safety and supports first class functions and objects, since both paradigms fit together with the persistent store and interchange formats, and express conveniently typical high-level processing primitives. It provides strong, static typing (but allows optional dynamic type checking when such is desired *e.g.*, when used interactively), favors immutability and data copying (but allows handing off pointers if particular promises are kept), and provides mechanisms for containing side-effects.

Sandboxing Traditional isolation mechanisms based on address space segregation incur high overhead, an overhead that becomes prohibitive when thousands of processing primitives need to collaborate with each other to provide services across commodity hardware. The nucleus provides efficient software fault isolation using linguistic mechanisms (*e.g.*, type system, scoping and binding rules, function closures (environments), protected calls) to contain side effects. Most of these are enforced during compile time, while some checks are done during runtime; the user has a limited freedom of moving some of these checks between compile time and runtime. With the kernel architecture as described earlier, all device drivers run at the mantle, in their own fault domains, and communicate through messages. Since manual memory management introduces many bugs and vulnerabilities that, in the case of a distributed system, affect a shared and extended pool of resources, the language runtime offers automatic memory management in the form of garbage collection. The subsystem responsible for resource reclamation takes care of collecting conventional, local objects (*e.g.*, variables, fibrils) as well as objects from mantle strata (*e.g.*, open remote connections, channels).

Data Interchange In a system where data and code are often in-flight between nodes, data interchange becomes vital, and a usual hot-spot for problems. Echoing Dan Bernstein's concerns [4], parsing and converting unstructured blobs into structured data is usually a recipe for disaster. Andromeda features object serialization and interchange baked into the language, with the format being the object constructor itself (similar to Lua's "BibTeX" [14], Lisp's S-Expression, and JavaScript's JSON [8] formats). Contrary to CSV or XML, the piece of data itself is a valid program – if it compiles, it is parse-able, and can be manipulated and introspected as a first class citizen. Receivers can directly invoke methods of the objects transfered or retrieved. The format is human-readable, self-describing and light-weight. As an added benefit, Andromeda can optimize interchange loads using a compressed abstract syntax tree (AST) encoding of the above format [6].

Programming Interface Contrary to existing distributed operating systems, Andromeda focuses on promoting distributed computing to fully *interactive* use. Users interact with M^{31} using a REPL that interprets exactly the same strongly-typed, higher-order programming language that components and user applications are written in. Somewhat contrary to `fork`/`exec` Unix (and, subsequently, POSIX) primitives, it allows dynamic module loading, reflection and fibril spawning, enabling the user to load libraries and execute programs in a unified interface. Users can compose and overload simple processing primitives, introspect data objects and interact with the file system. They can also set up, enter and exchange sandboxes dynamically, and channel results through synchronous and asynchronous primitives. Environments initiated in this manner inherit a small set of default callbacks that can be augmented or overridden.

Everything under the sun? The discussion above tries to distill key ideas in terms of the mechanisms and abstractions provided by M^{31}, omitting many interesting details (*e.g.*, node authentication, object versioning, programming libraries), policies (*e.g.*, consensus, coordination, scheduling) and optimizations (*e.g.*, distributed memory-caching schemes, internal networking protocols). But let us return to the problem of counting words.

3 Back to Frequencies

In Fig. 2, we are working on extracting word frequencies from thousands of books hosted on Project Gutenberg [12]. We use a pseudo-Pythonic syntax only to have something concrete to discuss. Also, to better explore Andromeda's internals, we do not present a pretty one-liner, but rather the primitives that can be easily wrapped to allow the one-liner.

Postponing the discussion of `main` until we cover some preliminaries, we initially configure the distributed storage for words (*e.g.*, how many failures to tolerate, if we need data versioning) and run the equivalent of `mkdir`. This takes the configuration object as an argument and picks the user's default options when parameters are missing. The value of `attrs` is itself an arbitrary object. Since we need to inform the file system about the type of values to be stored, we pass a type that can be used to retrieve object interfaces or constructors for destructuring incoming objects.

