ABSTRACT

Title of dissertation: DYNAMIC UPGRADES FOR HIGH AVAILABILITY SYSTEMS

Karla Saur, Doctor of Philosophy, 2015

Dissertation directed by: Professor Michael Hicks and
Professor Jeffrey S. Foster
Department of Computer Science

In this thesis I show that it is possible to build general-purpose frameworks for efficient, on-line data transformation in support of flexible system services, especially dynamic software updates (DSU). This approach generalizes some of the ideas from prior work on DSU, making those ideas applicable to more situations. In particular, I generalize DSU's notion of in-memory state transformation—normally used to upgrade run-time state to be consistent with the new software—so that it can be applied to data not necessarily stored in memory, and for services other than DSU.

To support this thesis, I present three artifacts. First, I present C-strider, a generic, type-aware C heap traversal framework. C-strider constitutes a flexible, easy-to-use framework with which developers can program reusable services that have a heap traversal at their core, e.g., serialization, profiling, invariant checking, and state transformation (in support of DSU). C-strider supports both parallel and single-threaded heap traversals, and I demonstrate that C-strider requires little programmer effort, and the resulting services are efficient and effective. Second,

I present KVolve, a data transformation service for NoSQL databases. KVolve is notable in that transformations are carried out on-line and on-demand, as data is accessed, rather than off-line and all at once, which would reduce service availability. Experiments with on-line upgrades of services using KVolve show little overhead during normal operation, and only brief pauses at update-time. Finally, I present Morpheus, a dynamically updatable software-defined network (SDN) controller. Morpheus' architecture is fundamentally distributed, with each service running as a separate process that accesses a shared KVolve instance. Morpheus can update multiple controller applications without loss of availability or degradation of performance.

DYNAMIC UPGRADES FOR HIGH AVAILABILITY SYSTEMS

by

Karla Saur

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor Michael Hicks, Chair/Advisor
Professor Jeffrey S. Foster, Co-chair/Advisor
Professor Rajeev Barua, Dean's Representative
Professor Alan Sussman
Professor Tudor Dumitraş

To Karyn and Shelley,

and all of the other brilliant, resilient women

making their way in the crazy world of computer science.

And to Julian,

my cohort in escapades.

Acknowledgments

I cannot fully express how crucial Mike Hicks has been to my growth and development as a computer scientist and researcher. Mike was willing to work with me and my diverse interests, and most importantly was able to fully provide the *patient* guidance that was so critical to my success as a student. Few students are so lucky! Jeff Foster greatly helped me improve my writing, critical thinking, and articulation skills, and I am especially grateful for his help during those crucial first few years in learning the research process. I want to thank everyone in the PLUM (Programming Languages at the University of Maryland) Lab for all they've done for me throughout my Ph.D. experience.

I was also fortunate to be able to collaborate with many people during my Ph.D. program including Tudor Dumitraş, Chris Hayden, Ted Smith, Nate Foster, Arjun Guha, Laurent Vanbever, and Joseph Collard. These collaborators contributed greatly to this dissertation and my Ph.D. experience was fortified through these collaborations.

A special thanks to my friends that were so supportive throughout the process, especially Amber, Anup, Austin, Ewa, Ilya, Karyn, Marco, Shelley, Sofia, and Zhanna. Thanks to my family for their unfailing support. And most of all, thank you to Julian for encouraging me to actualize my aspirations.

This work was performed with support from a Google Faculty Research Award, NSF grant numbers CCF-0910530 and CCF-1116740, and the partnership between UMIACS and the Laboratory for Telecommunications Sciences.

Table of Contents

List of Tables viii

List of Figures ix

List of Abbreviations xi

1 Dynamic Upgrades for High Availability Systems 1
- 1.1 Software Upgrades . . . 2
  - 1.1.1 Traditional (shutdown-and-restart) upgrades . . . 2
  - 1.1.2 Dynamic software updating . . . 3
  - 1.1.3 DSU components . . . 4
- 1.2 Abstracting DSU Principles to General-Purpose Frameworks . . . 6
- 1.3 Contribution Overview . . . 7
  - 1.3.1 C-strider: a generic type-aware C heap traversal framework . . . 7
  - 1.3.2 KVolve: a data transformation service for NoSQL databases . . . 9
  - 1.3.3 Morpheus: a dynamically updatable SDN controller . . . 10

2 C-Strider: Type-Aware Heap Traversal for C 13
- 2.1 Developing Services with C-strider . . . 18
  - 2.1.1 C-strider API . . . 18
  - 2.1.2 Implementing serialization . . . 19
  - 2.1.3 Implementing deserialization . . . 22
  - 2.1.4 Using serialization for checkpointing . . . 24
  - 2.1.5 Full C-strider API . . . 27
- 2.2 Type-aware Traversal . . . 29
  - 2.2.1 Main components of the traversal . . . 30
  - 2.2.2 Generated code . . . 34
- 2.3 Customizing the Traversal . . . 37
  - 2.3.1 Type annotations . . . 37
  - 2.3.2 Customization in perfaction functions . . . 41
- 2.4 Limitations . . . 43
- 2.5 Applications and Experiments . . . 45

2.5.1 Programmer effort . . . 46
2.5.2 Heap serialization . . . 49
2.5.3 State transformation . . . 50
2.5.4 Heap profiling . . . 55
2.5.5 Heap assertion checking . . . 57
2.6 Related Work . . . 58
2.7 Conclusion . . . 61

3 KVolve: Evolving NoSQL Databases Without Downtime 63
3.1 Overview . . . 68
3.1.1 Background on Redis and NoSQL databases . . . 68
3.1.2 Example data format change . . . 69
3.1.3 Database upgrades using KVolve . . . 71
3.1.4 Describing data updates . . . 74
3.2 KVolve Implementation . . . 75
3.2.1 Design goals . . . 76
3.2.2 KVolve implementation overview . . . 77
3.2.3 String types . . . 79
3.2.4 Sets, hashes, lists, and sorted sets . . . 83
3.2.5 Installing an update . . . 84
3.3 Experimental Results . . . 85
3.3.1 Steady state overhead . . . 86
3.3.2 Redisfs . . . 91
3.3.3 Amico . . . 94
3.4 A Python client-side version of KVolve: Py-KVolve . . . 96
3.4.1 DB upgrades using Py-KVolve . . . 97
3.4.2 Implementing lazy updates . . . 100
3.4.3 Py-KVolve steady state overhead . . . 103
3.5 Related Work . . . 105
3.6 Future Work . . . 108
3.6.1 Distributed data updates . . . 108
3.6.2 Automatic generation of transformation functions in NoSQL updates . . . 109
3.7 Conclusion . . . 110

4 Morpheus: Safe and Flexible Dynamic Updates for Software-Defined Networks 112
4.1 Overview . . . 118
4.1.1 Simple restart . . . 118
4.1.2 Record and replay . . . 120
4.1.3 Solution: update by state transfer . . . 123
4.2 Morpheus Controller . . . 125
4.2.1 Architecture . . . 126
4.2.2 Components . . . 127
4.3 Dynamic updates with Morpheus . . . 130

4.3.1 Update protocol . . . 130
4.3.2 Update example: Firewall . . . 132
4.3.3 Coordination: Routing and Topology . . . 135
4.4 Experiments and Evaluation . . . 137
4.4.1 Firewall . . . 138
4.4.2 Routing and Topology . . . 139
4.4.3 Load Balancer . . . 141
4.4.4 Programmer effort . . . 143
4.5 Related Work . . . 144
4.6 Future Work . . . 146
4.6.1 Rollback in SDN controller updates . . . 147
4.6.2 Correctness in SDN controller updates . . . 147
4.7 Conclusion . . . 149

5 Conclusion 151

A Py-KVolve Details 153
A.1 Specifying Updates . . . 153
A.1.1 DSL syntax and semantics . . . 153
A.1.2 Example updates . . . 157
A.1.2.1 Manipulating JSON fields . . . 157
A.1.2.2 Renaming the database keys . . . 158
A.1.3 Implementing data transformations . . . 159
A.2 Details of Implementing Lazy Updates in Py-KVolve . . . 159
A.2.1 Installing an update . . . 160
A.2.2 Additional information about GET operations . . . 161
A.2.3 Additional information about SET operations . . . 162
A.2.4 Other commands . . . 162
A.3 Py-KVolve Update Overhead . . . 163
A.3.1 Update pause time using Py-KVolve . . . 163

Bibliography 167

List of Tables

2.1 Programmer effort (measured in LOC) 47
2.2 C-strider profiling heap data 56

3.1 Redis-bench with single instructions for Redis vs KVolve (times in seconds, median of 11 trials) 87
3.2 Redis-bench with 10 pipelined instructions for Redis vs KVolve (times in seconds, median of 11 trials) 88
3.3 Max resident set size (RSS) 89
3.4 Offline update pause times 96
3.5 Comparing running time of Py-KVolve and Redis (times in seconds) . 104

4.1 Update quiescence times for Topology and Routing (median of 11 trials) 140

A.1 The DSL directives 154
A.2 The DSL convenience tokens 156

List of Figures

2.1 C-strider architecture. . . . 14
2.2 Basic C-strider service API. . . . 18
2.3 An example service: serialization and deserialization. . . . 20
2.4 Example program employing checkpointing. . . . 25
2.5 C-strider API. . . . 27
2.6 Traversing using type information. . . . 32
2.7 Generated traversal code for struct dlist 34
2.8 Length annotation and generated traversal code. . . . 38
2.9 Generic annotation and generated traversal code. . . . 39
2.10 Customizing traversal in perfaction functions. . . . 41
2.11 perfaction code for Memcached. . . . 48
2.12 Serialization times (color plot). . . . 51
2.13 DSU state transformation times (color plot). . . . 54

3.1 Evolving a purchase order object 70
3.2 KVolve architecture. . . . 71
3.3 Example update function for JSON 75
3.4 Control flow for Redis and KVolve 77
3.5 Storing different data types 80
3.6 Lazy vs. eager updates for RedisFS 91
3.7 Lazy vs. eager updates for Amico 94
3.8 Py-KVolve architecture. . . . 97
3.9 Specified update to purchase order objects 99

4.1 Example application: stateful firewall. . . . 119
4.2 Example application: load balancer. . . . 121
4.3 Morpheus architecture. . . . 126
4.4 Firewall update 137
4.5 Routing and topology discovery update 139
4.6 Load balancer results 142

A.1 DSL commands 155
A.2 Example updates 158

A.3 Lazy vs. eager updates for gets and sets over the full range of 200,000 keys . . . 164
A.4 Lazy vs. eager updates for gets and sets over a 20,000 key subset in a 200,000 key database . . . 165

List of Abbreviations

DSL	Domain-Specific Language
DSU	Dynamic Software Updating
HS	HotSwap
GC	Garbage Collection
ISSU	In-Service Software Upgrades
JSON	JavaScript Object Notation
KLOC	Thousands of Lines Of Code
LOC	Lines Of Code
NIB	Network Information Base
NoSQL	Non-Structured Query Language
RHEL	Red Hat Enterprise Linux
SDN	Software-defined network
SIQR	Semi-Interquartile Range
SQL	Structured Query Language
UPDC	Update Coordinator
XML	Extensible Markup Language

Chapter 1: Dynamic Upgrades for High Availability Systems

Continuous availability of software systems is increasingly important to users globally. The effect of an outage can range from minor frustrations to costly to outright dangerous. Eleven million Americans in seven states experienced a 911 outage for six hours when a software glitch cascaded, causing 6,600 calls to go unanswered [119]. A 2014 Facebook outage lasting 30 minutes was estimated to have cost the company $500,000 [69]. One analyst went so far as to estimate that the economy (retailers, news outlets, and other companies using advertising) could be losing as much as $25 million per minute of Twitter outage [113]. Unplanned service outages may have many causes, such as power outages, hardware failures, or software glitches. However, despite the disruption caused by outages, one study by Hewlett-Packard Laboratories found that in 426 servers running high-availability applications, 75% of the 5,921 outages were due to *planned* maintenance rather than unexpected downtime [64]. They also found that planned outages lasted around twice as long as unplanned outages, further highlighting the need to improve the system maintenance process.

