DaLi: Database as a Library

Gowtham Kaki[1], KC Sivaramakrishnan[2], Thomas Gazagnaire[3], Anil Madhavapeddy[3], and Suresh Jagannathan[1]

1 **Purdue University**
2 **University of Cambridge**
3 **Docker, Inc.**

Abstract

The landscape of data-intensive applications today span the gamut from large-scale Web applications expected to provide persistent, fault-tolerant, high-availability, and low-latency geo-distributed services to loosely connected IoT networks comprised of millions of heterogeneous devices streaming and processing realtime data feeds. In both cases, application logic is usually expressed using high-level, often domain-specific, language abstractions, while data management issues are typically relegated to an opaque monolithic data querying and storage service. While this architecture encourages separation of concerns, it provides little opportunity for synergies between the application and database/storage boundary. In particular, applying well-understood programming language principles and verification techniques to ensure data management services enforce application-level invariants becomes difficult, jeopardizing safety and maintainability.

To overcome these drawbacks, we propose a radically different view of how applications and databases should interact with one another. Our approach encapsulates data management functionality within transparent libraries written in the same language as the application they support (OCaml in our case). An immediate benefit of our approach is that properties relevant to the application can be now directly interpreted, enforced, and verified by the data management layer. Similarly, database functionality related to consistency, integrity, scalability and fault-tolerance can be couched in terms of the data types manipulated by the application.

Our ideas can be thought of as a natural extension of unikernels, applied to data (as opposed to computation). We sketch the design of a library-centric data management stack called DaLi, and describe how we unify data representation issues across layers, and exploit language-level data type information to realize scalability and persistence.

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.23

1 Introduction

Modern day Web applications increasingly rely on distributed, fault-tolerant databases to provide important data management services that ensure an always-on experience. Complex application requirements have given rise to a large number of purpose-built distributed data management solutions. The software stack that props up these critical services is a messy and chaotic one. While developers use high-level language abstractions and data structures to program application logic, they nonetheless implicitly rely on lower-level data management, operating system, and runtime services for many important features such as persistence, multi-threading, network communication, etc. Unfortunately, the semantics of these services and the representation of the objects they manipulate are often defined independently from the high-level language structures that use them. As a result, unlike traditional database systems, distributed data management solutions in these environemnt routinely violate application-level consistency and integrity constraints.

Further complicating matters is the recent proliferation of Internet-of-Things (IoT) devices. Here, developers face the daunting task of programming against a loosely connected network

of millions of unreliable, resource constrained devices, where heterogeneity and failures are the order of the day, and issues of security and correctness are paramount. These devices often have access to personal and sensitive data (such as GPS-based locators and heart rate monitors) or control safety-critical systems (such as smoke alarms and home security). It is increasingly apparent that *ad hoc* data management solutions running over unstable system software will contribute to catastrophic failures in these environments.

In response to the concerns, recent efforts have looked at shrinking the number of layers (and their complexity) in the software stack through the use of principles expounded by library OS designs [7, 11]. These techniques lift many of the services traditionally performed by operating systems into the application layer, and link the application directly to the hardware resources rather than depending on an ambient operating system kernel to mediate such traffic. Protocols and services that would ordinarily be encapsulated with operating system modules are instead handled by high-level application libraries that selectively use drivers required to drive the underlying architecture (either bare-metal, more commonly a hypervisor to provide virtual memory and coarse process multiplexing). These so-called *unikernels* [13] thus provide specialized computation environments tailored for both the target hardware and for the applications that execute on them.

While unikernels have been promoted as an effective means of reducing *control* complexity, this paper examines their design rationale as a vehicle for reducing *data* complexity. Broadly construed, data complexity relates to various aspects of data management and manipulation that form the bulk of modern-data enterprise application activity that typically consists of three superficially distinct layers:

- An application layer that enforces a *domain model* to capture application-specific invariants and relationships;
- A *data model* layer that represents the translation of these invariants into a lower-level logical representation (e.g., SQL) amenable for efficient manipulation and querying;
- A *storage model* that handles issues of persistence, consistency, fault tolerance, and data distribution.