After storage is set up, we spawn a fibril that will start crawling the project page intended for robots, download books and calculate frequencies. The `spawn` primitive wraps its arguments (a function and the function's arguments) in the interchange format discussed previously and returns a channel endpoint pair (at the same time, the new fibril itself gets the other parts of the pair). The receiver can be monitored for updates and the sender can be used to send data to the child, in a possible analogue to Unix processes inheriting standard streams. Using a limited form of static analysis, the runtime communicates this package containing code and data to a node that is highly probable to contain related data. The wrapper object also encodes any channel endpoints required for communication, and after migration negotiation, upon acceptance, both nodes make their respective registrations.

We can now dissect `main`. We already know that this function is going to be compiled and executed remotely,

```
fn main(url: String):
  import http, fs, sys, jam
  res = http.req(url)
  match res.contentType:
    "application/zip" =>
      book = jam.unzip(res.content())
      for b in book.split():
        fs.invoke({word:b, freq:Int.incr)
      PARENT.send(sys.SUCCESS)
    "text/html"     =>
      cs = []
      for url in res.urls():
        cs.append(sys.spawn(main, [url]))
      PARENT.send(sys.collect(cs))
    _               =>
      PARENT.send(sys.FAILURE)

words = { key: "word", value: {"freq": Int}
    , opts: { partitions:8, failures:2
          , versioning: False}}
fs.addSpace(words)
seed = "//gutenberg.org/robot/harvest"
s,r = sys.spawn(main, [seed])

match r.recv():
  sys.SUCCESS => fs.search({word:"pi"}
  _           => ("Impossible", -1)
```

Figure 2: Word frequencies (a typical "hello, world!" program among distributed frameworks) on Andromeda.

so we need to include particular imports (everything is loaded on the fly). It first downloads the file and checks if it's a book (zip) or a url containing books. In the first case, it just processes each word in the file and signals completion. `Int.incr`, and generally any `fs` invocation, is guaranteed by the runtime to run atomically on every record. In the second case, it spawns more fibrils and ships the required parts accordingly (data and fibril id's). In particular, there is no need to resend a function body, since the runtime has already registered this particular version across nodes that, when presented with the function's fingerprint, can retrieve it accordingly. In any case, the fibril communicates its status with the parent, with the root blocking until all computation is complete. The runtime makes sure to reschedule any failing or unreachable computation. When complete, we just ask for the frequency of the noun "pi".

4 Related Work

There has been a significant body of work on distributed operating systems dating from the 1970s [20, 23, 22, 15, 30], but most of this work never managed to blossom beyond academic interest. Although the exact reasons might be different for each project's fate, one can clearly identify some of the causes. Most of these systems lacked the maturity that systems nowadays enjoy, raising the barrier of adoption, given that they were already providing different abstractions from existing systems. Moreover, network speeds were only a fraction of processing speeds, and while a lot of effort was put into masking latencies and lack of bandwidth, users of these systems never experienced today's luxury. Most importantly, however, *there was never a desperate need for such systems*. Only today have workload requirements motivated the properties pursued by these systems.

Since the environment has changed, it is reasonable to challenge the decisions made in the past, or at least evaluate them carefully – after all, it has been more than twenty years since most of these systems were designed! For instance, due to resource cost, and therefore scarcity, many of these systems assumed batch, non-interactive workloads on non-uniform clusters, and designers chose to expose such heterogeneity across nodes. A more important difference is that older systems were designed with much more powerful, relative to contemporary, hardware in mind whereas Andromeda is specifically designed for extremely cheap, failure-prone computational substrate. This puts the main responsibility on the software layer, whose complexity needs to be abstracted away from developers and users. Deliberate, precise abstraction is becoming vital, due to the ongoing shift towards third-party, non-systems programmers. Moreover, Andromeda is the only system that explicitly targets distribution at scale, and puts a conscious effort in handling the complexities introduced by unwieldy, astronomical growth.