1.1 Software Upgrades

These planned outages often involve various forms of software upgrades. Even systems that must be constantly available sometimes need to be upgraded to add new features or to fix existing security problems or other bugs. Many sites that provide high availability services such as Amazon Web Services and Salesforce post a planned maintenance schedule, notifying customers of scheduled unavailability [88, 100]. There are several options for updating software, and this section provides an overview.

1.1.1 Traditional (shutdown-and-restart) upgrades

The most straightforward way to update a program is to simply shut down the old version and restart the new version. However, this method loses all of a program's state and makes the program unavailable during the switch-over. It is therefore not practical for systems requiring high availability.

As such, many administrators and end-users find software updates annoying and in some cases attempt to postpone or evade critical updates [5,27,74]. However, this practice can be very dangerous, particularly in the case of updates that harden program security. For example, even after the detection of the the Heartbleed vulnerability (a major flaw in OpenSSL's software), four months later only half of the servers with the vulnerability were patched [60]. In 2012, Skype announced the results of a survey of why users do not update their software when prompted, and found that nearly half of all users fail to update their software regularly, often citing

that they don't know why they should bother to update, and that "upgrades take too long" [105].

1.1.2 Dynamic software updating

The goal of dynamic software updating (DSU) is to update a program while it is executing without causing downtime. Because this practice minimizes the negative impacts of software updates, it mitigates the problem of users ignoring upgrade prompts or administrators postponing updates, as the users of a system should be largely unaffected or even unaware of the dynamic upgrade.

Researchers have developed many DSU systems for programs written in several languages such as C, C++, and Java [16, 22, 37, 46, 48, 65, 71, 78, 111]. Some DSU systems deal with updates in the form of patches to existing software [13, 75]. In contrast, many DSU systems such as Kitsune [46] are for whole-program updates, meaning that the upgrade is from one complete version of a program to the next complete (updated) version.

Overall, these DSU systems have proven that they can greatly minimize downtime when applying updates to the systems that they support. Kitsune has been used to dynamically update various C server programs over several years worth of these programs' release history.

1.1.3 DSU components

The process of performing a dynamic software update consists of two parts: upgrading a program's code, and transforming the program's corresponding state:

Updating the code: Updating a program's code is the first thing that many people think of when they think of software updates. This involves changing the code that is being executed, either by moving from one version of the code to the next or by applying a specific patch. In dynamic software updating, the DSU system performs this action in a way that does not incur downtime or prevent connected users from utilizing the software while it is being updated.

One method of upgrading the code while minimizing downtime is rolling upgrades, where clients that are connected to a running program are slowly switched over to the new version of the program. This method of rolling upgrades usually involves multiple instances running at both the old and the new versions [40, 101]. However, this method alone does not work for many programs that contain state that must be retained, discussed next. A more nuanced approach involves redirecting the *control flow* from the old version of the program into the new version of the program, and then performing a hot swap of versions, which is done with many DSU systems [46, 78].

Updating the state: If a program has state that must be maintained throughout the update, methods such as rolling updates will not work. Something must be done to preserve the program's state, so that the program may logically continue

to execute where it left off. Examples of program state are structures on the heap and socket connections to clients. In the case that the program state representation does not change in the upgraded version of the software, the state must simply be *maintained* such as by serializing the state to disk or preserving it within the running system in a way that the new program version can access it.

In the case where program state representation changes between upgrades, the state must be transformed in order to be used by the new program. An example of such a change is adding a field to a structure, or changing the type of a variable from an integer to a string. In a case like this, the state must be transformed to match the new program's expectations. Most DSU systems do this by modifying the old in-memory state. Specifically, Kitsune does this by traversing the program's entire heap, transforming the in-memory state to conform with the new program's state expectations. This in-memory state transformation, coupled with control flow migration, enables the new program version to utilize the updated state so that it can seamlessly pick up where the old program left off.

In many DSU systems [46, 65, 71, 78], the processes of "updating the code" and "updating the state" are effectively intertwined. Streamlining these steps in DSU makes sense in many cases from an ease-of-use perspective, and helps ensure an expedient update.

1.2 Abstracting DSU Principles to General-Purpose Frameworks

My thesis is that it is possible to build general-purpose frameworks for efficient, on-line data transformation in support of flexible system services, especially dynamic upgrades. To do this, I build on some of the ideas from prior work on DSU, making those ideas applicable to a broader variety of situations. Specifically, I generalize DSU's notion of in-memory state transformation so that it can be applied to data not necessarily stored in memory, and for purposes that are not necessarily DSU.

First, I show how we can generalize the upgrade process by decoupling the data transformation from a code update. I present a general framework, C-strider, that allows greater flexibility in the application of data transformation, and this allows us to implement services besides DSU, such as serialization and heap profiling. These additional services are implemented in a few lines of code that are program-independent and therefore can easily be reused in other programs.

Next I show how data transformations can be generalized beyond the confines of a process' memory with KVolve, a system for NoSQL data migration. I show that KVolve can be used to transform entries in the NoSQL database Redis in a variety of use cases, as they are accessed by clients, and present a template framework to assist programmers in writing their updates. KVolve demonstrates flexibility in data transformations across disparate kinds of data changes.

Finally, I show how we can extend updating a single server program to coordinating updates for multiple components of a system with Morpheus, an approach to dynamic software-defined network (SDN) controller updates. Morpheus uses an

update coordinator to orchestrate update timing throughout the controller, including updating multiple applications and program state. I provide the user with a DSL (Domain-Specific Language) to assist them in writing their updates to work with the Morpheus framework.

Together, these works show that it is possible to generalize DSU by building general-purpose frameworks for dynamic data transformation and upgrades in highly available systems.

1.3 Contribution Overview

In this dissertation, these three contributions are evidence that we can expand DSU ideas to general frameworks that have broader data transformation purposes. This section provides an overview of each artifact.

1.3.1 C-strider: a generic type-aware C heap traversal framework

Our C-strider framework (Chapter 2) shows how we can enable flexibility in both *what* and *how* heap data is transformed by generalizing the traversal performed by Kitsune. The first step toward bringing existing DSU work towards a general transformation framework was the observation we often want to traverse program state whether or not we are loading new program code. As mentioned in Section 1.1.3, existing DSU work intertwined updating the state and updating the code into a single command. While ideal for explicitly performing a dynamic update in a server program, having data transformation locked into a software update

constrains the ability to independently transform the data. There are many types of program transformation that do not involve modifying the software, and separating out these two actions lead to greater variety of use cases.

Similar to the traversal used with Kitsune, C-strider creates a type-aware, program-specific heap traversal by generating traversal functions for each type of heap item in a program. However, the framework that C-strider provides can tailor the heap traversal to different purposes, assisted by four user-written callbacks. These callbacks are triggered during the heap traversal and allow the traversal-writer to support other services that they would normally write by hand, specific to a program. C-strider also provides an API that allows a programmer to determine which heap elements to transform (e.g. all heap items or specific elements only), and provides functions that can be used to direct the traversal and to manipulate type information. C-strider supports both parallel and single-threaded heap traversals.

We demonstrate how the C-strider framework can be used to implement general services such as heap serializing and heap profiling, and apply these services to three separate open-source programs. For example, we apply serialization to the in-memory databases Redis and Memcached, and find that serializing 30,000 10-byte key-value pairs takes ~190 ms and ~80 ms, respectively. Implementing both the serialization and deserialization traversal services takes only 78 lines of total code to implement the callbacks, and customizing the traversal to serialize only the database structures took 44 lines of code for Memcached and 46 lines of code for Redis. We also show how we can use C-strider's parallel heap traversal to expedite state transformation in Kitsune, enabling faster updates, such as updating Redis in

less than half of the time required for Kitsune. C-strider is released open source on GitHub [92].

1.3.2 KVolve: a data transformation service for NoSQL databases

Prior DSU work focused on updating heap elements stored in a process' memory, allowing the newly updated software to utilize heap items correctly. We can expand this technique to the problem of updating data in a database when the database schema changes. Similar to the way a program expects a struct to conform to a particular type, database client programs that are connected to the database expect the stored data to conform to a particular schema. Therefore to upgrade the database client software, the data in the database must also be upgraded.

Unlike traditional relational databases, NoSQL schemas are often *implicit*, defined only by software expectations. With KVolve (Chapter 3), we show how we can transform data stored in the key-value store Redis (a type of NoSQL database) so that the stored data reflects the expectations of the implicit schema in the updated client software. KVolve operates by requiring the connecting clients to declare *namespaces*, where all key-value pairs belonging to the namespace have the same schema. These namespaces allow KVolve to track and facilitate updates to the key-value pairs. In order to update the database data, an administrator submits an update specification, describing how to update all key-value pair members in a specific namespace. This update specification consists of a simple framework that can specify a transformation using C code. KVolve applies this transformation *lazily*

as the key-value pairs are accessed by clients, avoiding the pause time of migrating the entire key-value store all at once.

Lazy updates with KVolve are a significant improvement over the traditional offline migration. Offline migration, which involves taking the database offline and migrating all of the data at once, may take several minutes or even hours, resulting in lengthy downtime for end-users during the database's unavailability. Lazy migration with KVolve results in almost no downtime. We show that we can use KVolve to dynamically and lazily update the data for several types of schema changes in two open-source programs requiring schema changes as part of a software update. Specifically, we show that we can apply an update involving key renaming to Amico, a social networks program, reducing the pause time for renaming ~800K test keys from ~100 seconds in the offline case to almost zero seconds with KVolve. We perform an update to both the keys and the values of redisfs, a Redis-backed userspace file system, and show the pause time to be close to zero as opposed to 12 seconds for an offline migration. We demonstrate how we can combine our new update techniques with Kitsune to simultaneously update both the program's code and the data stored in the database, by applying both Kitsune and KVolve to redisfs. We also show that the steady-state overhead for KVolve is in the noise for normal use.