While programmers prefer to reason about application semantics in the application layer (i.e., in terms of the domain model), the architecture of the stack requires them to reason about a combination of properties across all the layers. The resultant tension is pithily captured in the following quote by David Heinemeier Hansson, the creator of Ruby-on-Rails [6, 2]:

> I don't want my database to be clever! ... I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you'll have to pry that logic from my dead, cold object-oriented hands ... I want a single layer of cleverness: My domain model.

Unfortunately, keeping the domain model hidden from the database while still realizing efficiency and correctness is a non-trivial challenge: as data moves from one layer to the next, its representation can change dramatically (e.g., a tree structure in the application may be transformed into a relational table or key-value persistent object in the data store). Change in representation also entails change in specification and invariant definition; relating high-level invariants on application data structures to low-level invariants on store-defined objects thus requires new reasoning principles. System-specific properties of the storage model—such as consistency and replication—also affect application integrity, further complicating this reasoning. Indeed, existing attempts to isolate domain specific knowledge from the database has had mixed success at best, leading to surprising violations of application integrity [2].

There are two issues we believe conspire against building efficient yet verifiably correct data-centric applications:

- A representation mismatch between the domain model of the application and the data model of underlying data store, and
- A semantic mismatch between high-level integrity specifications of the domain model and the low-level system-specific properties of the data storage model.

Rather than letting low-level concerns dictate data representation choices at the storage layer (e.g., SQL relations) or the application layer (e.g., CRDTs [16]), our solution (called DaLi) *unifies* data representations across all layers of the stack to the one adopted by the domain model. To support high-level data representation at the database storage layer, DaLi adopts a content-addressable distributed memory abstraction that provides persistent storage within a versioned Git-like architecture. This design enables system-specific aspects of database functionality such as persistence, replication, data consistency and fault-tolerance to be expressed in terms of application-meaningful semantics. Our prototype implementation builds *all* of this directly in OCaml, with data management and storage functionality described as OCaml module signatures and implemented as OCaml modules, and with applications parameterized across just the signatures they need using OCaml functors.

DaLi is thus a distributed data management system implemented as a collection of *replicas* each of which is instantiated to precisely provide functionality necessary to guarantee salient application invariants are preserved. Each of these replicas execute on top of a hypervisor and thus subsume the functionality of a typical unikernel. Our vision thus conceives a database system to be a library of specialized components rather than as a general-purpose service; the components required for a specific application depends entirely on the application's needs. Because each of these components are expressed using the same OCaml language features as the application itself, whole system reasoning becomes feasible and effective, thus bridging the semantic mismatch between the domain model and the data and storage models.

While the conceptual elegance of a unified language framework for programming data-centric distributed systems has obvious appeal, we believe the practical significance of such a system—especially in the context of emerging application areas like IoT—will be particularly important. The inherent diversity with respect to scale, heterogeneity, and functionality in such systems demands solutions that both simplify system complexity and facilitate end-to-end correctness arguments. DaLi's unikernel-inspired framework provides these ingredients by radically eliminating and simplifying abstraction boundaries in the data path.

2 Motivation

In a conventional application stack, the domain model is usually captured as a set of inter-related datatype definitions in a high-level language, while business logic is implemented around in-memory objects and their data structures. Consider the example in Fig. 1. The figure shows the domain model of an IoT device tracking application written in OCaml that, for illustrative purposes, only includes the information about a device's security setting, and the task groups in which the device participates (e.g., a networked speaker might participate in a music streaming task with other speakers in the home). The domain model nonetheless captures important representation and semantic invariants of the application. For example:

- A device or a group is represented as a record. Security information is a variant record, whereas the role a device plays in a group is defined in terms of an enumerated type (`Coordinator` or `Participant`).

```
type role = Coordinator | Participant
type security =
  | Public of { ip: string; }
  | Protected of { auth: string; network: net ref }
type device =
  { name     : string;   joined : Time.t;
    security : security; groups : (group ref * role) list }
and group =
  { name : string; desc : string; devices : device ref list}
```

Figure 1 The domain model of a device tracking application in OCaml

- A device is part of zero or more task groups[1]. Conversely, a group contains zero or more devices. Both these invariants follow from the inductive definition of lists. The type system also ensures that a protected device is part of precisely one network.
- Type-safety guarantees provided by ML ensure that if a device, a network, or a task group refers to one another, then the referenced entity is guaranteed to exist.