Over the past few years, researchers have made similar observations about workload needs and user trends, and the necessity of a scalable, distributed, minimal operating system or language runtime [17, 26, 18, 25, 13]. Moreover, they identified that such systems require new linguistic paradigms and operational abstractions, effectively dropping varying degrees of backwards compatibility. Surprisingly, however, the community has not seen many implementations, and even these typically tackle only a subset of the issues Andromeda aims to address. Osprey [25] and DIOS [26] are well in their development phase, but, to the best of our knowledge, they seem to be Linux library OSes targeted for the cloud rather than OS kernels themselves. Cluster management suites [19, 16] and container engines [16, 7] (both of which are often dubbed as cloud operating systems!) are usually just another layer added on top of the conventional stack with the goal of making application development and management easier.

5 Conclusion

In this work we try to answer an simple, old question under new lenses. Our inability to find a satisfying answer motivates a discussion about a massively distributed OS targeting commodity hardware. The features and abstractions offered by Andromeda, although not novel *per se*, act in synergy to offer sorely-needed guarantees for tomorrow's user.

Acknowledgements We would like to thank the anonymous reviewers for their comments and questions. Andreas Haeberlen, John Cotton, Leonidas Lampropoulos, Perry Metzger, Cassi O'Pea, and Yang Wu provided helpful feedback on earlier versions. This material is based upon work supported by DARPA contract FA8650-11-C-7189.

References

[1] J. Armstrong. What's all this fuss about Erlang? 2007.

[2] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *HotOS*, 2009.

[3] J. Bentley, D. Knuth, and D. McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[4] D. J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 1–10. ACM, 2007.

[5] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 4. ACM, 2013.

[6] M. Burtscher, B. Livshits, G. Sinha, and B. G. Zorn. JSZap: compressing JavaScript code. In *Proceedings of the USENIX Conference on Web Application Development*, 2010.

[7] CoreOS. Linux for massive server deployments. `http://coreos.com`, 2013.

[8] D. Crockford. The application/json media type for javascript object notation (JSON). 2006.

[9] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[10] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[11] B. Ford and S. Susarla. CPU inheritance scheduling. In *OSDI*, volume 96, pages 91–105, 1996.

[12] M. Hart. *Project gutenberg*. Project Gutenberg, 1971.

[13] D. A. Holland and M. I. Seltzer. Multicore OSes: looking forward from 1991, er, 2011. In *Proc. HotOS*, volume 11, pages 33–33, 2011.

[14] R. Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[15] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, 1988.

[16] Kubernetes. Manage a cluster of containers as a single system. `http://kubernetes.io`, 2013.

[17] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. The case for the holistic language runtime system. In *Proceedings of the 1st International Workshop on Rack-scale Computing*, 2014.

[18] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the eighteenth international conference on architectural support for programming languages and operating systems*, pages 461–472. ACM, 2013.

[19] Mesosphere. Datacenter operating system. `http://mesosphere.com`, 2014.

[20] S. J. Mullender, G. Van Rossum, A. Tanenbaum, R. Van Renesse, and H. Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[21] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[22] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[23] R. Pike, D. Presotto, K. Thompson, H. Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[24] D. M. Ritchie and K. Thompson. The unix time-sharing system. *SIGOPS Oper. Syst. Rev.*, 7(4):27–, Jan. 1973.

[25] J. Sacha, H. Schild, J. Napper, N. Evans, and S. Mullender. Message passing and scheduling in Osprey. 2013.

[26] M. Schwarzkopf, M. P. Grosvenor, and S. Hand. New wine in old skins: the case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 9. ACM, 2013.

[27] M. I. Seltzer and N. Murphy. Hierarchical file systems are dead. In *HotOS*, 2009.

[28] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[29] C. A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. MIT Press, May 2013.

[30] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*, volume 17, pages 49–70. Acm, 1983.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.