1.3.3 Morpheus: a dynamically updatable SDN controller

In contrast with traditional DSU where updated state was stored within the context of the process or a single piece of software, KVolve introduced updates for

data stored in a database. When a program stores some or all of its state in an external database, this means that any upgrade to that data will affect not only the updated program, but also any other programs making use of that data. This setup forces us to generalize the upgrade procedure from a single entity to a multiple-component system.

An example of such a multiple-component system is a software-defined network (SDN). Traditional networks have a control plane for making routing and forwarding decisions, and a data plane physically enacting those decisions. SDNs split out the implementation of the control plane into one machine in software, the controller, and leave the physical switches only with the data plane, implementing the controller's decisions. Because the SDN controller is the critical "brains" of the network, it must be constantly available to ensure that the network runs properly. Therefore controller downtime must be avoided, even during upgrades. Just shutting down and restarting the controller loses any state it has accumulated, and would necessitate wiping and reinstalling rules on the switches. At best, this process disrupts network operations, and at worst it leads to incorrect behavior.

In Chapter 4, we show how we can extend the action of updating a single self-contained program to the broader action of updating the multiple components of an SDN controller. Our prototype SDN controller, Morpheus, demonstrates the necessity of coordinating updates across all applications running in the controller in order to ensure correctness. All upgrades to applications in Morpheus involve making sure that code and data updates work together. To do this, we use an update coordinator to ensure that all applications that use to-be-updated data are notified

of an update and that no applications ever see outdated (and thus incorrect) data. Similar to KVolve, an administrator must submit an update specification describing the update, and we provide a DSL to assist the administrator in using our update framework.

We use Morpheus to perform a dynamic upgrade to applications including a firewall, a load balancer, and a routing application. These updates involve upgrading the state of the application stored in the database and also the applications themselves. We show that dynamic upgrading by transforming the state is fast and does not cause disruption, unlike other methods of updating such as simple restart or record and replay [116]. For example, we show that for a load balancing application, both the simple restart approach and the record and replay approach drop connections after applying an upgrade that adds an additional host, whereas Morpheus does not drop any connection or cause disruption. Additionally, we show that using our update framework requires minimal programmer effort in our test applications and find that our framework should be general enough to apply to updates for several widely available open source controller applications. For example, an update to our firewall application that adds two new fields to each entry requires only 4 lines of DSL code, and an update to our topology and routing application requires only 3 lines of DSL code to initialize a new field to a default value.

Chapter 2: C-Strider: Type-Aware Heap Traversal for C

Researchers have developed many compelling application services that work by traversing the heap pointer graph of a program, including checkpointing [62, 85], profiling [41], dynamic software updating [46], OS kernel integrity monitoring [77], and data-structure assertion checking [6, 14, 52, 81, 117] and repair [29]. When implemented for programs written in a language like Java, these services can piggyback on a tracing garbage collector. But supporting programs in C, which lacks both a garbage collector and the run-time type information, has to date required a heroic effort.

In this chapter, we present C-strider, a general framework for writing services that traverse and/or transform a C program's heap. C-strider generalizes DSU's notion of in-memory state transformation so that it can be applied in a flexible way for purposes that are not necessarily DSU. Figure 2.1 depicts C-strider's architecture. Given an input C program prog.c, C-strider generates a program-specific traversal (prog_stride.c) that walks the heap starting from any location given its type. As heap locations are visited, the traversal invokes one of a small set of callbacks that implement a program-independent *service* (in service.c), e.g., to implement serialization or dynamic updating. The service code in turn invokes the C-strider run-time library,

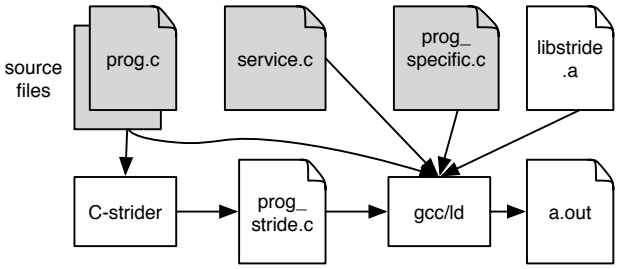

Figure 2.1: C-strider architecture.

libstride.a, for services such as bookkeeping to ensure locations are visited just once. C-strider's API has been designed to be easy to use: Section 2.1 illustrates how to use C-strider to build a portable serialization and deserialization service (e.g., for checkpointing).

A key feature of C-strider is that it is *type aware*—C-strider knows the type `t` of each heap location visited, and passes a representation of `t` to the callback functions, which can vary their behavior depending on the type. C-strider's type information is *exact*, which is critical for many services. For example, a serialization service must know the precise size of objects, and must know precisely whether an object is a pointer and what it points to. As such, conservative garbage collection [20] is not a suitable starting place because its identification of pointers is approximate, and it does not know the types of non-pointer values. We observe that in practice, there is nearly always a statically well-typed pointer/**struct** chain to any location. Accordingly, C-strider analyzes the program and statically generates a traversal that follows well-typed paths to reach most memory, invoking callbacks and passing them type information along the way. The traversal is (mostly) breadth-first and employs a work queue serviced by one or more threads. (Section 2.2 describes C-strider's

traversal.)

However, most C programs lack some type information needed to describe the whole heap. For example, C-strider cannot traverse unions or **void*** pointers without knowing which arm of the union is valid or what the actual pointed-to type is, respectively. The programmer can fill in such missing type information in two ways. One way is programmer annotations, in the style of Deputy [24] and Kitsune, which can provide array length information and precise types for **void*** pointers in generic data structures. The other is by customizing the traversal to consider an object's type specifically in the service code callbacks. For example, a callback can observe when it has reached a **union** embedded in a **struct** and use the values of other fields in that **struct** to determine how to interpret the union value. This program-specific code (in prog_specific.c) is implemented by proxying the callbacks and delegating to the service code to perform the underlying service. Thus, C-strider provides a clean separation between two reusable pieces of code: the *program-specific* traversal (the annotations, the generated traversal, and the code in prog_specific.c, all of which can be used for many services in the same program) and the *service-specific* code (in service.c, and which can be used by many different programs). In addition, the code can be arbitrarily customized for particular program/service pairs, if needed. (Section 2.3 describes the annotations and other forms of program-specific customization.)

We used C-strider to implement four traversal-based services (details in Section 2.5), and used them with three different programs, Memcached (a data caching service), Redis (a key-value store), and Snort (an intrusion detection system).

Serialization and deserialization: This service, suitable for checkpointing, is implemented in just over 60 lines of code (LOC), and is carefully designed to be robust against changes to the program at load time, e.g., symbol relocations due to address-space layout randomization. Checkpointing is relatively fast; we could serialize and deserialize a Memcached heap with 30,000 key-value pairs in under 100ms.

Dynamic Software Updating: We modified Kitsune to implement its state transformation component using C-strider, requiring only 24 LOC. Taking advantage of C-strider's ability to perform multi-threaded traversal, we could dynamically update Redis with a heap of 100,000 key-value pairs in 38% of the time required by Kitsune.

Heap profiling: We developed a profiling service that counts the number and amount of memory consumed by objects of different types. The service required 65 LOC and generates an itemized log, which we used to better understand memory usage in our example programs.

Heap assertion checking: C-strider can be used to check heap assertions in the style of GC assertions [6]. We implemented several simple assertions such as ensuring that linked lists are well formed, timestamps are not in the future, and **enum** fields are valid.

Each application required some service-agnostic customization to provide missing type information, and in some cases some service-specific handling, usually to optimize performance. In all cases, traversal customizations required roughly 20 LOC,

while type annotations depended on program size. For example, for Snort 143 annotations were required across its 215 KLOC, while for Memcached only 6 annotations were required across its 4 KLOC.

C-strider has been carefully designed to implement heap traversal as a *library* employing entirely standard features of C. In particular, C-strider does not modify data structure representations (which could break important assumptions about object size and layout) and does not require custom compilation (which could inhibit compiler optimizations). C-strider also deliberately keeps its type annotation language simple for the common cases, with the ability to drop back to C for reasoning about more complex type invariants (e.g., tagged unions); in our experience, C programs always have unusual, exceptional cases not captured by any sensible annotation system. Finally, C-strider's traversal itself is easy to understand—as we will see in Section 2.2, the entire traversal is simple, written in a few tens of lines of code. Thus, we believe C-strider is accessible to a broad audience and adaptable to many uses.

To our knowledge, C-strider is the first general-purpose, type-aware heap traversal framework for C. We believe C-strider will prove useful for many applications in addition to the ones explored in this chapter, and we plan to release C-strider under an open source license.

```
struct traversal {
  /* Processing primitive values. (int, float, etc.) */
  void (* perfaction_prim )(void *in, typ t, void *out);

  /* Processing struct values. Return 1 to traverse fields in the struct, else return 0.*/
  int (* perfaction_struct )(void *in, typ t, void *out);

  /* Pointer processing; first visit. Return 1 to follow the pointer, else return 0. */
  int (* perfaction_ptr )(void **in, typ t, void **out);

  /* Previously encountered pointer processing. */
  void (*perfaction_ptr_mapped)(void **in, typ t, void **out);
};
```

Figure 2.2: Basic C-strider service API.

2.1 Developing Services with C-strider

This section focuses on the design of C-strider from the developer's perspective. The key challenge in C-strider's design is trading off simplicity—we want the API to be easy to use—and expressivity—we need it to be useful in many situations. We will illustrate our design by showing how to use C-strider to build a serialization and deserialization service.

2.1.1 C-strider API

C-strider traversals aim to visit and transform elements of a program's heap. There are three kinds of elements: *primitives* like integers, enumerated types, etc.; *aggregates* like structs, which are single objects that contain multiple elements; and *pointers* to other elements. C-strider's API asks a service developer to write three

main callback functions, corresponding to the three kinds of heap elements.[1] The names and type signatures these three callbacks are listed in Figure 2.2; we will discuss the last callback below. Pointers to the callbacks reside inside **struct** traversal, which is passed as a parameter to initialize C-strider.

Each callback has the same three arguments. The first argument, in, points to the program data being visited, which has a type represented by t, the second argument. The last argument's use depends on whether the programmer is developing a *transformation* service that modifies memory locations of heap objects, or a *traversal* service that does not. A traversal service like serialization only has one set of traversal roots, and simply ignores out. For a transformation service, like deserialization, out points to replacement memory such that writing out specifies how in's data should be transformed.

2.1.2 Implementing serialization

To explain these three callbacks in more depth, we refer to the code in Figure 2.3a, which implements a serialization service. This service is a traversal service, so out is unused. The functions are passed as callbacks to C-strider in an instance of **struct** traversal from Figure 2.2.