Unfortunately, the knowledge captured by the domain model does not translate straightforwardly to the data model of a typical data store. For instance, if our application were to use a relational database to provide efficient querying capability, persistence, and fault tolerance of device state, the various datatype definitions given in Fig. 1 would have to be translated to relational schema definitions. Since there may not exist a store-level analogue of a high-level type, the application has to ensure that every high-level type maps to an appropriate store type during the translation (e.g., `role` and `string` types might be mapped to MySQL's `varchar` type). Furthermore, techniques such as normalization may need to be employed to appropriately map compositionally defined ML datatypes to flat relational schemas.

More significantly, invariants that are implicit in datatype definitions need to be made explicit in the relational schema. In our example, this would entail establishing foreign key relationships between device and group relations. Integrity constraints such as non-zero group membership that can be enforced naturally in the source, now need to be defined through other (usually tool-specific) means. The task of adding efficient data management services to an application strips the programmer of the comforts of high-level reasoning, requiring reasoning in terms of a data model that is far removed from application semantics.

The above example demonstratates a so-called *object-relational impedance mismatch* [10]. While several object-relation mapping (ORM) frameworks exist [15, 3], their solutions are *ad hoc* at best, and provide no formal, rigorous correctness guarantees. For example, using a NoSQL store such as MongoDB or Cassandra, instead of a relational one introduces further complications. High scalability and availability requirements compel a NoSQL store to eschew strong data consistency guarantees that are otherwise provided by a relational store, making it difficult to understand how application integrity invariants can be enforced. For instance, in the absence of any provision to specify foreign key relationships, how does a NoSQL store ensure referential integrity between devices and groups? Are transactional semantics required? If so, in what form? Answering such questions is understandably hard for an application programmer not familiar with the nuances of data store functionality. The DaLi stack is designed to address challenges like the ones described above.

[1] Devices, such as Amazon's Echo, can multi-task.

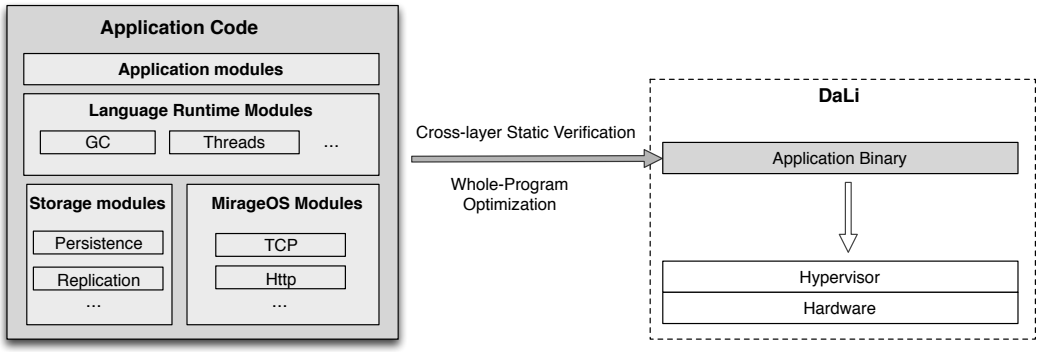


Figure 2 The DaLi application stack is shown on the right. Components and workflow that lead to the application binary are highlighted in gray.

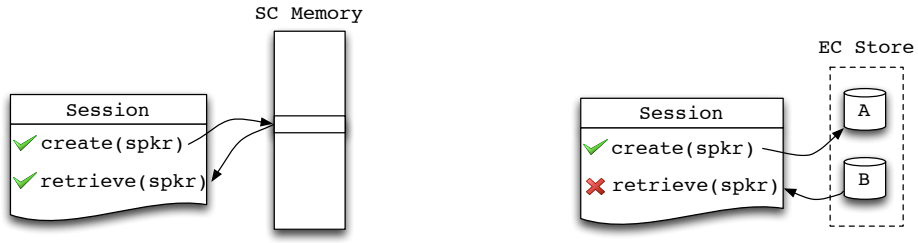
3 DaLi

Figure 2 summarizes the architecture of the DaLi stack, and the typical workflow of DaLi applications. DaLi applications code necessarily includes modules that implement application logic, and optionally includes various supporting modules, such as a language-level thread library, libraries that implement persistence and replication, libraries that interface with the hypervisor (e.g., MirageOS [13]) etc. The presence of the entire application code base enables cross-layer static verification and whole-program optimization, resulting in an optimized application binary that runs directly on the hypervisor and the hardware. While each component of the DaLi stack and workflow merits discussion, space considerations lead us to focus our attention on the verification (Section 3.1) and data representation and storage (Section 3.2) components. Both these components are critical to help DaLi achieve its twin goals: while the verification library component bridges the semantic gap between high-level application requirements and lower-level data management guarantees, the data store library component is designed to overcome the mismatch between application and persistence layers with respect to data structure representation and management. The DaLi compilation strategy also raises interesting challenges briefly addressed in Section 3.3.

3.1 Cross-layer Static Verification

As discussed in Section 2, the task of translating high-level application invariants to appropriate store-level properties is non-trivial, especially in the presence of replication and the desire for high availability. It is therefore imperative for an alternative software stack that aims to simplify database programming to provide automatic reasoning tools that help applications map their high-level requirements to appropriate lower-level datastore guarantees. We have built a static verification engine called Q6 to serve this purpose.

Q6 is primarily designed to verify whether an application's invariants can be preserved under a given choice of a data store-level configuration. In its current form, Q6 focuses on the store configuration pertaining to operation consistency [18] and transaction isolation levels [4]. To understand the functioning of Q6, let us consider the record keeping application from Fig. 1. Suppose the application supports `add` and `retrieve` operations on device records, the former creating a new device record and inserting it into a `devices` collection, and the later retrieving a device record (by name) from the collection. Consider a sample client session that adds a new device record for spkr, and subsequently retrieves it. Given a



(a) Under an SC memory abstraction, `retrieve` followed by `add` is guaranteed to succeed.

(b) Under an EC replicated store, `retrieve` may fail if it contacts a different replica than the one used by `add`.

Figure 3 A client session executing against an SC memory abstraction and an EC replicated store.

sequentially consistent (SC) memory abstraction, the retrieve operation for spkr's record is always guaranteed to succeed, and the session never throws an error (Fig. 3a). This simple invariant, enforced implicitly by language mechanisms, is however not guaranteed to hold if we allow persistence and weakly consistent replication, which allow operations applied to just a single replica of the store to immediately succeed, asynchronously merging their updates at other replicas in the background. In the current example, this admits the possibility (Fig. 3b) that an `add` operation is applied to one replica (A) of the store, while a subsequent `retrieve` operation contacts a different replica (B) that hasn't yet merged A's updates, causing `retrieve` to fail. To avoid such anomalies, we need to wrap `retrieve` within an implementation of a so-called *Read-My-Writes* (RMW) session guarantee [17], to ensure the operation witnesses all effects of all previous operations from the same session. Unfortunately, RMW is only one of the many possible consistency guarantees [18] that can extend the capabilities of weakly consistent replication. The canonical way of specifying such guarantees is through a detailed axiomatic semantics defined in terms of relations over low-level system traces [5], a characterization far-removed from application semantics.

Q6 automates the aforementioned reasoning process entirely. Using the data store's cost model that associates cost to alternative configurations of the store (e.g., an *eventually consistent* (EC) configuration has the least cost, RMW configuration has a slightly higher cost, and an SC configuration has the highest cost), and with help of provable counterexamples that demonstrate the violation of application integrity, Q6 infers the least cost dynamic store configuration strategy that is guaranteed to preserve salient application invariants. In the current example, Q6 strengthens the consistency level of a `retrieve` operation, as described above, but ensures weak consistency for other operations to retain performance.

At its core, Q6 is a symbolic execution engine that generates first-order constraints which encode application integrity and data consistency properties, relying on an SMT solver to solve constraints and generate counterexamples. Fortunately, there exist a range of well-known consistency and transaction isolation guarantees with clearly-defined (albeit low-level, as described above) first-order semantics. These guarantees can be implemented as modular extensions to a weakly consistent replicated store, thus letting us extend store capabilities as required. An application may not need the whole range of consistency/isolation guarantees, in which case Q6 can be used to determine the required subset of consistency guarantees that need to be included in the application binary.