The first callback, serial_prim (assigned as a callback for perfaction_prim), handles data of primitive type (e.g., **int**, **double**). The traversal calls this function with the address of each global variable and each struct/union field that is a primitive

[1] Arrays are another kind of aggregate element; these are handled automatically by the traversal, as discussed in Section 2.2.

```
1  void serial_prim (void *in, typ t, void *out){
2    fwrite(in, get_size(t), 1, ser_fp);
3  }
4  int  serial_struct (void *in, typ t, void *out){

5    return 1;
6  }
7  int  serial_ptr (void **in, typ t, void **out){
8    if (*in == 0)
9      uintptr_t zero = 0;
10     fwrite(&zero, sizeof(uintptr_t), 1, ser_fp);
11   else {
12     char *symbol;
13     if ((symbol = lookup_addr(*in))) {
14       uintptr_t two = 2;
15       fwrite(&two, sizeof(uintptr_t), 1, ser_fp);
16       serial_string (symbol);
17     } else {
18       serial_prim ((void *)in, t);
19       if (t == TYPE_PTR_CHAR)
20         serial_string ((char *)*in);
21       else return 1;
22   } }
23   return 0;
24 }
25
26
27
28
29
30
31
32
33
34 // helper function, not part of API callbacks
35 void  serial_string (char *str){
36   /* assumes str != NULL */
37   int len = strlen(str)+1;
38   fwrite(&len, sizeof(int), 1, ser_fp);
39   fwrite(str, len, 1, ser_fp);
40 }
```

(a) Serialization

```
41 void deserial_prim (void *in, typ t, void *out){
42   fread(out, get_size(t), 1, ser_fp);
43 }
44 int  deserial_struct (void *in, typ t, void *out){

45   return 1;
46 }
47 int  deserial_ptr (void **in, typ t, void **out){
48   void *ptr, *tgt;
49   fread(&ptr, sizeof(void*), 1, ser_fp);
50   if (ptr == (void*)2) { // pointer to symbol

51     char * symbol;
52     deserial_string (&symbol);
53     *out = lookup_key(symbol);
54     free(symbol);
55   }
56   else if (ptr == NULL) // null pointer
57     *out = 0;
58   else if ((tgt = find_mapping(ptr))) // revisit

59     *out = tgt;
60   else // haven't seen it; read the data
61     if (typ == TYPE_PTR_CHAR) { // a string
62       deserial_string ((char **)&tgt);
63       *out = tgt;
64       add_mapping(ptr, tgt);
65     } else {
66       int sz_new = get_size(get_ptrtype(t));
67       tgt = malloc(sz_new);
68       *out = tgt;
69       add_mapping(ptr, tgt);
70       visit (tgt, get_ptrtype(typ), tgt);
71     }
72   return 0;
73 }
74 // helper function, not part of API callbacks
75 void  deserial_string (char **str) {
76   int len;
77   fread(&len, sizeof(int), 1, ser_fp);
78   *str = (char *)malloc(len);
79   fread(*str, len, 1, ser_fp);
80 }
```

(b) Deserialization

Figure 2.3: An example service: serialization and deserialization.

type, indicating the specific type of primitive with typ t. The code for serializing primitives using this function is given at the top of Figure 2.3a. It does the obvious thing: it uses API function get_size to compute the number of bytes for type t, and writes the data via the serialization file handle, ser_fp.

The second callback, serial_struct (assigned as a callback for perfaction_struct), is called with the address of every **struct** reached during the traversal. The particular **struct** can be determined by examining t. Oftentimes, as is the case here, this function does nothing other than return 1, indicating the traversal should continue by visiting each of the struct's fields. Alternatively, the callback can return 0 to indicate traversal should not continue automatically, in which case the function can call the visit library function (explained in Section 2.1.5) to selectively visit fields.

The third callback, serial_ptr (assigned as a callback for perfaction_ptr), is invoked for each pointer when it is first reached. Here the type of in is **void** **, rather than **void** *, because it is the address of a pointer. For serialization, the callback checks whether the pointer is null; if so, it writes null to the file. Otherwise, it calls C-strider's lookup_addr function to check whether the pointer is to a global variable, i.e., one that has a symbol name. If a symbol name is returned, the code first writes 2, which we assume is not a legal pointer, followed by the symbol name. Otherwise there is no symbol, so the code writes the pointer value itself. If the pointer is to a string, as determined by its type, the code writes the contents of the string; note that if we were to dereference the pointer and continue as usual, we would only write the first character of the string. Finally, if traversal should continue to the pointer contents, then we return 1; else we return 0 to prevent further traversal though this

pointer.

As presented so far, the API has one limitation: If there are cycles among pointerful data structures in the heap, traversal may loop forever. Rather than force the programmer to add ad hoc logic to avoid this case, C-strider instead includes support for tracking previously visited pointers. Thus, the C-strider API includes a fourth callback, perfaction_ptr_mapped, which is called for each pointer that is reached but has been visited previously during the traversal. C-strider maintains a map from each visited *in pointer to its *out counterpart; this mapping is set when we return from perfaction_ptr , mapping the (original) value of *in to the final value of *out. When the traversal code considers a pointer to visit, it checks whether that pointer appears in the mapping table. If so, it calls perfaction_ptr_mapped; otherwise it calls perfaction_ptr . In the case of serialization, the corresponding serial_ptr_mapped callback function is not shown because it is quite similar to the serial_ptr function; the difference is that it knows the pointer cannot be null, so only the **else** case of the above code is needed, and the code for writing strings is elided, since the string has already been written. No further traversal is needed for a mapped pointer, so nothing is (ever) returned.

2.1.3 Implementing deserialization

While serialization is a traversal service that only reads the heap, deserialization is a transformation service that modifies the heap during traversal. It begins by writing global variables and then continues to initialize the heap as it processes

the serialization file. Deserialization is unusual because the input heap comes from prior program run, as captured in the serialization file. Thus, we will see that deserialization must call API functions that add mappings, read from this file, and direct the traversal. A more typical transformation service, like dynamic software updating (Section 2.5.3) or hot swapping [107], would simply traverse the existing heap while transforming it, requiring no extra programmer assistance.

Figure 2.3b shows the deserialization callbacks, which are passed to C-strider in an instance of **struct** traversal . The first callback, deserial_prim , is just like serialization, but reads data instead of writes it. deserial_struct returns 1, which means struct fields are traversed as usual, i.e., in the same order they were serialized.

The callback deserial_ptr inverts the logic from the serialization case, allocating new memory as needed and updating the mapping. It starts by reading the pointer itself from the file. Notice this is the address from the checkpointed program. Since the address space of the current run of the program may be different, the traversal uses the mapping table to map the checkpointed run's addresses to the equivalent ones in the current run's address space. There are several cases. The code first checks whether the pointer is 2. If so, it points to a symbol, whose name is then read and looked up to find its current address, which is assigned to *out. Otherwise, if the pointer is null, then null is written to *out. If this particular pointer was seen before—i.e., it was previously read from the file and therefore is in the mapping table—then *out is assigned to the value from the map, which is the corresponding address in the current run of the program.

Otherwise, the code starting on line 60 handles a non-symbol, non-null pointer

that has not been seen before. If it is a string, the code reads in that string, and then adds a mapping to between ptr (the address in the file) and tgt (the address of the same memory in the program). For non-strings, the code extracts the target type of the pointer and computes its size (if the pointer is to an array then the size of the target type will cover the entire array). Then it allocates memory for the target object, assigns the allocated pointer to out, and sets up the mapping. Finally, the code calls visit to manually direct to the traversal to the newly allocated memory. Thus, the next read from the file will write into that memory. In all cases the deserial_ptr function returns 0, since either no subsequent traversal is needed, or the traversal was manually triggered by calling visit .

The deserial_ptr_mapped function (not shown) will never be called, because translation of pointers from the serialization file always happens through manual lookups of the table on line 58. If a pointer from the serialization file is seen again in deserialization, then there is no attempt to traverse it (deserial_ptr returns 0).

2.1.4 Using serialization for checkpointing

Given the callback definitions, C-strider provides, or generates, the code that actually traverses the various elements of the heap, invoking the callbacks as it goes. To use a C-strider service, therefore, the programmer then needs to invoke the traversal at the appropriate place in their program.

Using the (de)serialization service is simple. Consider Figure 2.4, which uses this service for checkpoint and restart. Assume the data to be (de)serialized is stored

```
81 t1 *global;
82 t2 *other_global;
83
84 int main(int argc, char **argv){
85   if (! do_deserialize (argc,argv)){
86     /* do standard initialization */
87   }
88   /* long-running loop for
89            rest of the program... */
90   while(1){
91     if (...) checkpoint();
92     ...
93   }
94 }
95 int do_deserialize (int argc, char **argv){
96   if ((char)argv[1][0] == 'D'){
97     ser = fopen("ser.txt", "rb");
98     init (& deserial_funs , 0);
99     visit (&global, TYPE_t1, &global);
100    finish ();
101    fclose (ser);
102    return 1;
103  }
104  return 0;
105 }
106 void checkpoint(void){
107   ser = fopen("ser.txt", "wb");
108   init (& serial_funs , 0);
109   visit (&global, TYPE_t1, &global);
110   finish ();
111   fclose (ser);
112 }
```

Figure 2.4: Example program employing checkpointing.

in some variable global of type t1. When the program begins, it calls do_deserialize on line 85 to check whether the user has requested to restart at a checkpoint. If not, do_deserialize simply returns 0, and initialization continues as usual. Otherwise, the do_deserialize function invokes the deserialization traversal to initialize memory. First, C-strider must be initialized by calling init (line 98), which takes two parameters: a struct specifying which traversal functions to be used (having type **struct** traversal from Figure 2.2), and a flag specifying whether the traversal should be run in single-threaded mode, guaranteeing an in-order traversal (0), or parallel mode with multiple threads (1). In this example, for deserialization, the first init parameter is a **struct** traversal with the function pointers set to the C-strider API functions from Figure 2.3b, and the second parameter is 0 to signify a single threaded traversal, as (de)serialization relies on an in-order traversal to preserve the

order of the heap data. The init call also initializes some auxiliary data structures that C-strider uses to perform the traversal.

Next, the programmer selects which objects of the heap to traverse. One option is to call visit_all, a function generated by C-strider that traverses all objects reachable from global variables. Alternatively, as we do here (line 99), the programmer can call visit to traverse starting from a specific root. This is useful when a full heap traversal is not appropriate, e.g., here where we do not want to traverse other_global. Deserialization is a transformation service, but as mentioned earlier, it is an unusual one: the source heap is read from the serialization file, while the target heap is that of the current program. As such, we must specify out as &global. For in, we also use &global but this ends up not being important—as we can see from Figure 2.3b, the in argument is actually ignored by the deserialization code. (A more typical transformation service is discussed in Section 2.5.3.)