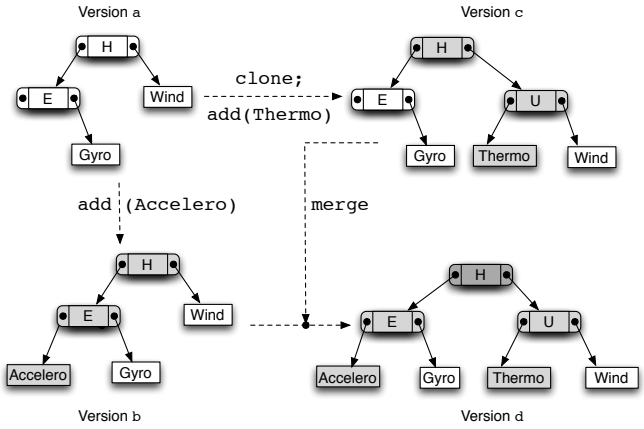


Figure 4 Concurrent modifications to a replicated data structure results in branching in Irmin. Grey background highlights modifications on each branch. A three-way merge involving diverging versions (b and c) and their common ancestor (a) is performed to collapse the branches. The merged version (d) includes modifications from both the branches, with all conflicts resolved (dark gray).

3.2 Storage Library

The cornerstone of the DaLi stack is a library called Irmin that implements a persistent multi-versioned store with a content-addressable memory abstraction[2] [8]. Multi-versioning and content-addressability lets Irmin support persistent OCaml data structures, thereby eliminating the representational mismatch between the application layer and the data store layer. Reasoning in terms of functional data structures is now uniformly applicable. Irmin's intrinsic support for object-level versioning facilitates application integrity enforcement. For instance, in our running example, suppose patio thermometer's device record refers to the record of the temperature group, and suppose the application updates the group name to indoor temperature, while also removing patio thermometer from the group's `devices` and the group from patio thermometer's `groups`. Despite these changes, a computation that references an older record of patio thermometer would continue to see it as part of the temperature group, thus witnessing a consistent state.

Persistence itself is however not the only service expected of the storage layer; sophisticated applications often require replication, fault-tolerance, and concurrency control. To admit replication, Irmin's storage model is based on the principles of distributed version control systems. This model fits naturally with the challenges of distributed data management, where the replicas are expected to serve clients requests even when there are network partitions. At its core, Irmin provides an efficient way to *clone* data structures, and enables rich and expressive ways to be able to efficiently *merge* back the cloned structures. To see how clone and merge are useful in maintaining replicated data structures, let us consider a B-tree data structure of device records that the tracking application (Fig. 1) might use to maintain the collection of all devices. Suppose an `add` operation intends to insert a device record for `thermometer` into the tree while another insertion operation (for `accelerometer`) is in process. Instead of waiting for `accelerometer`'s `add` operation to conclude, `thermometer`'s `add` action can execute at a replica of the tree obtained by cloning an earlier consistent

[2] Simply put, content-addressibility means that the address of a data block is determined by its content. If the content changes, then so does the address. Old content continues to be available the old address.

version. Following the analogy of distributed version control, `accelerometer`'s `add` is said to execute on the master branch, whereas `thermometer`'s `add` executes on the cloned branch. After both insertions complete, the branch can be merged back to the master resulting in a version that contains both insertions (Fig. 4). While generic merge strategies are often useful, Irmin lets applications define merge semantics for each data structure via a three-way merge function that operates on the two diverging versions and their common ancestor. In addition, the Irmin library includes implementations of well-known data structures equipped with efficient 3-way merges such as counters, logs, queues, ropes, etc [8]. Notably, the specification of these merge operations can be given in terms of OCaml data structures and semantics since there is no longer a representation mismatch between the application and storage layers.

3.3 Uncollapsing the stack

The unikernel compilation strategy offers benefits of cross-layer verification and whole-program optimizations. But unikernels, as traditionally envisioned, also produce monolithic executables as compilation artifacts. Adopting this philosophy to DaLi means combining the application, data management, and data storage layers within the same executable. While small optimized executables are beneficial in terms of low-latency service scalability and isolation [12], monolithic executables stand in stark contrast to state-of-the-art database systems where the application and data storage layers are deployed as independent services.