Lastly, the programmer calls finish, which tells C-strider to clean up the auxiliary data structures it uses for the traversal. At this point, all data has been deserialized, so the program closes the file, and do_deserialize returns 1 on line 102, thereby skipping from-scratch initialization.

The main program now proceeds, entering a long-running loop. Intermittently it calls checkpoint to serialize the current state. The function checkpoint (line 91) is similar to do_deserialize, except now the programmer uses the serialization functions, which write to the serialization file rather than read from it. Because serialization is a traversal service, we specify the in argument to visit as &global; the third argument is unimportant since serialization will ignore it, but we specify it as &global to be

```
// Directing the traversal
void init(struct traversal *funs, int paral);
void visit(void *in, typ t, void *out);
void visit_all(void);
void register_root(void * in, typ t);
void deregister_root(void * in);
void finish(void);
// Functions for querying the symbol table
char *lookup_addr(void *addr);
void *lookup_key(const char *key);
// Functions for manipulating the pointer
// mapping table
void add_mapping(void *in, void *out);
void *find_mapping(void *in);
```

```
// Functions for manipulating type
// information
int is_prim(typ t);
int is_ptr(typ t);
int is_funptr(typ t);
int is_array(typ t);
int get_size(typ t);
int get_num_array_elems(typ t);
int get_ptrtype(typ t);
// Constructors
typ mktyp_ptr(typ t);
typ mktyp_arr(int arrlen, typ t);
// Generics
int get_maintype(typ t);
typ * get_generic_args(typ t);
int get_num_gen_args(typ t);
typ mktyp_instantiate(typ t, int nargs, typ *args);
```

Figure 2.5: C-strider API.

compatible with the expectations of automatically generated code (cf. Figure 2.7 in the next Section). For traversal services such as serialization, a call to visit_all will automatically set out equal to in for all elements of the traversal.

2.1.5 Full C-strider API

We now wrap up our discussion of C-strider's basic design by discussing the remainder of the API, given in full in Figure 2.5. Section 2.3 will discuss in more detail how these functions can be used to customize the generated traversal.

Starting in the upper-left, the first set of functions controls the initialization (init), starting points (visit or visit_all), and conclusion of the traversal (finish), as briefly explained in Section 2.1.4. Figure 2.4 showed how the visit function can be called on any location to start traversal from that point onward. Certain applications, however, need to traverse the entire heap. For example, full heap

traversal is needed for dynamic software updating and for heap profiling. To support these cases, C-strider provides a function visit_all , which traverses the entire heap automatically by calling a list of generated custom traversal functions for each global variable in the program. Additionally, the programmer can use register_root and deregister_root to add and remove, respectively, additional locations for visit_all to visit. For example, we use this to traverse certain local variables in dynamic software updating, discussed in Section 2.5.

The next two functions, lookup_addr and lookup_key, perform lookups in the C-strider symbol table, mapping from symbol names to addresses or vice-versa. The last two functions on the left, both of which we saw above, manipulate the mapping table used in transformation services. While we could have left it up to the user to implement these features on an ad hoc basis, we incorporated them into C-strider because we found them useful for a range of services.

Finally, as we saw in the serialization and deserialization code, one of the major features of C-strider is that it is type aware, which not only allows the traversal to be precise, but also allows service code to adjust its behavior by type. For example, strings of TYPE_PTR_CHAR are serialized differently than other pointer types. In general, C-strider generates a set of type names for those types that are statically present in the code, e.g., TYPE_INT, TYPE_PTR_INT. User-defined types also available at run time, e.g., TYPE_STRUCT_DLIST for a program that includes **struct** dlist .

The functions on the right of the figure manipulate types. The first group of functions manipulate standard types, e.g., determining whether they are primitives

(is_prim) or pointers (is_ptr), calculating their size (get_size , get_num_array_elems), and returning the pointed-to type (get_ptrtype). The particular set of queries shown here was derived from our experience building several services (Section 2.5); other applications could potentially require other accessors, which are easy to add.

The next two functions, mktyp_ptr and mktyp_array, create new pointer or array types, respectively, at run time. We use these to maintain type information for arrays and other pointed-to blocks whose length is only determined at run time. We will see an example in Section 2.3. The last set of functions lets us create and query generic types, e.g., parametrically typed linked lists. Section 2.3 discusses generics in detail.

Notice that, since types are available at run-time and can be fully queried, it is not strictly necessary to have four perfaction callbacks. In theory, perfaction_prim , _struct , and _ptr could be combined into a single callback that would just test the type and behave appropriately. However, we have found this particular grouping to be useful, because we often want to treat each of those groups independently (e.g., as in serialization and deserialization).

2.2 Type-aware Traversal

Section 2.1 showed C-strider from the service developer's perspective. This section provides details on the inner workings of the traversal, in two parts. First, we discuss the main components of the traversal, which are the visit function, which is called for each element of the heap, and the *task queue*, which organizes the work

of the traversal. Second, we explain how the visit function relies on several bits of C-strider-generated code, in particular a table of type representations, a set of struct-specific visit functions, and finally a visit_all function, which can be used to initiate a heap traversal from the program's roots.

2.2.1 Main components of the traversal

To perform the traversal, C-strider must determine which perfaction_ function to call with which addresses, and it must keep track of those addresses it has visited and the addresses to visit next. The overall traversal is orchestrated by two pieces of code: the visit function, shown in Figure 2.6, which calls the appropriate perfaction_ function based on an object's type; and a task queue that contains pointers to **struct**s or arrays whose elements need to be visited.

Traversal visit function: The visit function is the workhorse of the traversal. It is invoked for each visited heap object, starting with the roots. As arguments, visit is given a pointer to the heap object to consider, a representation of that object's type (expressed as an object of type typ), and a pointer to the corresponding object in the transformed heap. We defer discussion of the implementation of type representations to the end of this section.

The body of visit is straightforward. For primitive types (**int**s, **char**s, etc) and other non-standard terminal primitive types (mutex_t, time_t, size_t), the traversal calls perfaction_prim on line 115, which in turn calls the user-provided callback from **struct** traversal , passing it a pointer to the primitive being traversed.

If typ is a pointer type, the traversal returns immediately if the pointer is null (line 117). Otherwise, there are two cases, depending whether the pointer has been visited before (line 119).

If the pointer has been visited, the code calls perfaction_ptr_mapped, passing the address from the mapping as its last (out) parameter. The code passes &lookup because perfaction_ptr_mapped takes a **void** **, i.e., a pointer to a location containing the pointer of interest. The code then writes the mapped-to value from lookup to *out.

If the pointer has not been visited, then the code calls perfaction_ptr and updates the mapping to record that the pointer has been visited. Then either traversal stops (if perfaction_ptr so indicated, or if t is a function pointer), or it continues at the pointed-to type (lines 127– 128).

If the type is a **struct**, the traversal calls perfaction_struct . If that function indicates the traversal should continue, the code calls enqueue, which creates a task that will visit t's fields and adds it to the queue.

Finally, if typ is an array type, the traversal likewise enqueues a task to recursively visit each of the array's elements. When performing a multi-threaded traversal, this code will divide this task into sub-tasks covering sub-ranges of the array; the number of tasks depends on the size of the array and the number of processors on the machine.

Note that the visit function skips unions. We made this choice because C has no standardized mechanism for determining which arm of a union is active. Thus, for these cases the developer has to customize the traversal with program-specific

```
113 void visit (void *in, typ t, void *out){
114   if (is_prim (t)){
115     perfaction_prim (in, t, out);
116   } else if (is_ptr (t)){
117     if (!in) return;
118     void *lookup;
119     if ((lookup = find_mapping(*(void**)in))) {

120       perfaction_ptr_mapped(in, t, &lookup);
121       *(void**)out = lookup;
122     }
123     else{
124       int retc = perfaction_ptr (in, t, out);

125       add_mapping(*(void**)in, *(void**)out);

126       if (!retc || is_funptr (t)) return;
127       typ t_p = get_ptrtype(t);
128       visit (*(void **)in, t_p, *(void **)out);

129     }
130   } else if ( is_struct (t)){
131     if ( perfaction_struct (in, t, out))
132       enqueue(in, t, out);
133   } else if (is_array (t)){
134     /* enqueue task that calls visit on each array element */
135 } }
```

Figure 2.6: Traversing using type information.

code (Section 2.3).

The task queue: The task queue consists of traversal work that remains to be done. It contains two types of tasks: structs whose fields need to be visited, and arrays whose elements need to be visited. The traversal in the former case is defined by a custom visit function that C-strider generates for each **struct** in the program (as described in Section 2.2.2). The traversal in the latter case is simply a loop over the specified range of the array. Either way, the traversal function simply calls visit on each of the elements of in and out and then returns.

Single and multi-threaded traversal modes: The handling of enqueued tasks depends on the mode C-strider is used in. In single-threaded mode, the main thread repeatedly pulls tasks off the queue and calls `visit`, until the queue is empty, forming a breadth-first search of the enqueued tasks. Using a queue enables C-strider to traverse cyclic data structures (which in C must always go through a **struct**) without requiring a deep stack.

To speed up the traversal process, C-strider supports multi-threaded traversal. In this mode, C-strider launches $n-1$ worker threads in addition to the main thread, where n is the number of processors. The threads consume tasks that are generated by `enqueue`, which are initially produced by C-strider's main thread.

C-strider uses a work-stealing scheduler: Each thread has a local queue (created initially at the call to `init`), and when it runs out of tasks it attempts to steal one from another queue [19]. Each queue is implemented as a large pre-allocated array and a single lock used for enqueue and dequeue operations. Each queue also maintains an exact number (using the lock) of in-flight tasks currently being processed. This is because even if all queues appear to be empty, if a thread is currently processing a task, it may enqueue new tasks. Therefore, we are not done traversing until all queues are empty and no tasks are in-flight.

In a multi-threaded traversal, the map of visited locations needs to be thread-safe. Using a single lock to protect it would introduce significant contention, so we use a hash table implementation with bucket-by-bucket locking.

A traversal completes once the service calls `finish`. At this point, the main thread and helper threads (if multi-threaded) process all remaining items in the task

```
struct dlist { struct dlist * next, *prev; int x; }
struct dlist *head = ...;
struct dlist **phead = &head;
```

```
139 void _visit_struct_dlist (void *in, typ t, void *out) {
140   struct dlist_i *in_l = in;
141   struct dlist_o *out_l = out;
142
143   visit (&in_l→next, TYPE_dlist_PTR, &out_l→next));
144   visit (&in_l→prev, TYPE_dlist_PTR, &out_l→prev));
145   visit (&in_l→x, TYPE_INT, &out_l→x);
146 }
```

Figure 2.7: Generated traversal code for **struct** dlist .

queues until they are empty. Once complete, temporary data structures are freed and the helper threads exit.

In C-strider we have made a specific design choice to only create tasks for structs and arrays. As an alternative design, we could have enqueued tasks for every element in the heap, including pointers and primitives. However, the cost of making a task, queuing it, dequeuing it, and executing it, would dwarf the cost of just executing the task directly, despite the lost opportunity for parallelism. We find that our current design generally balances overhead with opportunities for parallelism.