While we have argued about the deficiencies of such a design, its primary advantage - enabling each layer to be *independently* horizontally scaled on demand, is an important one. For example, storage layer nodes are typically provisioned on hardware with fast, reliable, redundant array of disks, while application layer nodes are provisioned on hardware with fast CPUs and high bandwidth network connections, with cloud providers offering virtual machines specialized for different data management service requirements [1]. With monolithic executables, provisioning becomes more challenging as the machines provisioned must be optimized along multiple axes simultaneously. Moreover, government regulations may stipulate legal restrictions on handling, storage and transfer of data across geographic boundaries [14]; compliance of these regulations becomes harder in the presence of homogeneous monolithic nodes, all of which store and manage data.

To address these concerns, we envision a compilation strategy that uses compile-time meta programming [19] to retain the advantages of unikernels but still reap the benefits of multi-tiered deployment. Our idea is to *co-compile* different layers in the DaLi stack, where compile-time knowledge is fed as configuration time parameters for each layer [9]. Each layer is described in terms of a uniform substrate that provides fine-grained representation and access capabilities, the actual manifestation of data distribution and storage is determined by backend implementations that can be linked, but not directly modified, by the application. Concretely, a DaLi instance can specify relationships and integrity constraints among Irmin objects, but the mapping and serialization of these objects onto a performant backend data store remains the purview of the Irmin implementation, not the application.

4 Conclusions

DaLi is a new approach to storage management that meets the challenge of modern application's needs for geo-distributed, loosely-coupled, heterogenous storage. The modular, library-based approach provides a principled foundation for tackling the emerging large-scale problem of security, confidentiality, long-term archiving, and delay-tolerant search presented by the billions of mobile and embedded systems being deployed across the Internet.

References

1 Amazon Ec2 Instances, 2017. Accessed: 2017-01-04 10:12:00. URL: https://aws.amazon.com/ec2/instance-types/.

2 Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2723372.2737784, doi:10.1145/2723372.2737784.

3 Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

4 Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM. doi:10.1145/223784.223785.

5 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535848.

6 Choose a single layer of cleverness, 2005. Accessed: 2017-01-03 12:21:00. URL: http://david.heinemeierhansson.com/arc/2005_09.html.

7 D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM. URL: http://doi.acm.org/10.1145/224056.224076, doi:10.1145/224056.224076.

8 Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy. Mergeable persistent data structures. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, 2015.

9 Functoria: A DSL to invoke otherworldly functors, 2017. Accessed: 2017-01-04 10:12:00. URL: https://github.com/mirage/functoria.

10 Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*, DBKDA '09, pages 36–43, Washington, DC, USA, 2009. IEEE Computer Society. URL: http://dx.doi.org/10.1109/DBKDA.2009.11, doi:10.1109/DBKDA.2009.11.

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.Sel. A. Commun.*, 14(7):1280–1297, September 2006. URL: http://dx.doi.org/10.1109/49.536480, doi:10.1109/49.536480.

12 Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 559–573, Berkeley, CA, USA, 2015. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2789770.2789809.

13 Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30:30–30:44, December 2013. URL: http://doi.acm.org/10.1145/2557963.2566628, doi:10.1145/2557963.2566628.

14 European union model clauses, 2017. Accessed: 2017-01-03 13:00:00. URL: https://www.microsoft.com/en-us/TrustCenter/Compliance/EU-Model-Clauses.

15 Active record basics, 2017. Accessed: 2017-01-03 13:00:00. URL: http://guides.rubyonrails.org/active_record_basics.html.

16 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-24550-3_29.

17 Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=645792.668302.

18 Paolo Viotti and Marko Vukolic. Consistency in Non-Transactional Distributed Storage Systems. *CoRR*, abs/1512.00168, 2015. URL: http://arxiv.org/abs/1512.00168.

19 Jeremy Yallop and Leo White. Modular Macros. In *Proceedings of the 2015 OCaml Users and Developers Workshop*, OCaml'15, 2015. URL: https://www.cl.cam.ac.uk/~jdy22/papers/modular-macros.pdf.