2.2.2 Generated code

In addition to the visit function and queues, C-strider relies on per-program generated code to perform the traversal, including type representations, a set of traversal functions for each **struct** in a program, and code to register globals to be visited by visit_all .

Type representations and the type table: C-strider traversals use *type representations* that describe each type in a program. Such type representations cover both built-in types (like **int** and **char**) and user-defined types. Type representations have type typ, and are used by both C-strider library functions like visit and the perfaction callback functions provided by the service developer with **struct** traversal.

A typ t is represented as an integer that indexes a generated *type table*. This table maps types to relevant information about them like their size in bytes, the number of elements (for arrays), the pointed-to type (for pointers and arrays), and, for a generic type, the base type and the type arguments this type was instantiated with (if applicable); Section 2.3 says more about generics.

C-strider creates a unique typ for each type appearing (statically) in the program text and gives it a predictable variable name, e.g., TYPE_PTR_CHAR from Figure 2.3a. A service developer can refer to such names in the perfaction functions. Types are extracted from the source program by an analysis implemented in CIL [72]. CIL takes as input preprocessed C source code and outputs information about the types of all variables and functions of the program.

C-strider also assigns a distinct typ to each type alias, e.g., **typedef int** size_t in the source would yield a new type TYPE_size_t with the same associated information as TYPE_INT. Distinguishing type aliases like this is useful for certain applications. For example, we found that Redis includes a string type that is aliased to **char** $*$, but is actually a pointer to the middle of a data structure. Thus, we can write the traversal so that when it visits this particular named type, it visits the structure as a whole and not just the string.

A traversal can generate type representations on the fly, using the mktyp_ functions (cf. Table 2.5). If the run-time generated type matches one already in the type table generated statically, then mktyp_X will return the associated index. However, if the type did not appear in the program text then it will not be in the table already. In this case, a new entry is added to the table dynamically, and the index to that element is returned.

Traversing structs: As seen in the previous section, the visit function directs the traversal by handling the pointers and calling the appropriate perfaction_ functions. To traverse inside a struct, C-strider must call visit for each field of the struct. To assist with this, in addition to generating the type table, C-strider generates a traversal function for each struct and each field within the struct. For example, Figure 2.7 shows the traversal code for a doubly linked list **struct** dlist, defined at the top of the figure. (Code slightly simplified for clarity.) Each generated traversal function is named by prepending _visit to the name of the struct. The parameters **void** *in and **void** * out are the addresses of the struct being traversed, and parameter typ t is used in the case of generics, as described in Section 2.3.1. C-strider generates code to assign the parameter addresses to the type of struct being traversed (shown on lines 140 and 141) to allow the memory to be accessed properly as struct fields. Finally, C-strider generates code to traverse each field of the struct (shown on lines 143 - 145) by getting the type information for each field from CIL. C-strider inserts the corresponding type name generated according to the pattern used when generating type representations (e.g., TYPE_dlist_PTR for

struct dlist ∗) and generates a call to visit for each field of the struct.

visit_all: During compilation, C-strider gathers a list of all non-static global variables of a program using CIL. When init is called, C-strider uses the list of non-static globals to populate hash tables mapping symbol name to address (for lookup_key from Figure 2.5), and address to symbol name (for lookup_addr). After initialization, if the user then calls visit_all , C-strider iterates through all entries in the lookup_key table, combines each entry with some auxiliary type information, and uses that information to call visit on each global. After visit_all calls visit on all globals in the hash table, it then calls visit on all roots the programmer has manually added with register_root . C-strider does not process static global variables, as discussed in Section 2.4.

2.3 Customizing the Traversal

For many C programs, the standard C types do not provide quite enough information to support type-accurate traversal. To support such programs, C-strider lets the programmer customize the traversal in two ways: adding type annotations to the program (Section 2.3.1), and writing program-specific code in perfaction functions (Section 2.3.2).

2.3.1 Type annotations

C's type system is not sufficiently expressive to describe many common programming idioms. For example, the type **int**∗ could describe a pointer to a single

```
typedef struct _dlist_arr {
  int arrlen ;
  struct dlist ** T_PTRARRAY(self.arrlen) arr_head;
} dlist_arr ;
```

```
151 void _visit_struct_dlist_arr (void *in, type t, void *out) {
152   struct dlist_arr *in_a = in;
153   struct dlist_arr *out_a = out;
154    visit (&in_a→arrlen, TYPE_INT, &out_a→arrlen);
155    visit (&in_a→arr_head,
156      mktyp_ptr(mktyp_arr(in_a→arrlen,TYPE_STRUCT_dlist_PTR)),
157      out_a→arr_head);
158 }
```

Figure 2.8: Length annotation and generated traversal code.

integer, or a pointer to an array of integers. C-strider permits programmers to express additional information as type annotations to remove some of the ambiguity. These annotations are borrowed from Kitsune and inspired by Deputy [24]. Currently, C-strider supports two kinds of type annotations that encode the most common missing type information: lengths of arrays, and types of **void** *'s used in generic data structures. These annotations can decorate types of either **struct** fields or global variables, in which case they modify the generated traversal function for that **struct** or global, respectively. We illustrate the annotations by example.

Type annotations for arrays: Depending on how an array is declared, C-strider may or may not be able to statically determine its length. If an array is declared with static length information such as **struct** dlist array_x[6], C-strider can determine that it needs to process 6 elements for array_x. However, when a user declares an array with dynamic length, such as **struct** dlist ** arr_head, C-strider cannot statically determine the length of the array and needs additional information to

```
struct list {                          // class List<T> {
  void T_VAR(@t) *val;                 //   T val;
  struct list T_INST(@t) *next;        //   List<T> next; }
} T_FORALL(@t);
```

```
163 void _visit_struct_list (void *in, type t, void *out) {
164   struct list *in_l = in;
165   struct list *out_l = out;
166   typ *args = get_generic_args(t);
167   int num_args = get_num_gen_args(t);
168   assert(num_args == 1);
169   typ t0 = args[0];
170
171   visit(&in_l->val, t0, &out_l->val);
172   visit(&in_l->next,
173     mktyp_instantiate(TYPE_STRUCT_list_PTR,num_args, args),
174     &out_l->next);
175 }
```

Figure 2.9: Generic annotation and generated traversal code.

determine how to process arr_head. C-strider includes annotations T_PTRARRAY(S) and T_ARRAY(S) to decorate a pointer or array, respectively, with a length S, which may be a constant integer or an expression of the form self.f, where f is a field at the same level of the current **struct**. For example, Figure 2.8 shows an annotated **struct** type declaration and its corresponding traversal code. The annotation states that arr_head is a pointer to an array whose length is contained in the field arrlen of the same structure (self). Notice that line 156 of the generated traversal code calls mktyp_arr to generate an array type of the appropriate length, and then wraps it in a pointer type.

If the user does not annotate an array, C-strider will presume it is a singleton. From the example in Figure 2.8, the code **struct** dlist ** arr_head; would be treated by C-strider as a single instance of a "pointer to a **struct** dlist pointer," rather than

a "pointer to array of **struct** dlist pointer" with the annotation.

Type annotations for generics: Because C-strider uses static type information to generate the heap traversal code, **void** ∗'s in a program present a roadblock to the traversal generation process as **void** ∗'s provide no type information. C-strider includes several annotations to type **void** ∗'s in generic data structures. Figure 2.9 shows a generic linked-list data structure and its corresponding traversal code. Here, the **struct** list type is parameterized by type variable @t, introduced with T_FORALL. The type variable is used with T_VAR to provide the actual type of val. Then the code uses T_INST to instantiate the type of next. For comparison, the Java generic linked list equivalent is shown in comments in the example. In the traversal code, the argument t is an instantiation of the generic type. The calls on lines 166 and 167 get an array with the instantiated arguments and the length of that array, respectively. The code then binds t0 to the first element of the array, i.e., whatever @t is instantiated as. That type is used to visit val, and then next is visited at the type of **struct** list instantiated with the same arguments (line 173).

If the user does not add an annotation to a **void** ∗, C-strider prints out a warning during the type generation process, reminding the programmer to add an annotation if necessary. If no annotations are added to a **void** ∗, C-strider does not traverse the pointer as no type information is available.

```
struct tagged_union {
   int tag; /* 1 selects u.x; 0 selects u.c */
   union{ int x; char c; } u; /* selected by tag */
};
```

```
180 int current_service ;
181 #define DESERIALIZE 0
182 #define SERIALIZE 1
183
184 struct traversal deser_prog_funs = {
185  . perfaction_prim = &deserial_prim,
186  . perfaction_struct = &s_d_prog_struct,
187  . perfaction_ptr = &deserial_ptr,
188  .perfaction_ptr_mapped =
189      &deserial_ptr_mapped
190 };
191
192 int do_deserialize (int argc, char **argv){

193   if ((char)argv [1][0] == 'D'){
194      current_service = DESERIALIZE;
195      ser = fopen("ser.txt", "rb");
196      init (&deser_prog_funs, 0);
197      ...
198   }
199   return 0;
200 }
```

(a) Custom deserialization initialization code

```
201 int s_d_prog_struct (void *in, typ t, void *out){
202   /* Service-agnostic, Program-specific */
203   if (type == TYPE_STRUCT_tagged_union) {
204     struct tagged_union *in_u = in;
205     struct tagged_union *out_u = out;
206      visit (&in_u->tag, TYPE_INT, &out_u->tag);

207     if (in_u->tag)
208       visit (&in_u->u.x,TYPE_INT, &out_u->u.x);
209     else
210       visit (&in_u->u.c,TYPE_CHAR, &out_u->u.c);
211     return 0;
212
213   /* Service-specific */
214   } else if ( current_service == DESERIALIZE){

215      /* Program-specific */
216      if (t == TYPE_...) {...}
217      /* Program-agnostic */
218      else return deserial_struct (in,t,out);
219   } else if ( current_service == SERIALIZE) {

220      ...
221     return serial_struct (in, type, out);
222   }
223   return 1;
224 }
```

(b) Custom (de)serialization struct callback

Figure 2.10: Customizing traversal in perfaction functions.

2.3.2 Customization in perfaction functions

In some cases, type annotations are insufficient to guide the traversal. For example, consider the type at the top of Figure 2.10, which defines a **struct** whose u field is either **int** or **char**, depending the tag field being either 1 or 0, respectively.

To select the correct field of u to visit, C-strider needs to know how to interpret tag.

To solve this and other problems, the programmer can write program-specific perfaction functions. Such functions (contained in file prog_specific .c in Figure 2.1) can assist C-strider in performing the traversal in a service-agnostic manner, per our example. Customizations can also adjust a service's functioning based on the particular program. For example, we might know that our program has an array of structs that all reference each other. Rather than traverse the entire array, we could customize serialization to write the whole array at once, and thereby avoid further traversing of the structs. When no customization is needed, the programmer can just delegate to the relevant service code.

Figure 2.10 provides a modified example of deserialization customized to handle our example tagged union. On line 180, **int** current_service keeps track of the current service, either SERIALIZE (1) or DESERIALIZE (0); we define this flag because both services will share code (though we focus on the code for deserialization, since serialization is similar). On line 184, **struct** traversal deser_prog_funs sets up the callback functions for the new set of program-specific deserialization functions. For perfaction_struct we specify a program-specific function, s_d_prog_struct (defined on the right side of the figure), while for the rest we use the standard functions from Figure 2.3b. The function do_deserialize (line 192) is similar to the previous version (Figure 2.4, line 95), with the addition of setting current_service on line 194 and passing deser_prog_funs to init on line 196. The checkpoint function (not shown) is similar to the version on line 106, with the addition of setting current_service = SERIALIZE and passing ser_prog_funs to init (which are initialized

analogously).

The right side of Figure 2.10 shows the program-specific implementation of s_d_prog_struct , which acts as a wrapper around serial_struct and deserial_struct . The first part of the function is program-specific, but service-agnostic: If s_d_prog_struct is called on **struct** tagged_union, then we visit the tag on line 206, and then switch on the tag and call visit on the appropriate sub-field, returning 0 on line 211 to indicate the default traversal (visit all fields) should not happen. Otherwise, for any other **struct** type, we check which service we are running and perform either a customized action or the default service action. For example, on line 214, if current_service is DESERIALIZE, we first perform service-specific, program-specific actions, e.g., we check for a particular type (line 216) to optimize serialization for it (not shown). Otherwise, on line 218 we call the default action, deserial_struct , which is a wrapper for the reusable service-specific code from Figure 2.3b, line 44. Similar to deserialization, the code for serialization (or any other service) performs service-specific checks, and then the default action.

Section 2.5.1 includes more details and examples of customizing the traversal for service specific customization.

2.4 Limitations

While C-strider is general and flexible, it does have some limitations, which we discuss here.

A general problem is that by being a generic framework, C-strider's perfor-

mance might suffer compared to a purpose-built service for a particular application. For example, C-strider's serialization/deserialization service will, by default, serialize an array by serializing each of its elements individually. But if those elements contain no pointers, then it would be faster to simply write the entire array directly (similarly to how C-strider serializes strings). C-strider also uses generated type representations which might not otherwise be needed for a purpose-built service. That said, C-strider does allow programmers can customize a service to their program, to improve performance.

C-strider is intended traverse the heap, but it may come across pointers that refer to other parts of memory, such as stack-allocated variables. C-strider does not reliably provide information to a service developer about whether a pointer is to the heap or not, but in many cases this can be determined simply by looking at the pointer's address; e.g., on Linux, the heap is in "low" memory and the stack is in "high" memory. If necessary, the programmer could write a `perfaction_` rule based on the memory address of the item being traversed, instructing C-strider to perform some custom action (such as printing an alert) if the traversed location is an unexpected memory location.

In principle, C-strider could be used to produce a type-aware garbage collector for C programs. Doing so would be challenging in practice, however. For one thing, C-strider would need assistance in finding all of the roots of the heap, which includes pointers on the stack, thread-local storage, etc. At the moment, this would require the programmer to explicitly register and deregister these roots (using `register_root` and `deregister_root`) when they come into and go out of scope. There is also an issue

with roots that are static variables, discussed below, and with any roots that might be stored by external libraries. Some of these problems could be overcome through a source-to-source transformation that registers and deregisters roots automatically, similar to the one proposed for Kitsune.

Another limitation of C-strider is that it cannot properly traverse variables declared as bit fields, because C does not allow taking the address of bit fields. For example, C-strider does not generate traversal code for a structure with a element field of **unsigned int** valid : 1; or else the generated code visit (&in→valid ... , would not compile. In our experiments, we modified the program code to not have bit fields.

C-strider generates the visit_all function by analyzing all of the source files in a program, and generating a single function that registers its global roots. This function will not have access to **static** variables in other files. The programmer either will need to make variables non-static, or only initiate traversals (using visit) on variables that are in-scope.

2.5 Applications and Experiments

We used C-strider to develop four services: *serialization*; *state transformation* for dynamic software updating; *heap profiling*; and *heap assertion checking*. We implemented each service for a subset of three programs, listed on the left of Table 2.1 along with their version numbers and sizes. Memcached is a widely used, high-performance data caching system employed by sites such as Flickr and YouTube.

Redis is a key-value database used by several high-traffic services, including Instagram and stackoverflow. Snort is a network intrusion detection system claiming millions of downloads and nearly 400,000 registered users.

This section describes the implementation of our services, characterizes the programmer effort required to write them, and provides some performance measurements. Measurements were conducted on a 32-bit, Intel Core i5-3320M at 2.60GHz, with 4 cores and 7.5GB mem, running Ubuntu 12.04. Medians were taken over 11 trials. Since C-strider adds no overhead to normal program execution (e.g., because it does not compile the program any differently), we consider the time from when the traversal-based service is requested until it completes and normal execution resumes.

2.5.1 Programmer effort

Table 2.1 tabulates three kinds activities involved in using C-strider: writing service code, customizing a particular program's traversal, and customizing the traversal in a service-specific manner. All three cases involve relatively little code, much of which is a one-time effort that can be shared across different services and/or different programs.

Writing a service: Writing a C-strider service takes relatively little effort, in terms of lines of code. The parenthesized number at the top of the last four columns of Table 2.1 count the lines of service-specific (but program-independent) code, tabulated by service (dynamic software updating, serialization, profiling, and heap assertions). Details of the implementation of each service is given in the following

Prog.	LOC	Traversal T_*	Traversal Cust.	DSU (24)	s/d (78)	prof (65)	asrt (0)
Memcd 1.2.3	~4K	6	0	17	44	9	20
Redis 2.0.2	~13K	45	26	19	46	11	36
Snort 2.9.2	~215K	143	21	n/a	n/a	4	67

Table 2.1: Programmer effort (measured in LOC)

subsections.

Customizing a program's traversal: While some programs can use a service out of the box, nontrivial programs will require some customization. The first step is customizing the traversal for that program, in a service-independent manner, as described in Section 2.3. The third column of Table 2.1 counts the type annotations we added, and the fourth column counts the lines of program-specific perfaction_ code we wrote. Nearly all of code we wrote was for handling **union**s or union-like data structures. For example, of the 26 LOC for the custom traversal for Redis, 17 LOC customizes the traversal of its main database item structure which contains a flag that determines the type of a **void** $*$ field in the structure. The other 9 LOC handles a structure containing a mask that determines whether other fields in the structure are valid.

Customizing the services: The lower right portion of the table counts the lines of code that are program- *and* service-specific. For example, we wrote 17 lines of code specific to Memcached for dynamic software updating. We write "n/a" where

```
225 int  perfaction_struct (void *in, typ t, void *out){
226     if( current_service  == DYNUPDATE){
227       if(t == TYPE_STRUCT_stats){
228         if (lookup_addr(out) != NULL){
229           int in_sz = get_size(t);
230           int out_sz = get_out_size(t);
231           assert( in_sz == out_sz);
232           memcpy(out, in, in_sz );
233         }
234         return 0;
235       } else return update_struct(in, t, out);
236     }
237     if( current_service  == SERIALIZE){
238       if(t == TYPE_item){
239         struct _stritem_cpy * it = in;
240         int len_total = ITEM_ntotal(it);
241         // write len_total and other struct fields to disk
242       } else return serial_struct (in, t, out);
243 }}
```

Figure 2.11: perfaction code for Memcached.

we did not implement a service for that program.

We implemented the program- and service-specific code using the pattern shown in Figure 2.11, which excerpts part of the perfaction_struct function for Memcached. This code switches based on the current service (lines 226 and 237). For each service, it then either performs type-specific actions (lines 227 and 238) or calls the service-specific action (lines 235 and 242). The current_service flag is set through the entry point to the service, e.g., the call to do_serialize in Figure 2.4 sets the service to deserialization, and the call to checkpoint sets the service to serialization (both calls initiate a full traversal after setting the flag).

2.5.2 Heap serialization

We now turn to the different services we implemented, starting with heap serialization and deserialization suitable for implementing checkpointing. The implementation of this service was presented in Section 2.1 (Figures 2.3a and 2.3b). In addition to those 69 lines of code, we have an additional 16 lines of code that deal with file manipulation (opening, closing, and some wrapper functions around fread and fwrite for simplicity). We must use a single-threaded traversal for this application to ensure data is written to the serialization file in a deterministic order.

While implementing serialization for Memcached, we developed one interesting performance optimization, shown in Figure 2.11. Here on line 240 we use a macro from Memcached to get the size of a *flexible array member* (**void** * end[]) contained in **struct** item; the length of this member is the sum of several of the item's fields. By tailoring the traversal here, we can write the entire item to disk at once rather than traversing each byte of the flexible array separately. We also wrote a similar function for deserialization. In total, we wrote 44 additional lines of perfaction_* code to optimize the traversal of Memcached's key-value database structures, as shown in the sixth column of Table 2.1. We performed very similar changes to optimize the Redis object structure for serialization, totaling 46 additional lines.

Time required for serialization: We measured the time it takes to serialize and deserialize the key/value database items of Redis and Memcached. (We did not serialize the rest of the program, such as statistics or connected user information, as

this information would be stale between program restarts.) Figure 2.12 shows how performance varies with the size of the heap, in terms of number of key-value pairs. The keys and values are approximately 10B in length each. (We say "approximately" because they consist of a string appended with an incrementing integer.) The parts of the heap that we traverse for serializing 30K key-value pairs contains 5,592KB of allocated data structures in Redis and 1,477KB for Memcached. We see that serialization and deserialization take nearly the same amount of time for a given program. Overall, Memcached traversal is faster; the reason is the performance optimization mentioned above, which lets C-strider write the key and value to disk as a single block. In contrast, Redis stores its key and value in separate structures that must be traversed separately.

Redis itself provides a serialization tool, allowing the user to load/store entries from/to a file. The Redis serialization process uses a custom iterator to skip to only populated elements of the array, serializing 20K (~10B-key,~10B-value) pairs to disk in ~13ms. This is in contrast with C-strider, which must visit every entry of the database array (2^{15} slots for 20K database entries) and maintain the mapping table, taking ~123 ms to serialize the same 20K entries. We suspect we could customize our traversal to apply similar optimizations but have not investigated further.

2.5.3 State transformation

C-strider grew out of our experience with Kitsune. As described in Section 1.1.2, DSU services are implemented by loading the new code (for Kitsune,

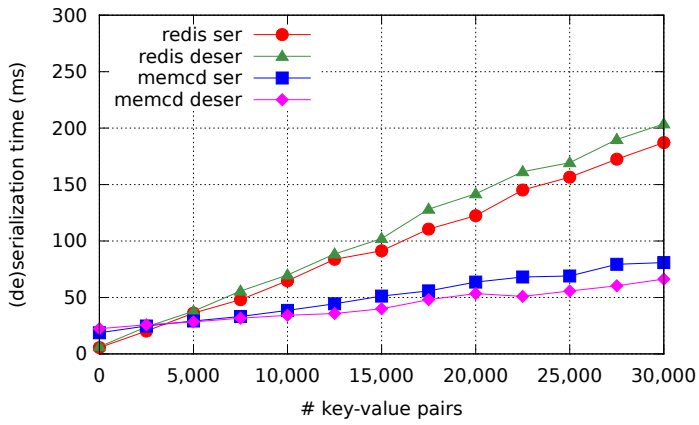

Figure 2.12: Serialization times (color plot).

compiled to a shared object) into the program, and then transforming the existing state (i.e., heap memory) to meet the expectations of the new code. For example, in the old version of the program a **struct** foo might have two fields, while in the new version it has three. The state transformation code must find all pointers to **struct** foo objects in the program, allocate new memory for those objects, initialize retained fields to the existing values and the new field to a new value, and then redirect the pointer to the new object after freeing the old one.

State transformation in C-strider: Kitsune state transformation can be implemented as a C-strider transformation service in 24 LOC. This service traverses and modifies the program heap (thus using both the in and out parameters of the perfaction_ functions with the old version of the program as the in set of roots and the new version of the program as the out set of roots). Additional code (outside of C-strider) was required to implement other elements of DSU, e.g., for loading in the new code and managing updating timing. We made one generalization to

the C-strider API to support updating: we defined two variants of lookup_addr and lookup_key (cf. Figure 2.5), one for the old version's symbol table, and one for the new version. We also needed to track the size information for the new type's size, so we added a function get_out_size to get the corresponding size of the new version of a type for the DSU case. (Notice this is a DSU-specific function because it needs to understand types from a different program version.)

The DSU perfaction_ functions do one of several things:

- When reaching a location in the new program that must be initialized from the old program (such as a global variable in the new program's data segment), the code simply memcpys the appropriate bytes over.
- When first reaching a pointer to a block of a type that has changed size, the code allocates a new block to hold the new, differently sized data, and adds a mapping from the old block to the new block to C-strider's hash map of visited locations. The program-specific action is then called to initialize the new block appropriately.
- When *re*-visiting a pointer to a malloc'd block for a type that has changed size, the code looks up the corresponding new block address and overwrites the old pointer with it (similarly to Figure 2.3b, line 58).
- When visiting a pointer corresponding to a known symbol, the code updates it with a pointer to the matching symbol from the new program version (similarly to Figure 2.3b, line 53).

The program-specific actions perform the actual state changes. Referring

back to our example of **struct** foo at the top of this section, we would customize perfaction_struct to look at type t, and if it is TYPE_STRUCT_foo we allocate new memory, initialize it with the retained values and the new one, and then write the result to *out.

For the programs we considered, the data representations did not change between versions, so the only program-specific DSU code we wrote simply optimized the traversal. For example, for Memcached, Figure 2.11 (line 227) shows how we cut off the traversal of **struct** stats since it contains only primitives that need not be traversed individually (when it is stored in a global variable, it must still be copied to the new code's address space).

As mentioned briefly in Section 2.1.5, C-strider allows local variables to be added as roots of the traversal. This ability is helpful in dealing with the runtime stack. We used this feature to support updates to Memcached—in particular, we inserted code to register local variables of the main function. These variables store information that the user supplies as command line options. Registering these local variables ensures that the command line option state is propagated to the next version when we call visit_all at the end of the main function. In total, we registered 7 local variables for Memcached.

Time required for a dynamic update: We measured the time it takes to deploy a dynamic update with single-threaded and multi-threaded traversal (4 threads). We also compare against the time for the same update with Kitsune.

Figure 2.13 shows how update times vary with heap size measured in terms of

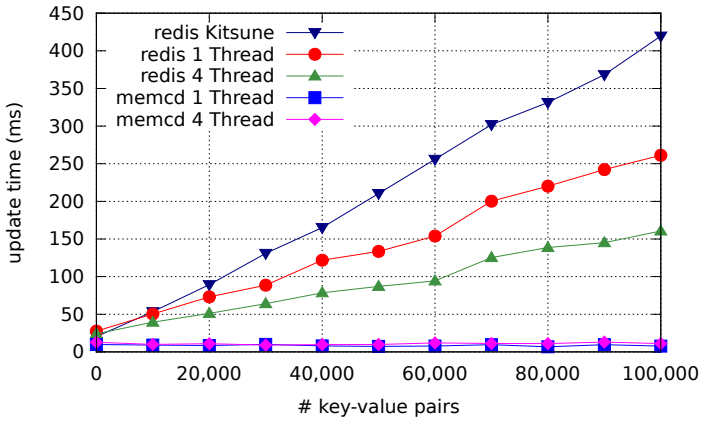

Figure 2.13: DSU state transformation times (color plot).

(~10B-key,~10B-value) pairs in the database for both Redis (version 2.0.1 to 2.0.2) and also for Memcached (version 1.2.2 to 1.2.3), using either one or four threads. For Redis we see that the benefit of parallelism increases with heap size, whereas with Memcached performance is flat. Investigating further, we found that Memcached has a very limited traversal because the majority of its heap can be left unmodified and thus not traversed at all.

Updating Redis with 4 threads took an average of 68% of the time taken with a single thread, with the speedup improving as the number of key-value pairs increases. For Redis we do not achieve perfect speedup as the majority of Redis' state is in linked lists, effectively serializing a large amount of the traversal.

C-strider's implementation of dynamic updating performs better than Kitsune on Redis. Kitsune's traversal is more heavyweight, allocating several data structures with type information for each item traversed, and several additional data structures for each instance of generic types. Redis makes heavy use of generics, which results in the allocation of a total of 17.16MB worth of data structures when updating

Redis with 100,000 key-value pairs with Kitsune. Rather than create and allocate generic type information for each generic instance like Kitsune, C-strider stores type information and reuses it for each entry by looking it up in the type table, so it does not incur additional overhead.

Memcached updated with a parallel traversal is slower than when using a single-threaded one, running at an average of 2.3ms slower than the original time across all trials, due to the overhead of using multiple threads. However, the update time for Memcached is quite small to begin with.

2.5.4 Heap profiling

It is often useful to profile a program's behavior when optimizing its performance. With C-strider, we implemented a type-aware heap profiler that is able to give accurate counts (numbers, and total bytes) of objects present in the heap. This information can be useful for, say, finding the leading sources of bloat. The implementation of our profiler is straightforward: the perfaction_ code simply keeps a hash table that maps the type `t` of a visited item to a pair tracking the total count and size in bytes. We are aware of only one prior C-heap profiler that provides such fine-grained per-type information [68].

We used our profiler to improve dynamic update times (Figure 2.13). The profiler showed what **struct**s had the most instances, which we then manually inspected. If the high-count **struct**s did not have any fields that needed to be updated or traversed (such as a structure with only primitive values), we then wrote perfaction

Program	Total calls to perf	Trav. (ms)	SIQR (ms)	Uniq types	Alloc'ed (KB)
Memcd 1.2.3*	81,844	49.36	(7.14)	65	360
Redis 2.0.2*	494,389	203.21	(16.71)	92	2,189
Snort 2.9.2+	10,299,744	4412.51	(194.71)	1,019	138,417

*For 4,000 entries +For traffic at 30 pkts/s

Table 2.2: C-strider profiling heap data

rules directing the traversal for the DSU service to return `0` when we reached them. For example, the high counts of `uint64_t` directed us to write a rule for **`struct`** `stats` in Figure 2.11, line 227.

Time required for profiling: Generating a full profile requires visiting every object in the heap, whereas for other services we can sometimes halt traversal early. Table 2.2 shows a summary of the results of profiling. The first column shows the program name and a footnote explaining the amount of state present at the time of traversal. The second column shows the total number of calls to `perfaction_ptr`, `perfaction_struct`, and `perfaction_prim`. The third column shows the amount of time it took to profile the heap using 4 threads, and the fourth column shows the semi-interquartile range (SIQR) for the measured times. (The SIQR, half the numerical distance between the upper and lower quartile points of our trial runs, is a measure of variability and shows that for this experiment there was only a moderate amount of variance in the trials.) The fifth column shows the number of unique types that were discovered in the traversal, and the final time is the sum in KB of all unique allocated

structures traversed. The time for traversing all of a Memcached heap with 4,000 key-value pairs is only slightly slower than serializing a Memcached heap with 4,000 key-value pairs. The time for traversing all of the Redis heap for 4,000 key-value pairs is significantly slower than serializing the Redis heap with the same number of entries because we did not employ any optimizations for the key-value items and traversed all fields individually. Snort is by far the largest heap to traverse with 10,299,744 calls to the perfaction_ functions and 138,417KB of allocated structures. While Snort has a high traversal time overall, the ratio of traversal time to calls to perfaction_* is similar for all three applications: 0.43 μs/call for Snort, 0.41μs/call for Redis, and 0.60μs/call for Memcached, which is higher because the smaller heap does not amortize the startup and teardown costs.

2.5.5 Heap assertion checking

Finally, we also used C-strider to implement a simple heap assertion checking system, inspired by Aftandilian and Guyer's GC assertions [6], which employ the Java garbage collector as a traversal mechanism. Heap assertions are tightly tied to particular data structures, and with C-strider we can write an assertion specific to each type. For example, we could assert that a doubly linked list is well-formed by adding a check during traversal like:

```
if (type == TYPE_linked_list_PTR)
    assert (&in→ next→ prev == &in);
```

This application has no service-specific code.

Time required for heap assertions: The amount of time required for heap assertions varies greatly on the size of the portion of the heap being traversed. For example, in Memcached, the primary hash table contains a series of doubly linked lists. Asserting the property above for these lists required essentially the same time as for serialization (Figure 2.12) as the traversal covers the same portion of the heap. For Redis, we created a set of assertions checking that timestamps were not in the future, file descriptors were valid (using `fcntl`), database item **enum** fields were valid, and that linked list pointers were as expected. We traversed the entire Redis heap, and the traversal times were similar to the update time shown in Figure 2.13.

For Snort, we implemented a checker that asserts that installed function pointers are only drawn from a whitelist, and have not been hijacked to perform malicious functionality [77]. Additionally, we asserted that structures in the custom memory pool had valid pointers, that all of the rule list nodes were set to the appropriate mode, and that the rule lists were correctly formed. In total, we asserted the correctness of 1,749 function pointers and 446 other heap objects in 279.98ms.

2.6 Related Work

There are several threads of related work.

Kitsune: The most closely related work is Kitsune, the DSU system for C that directly inspired C-strider. There are several major advances that C-strider makes over Kitsune. First, C-strider is much more general than Kitsune, as it can implement a variety of services beyond updating. Second, in addition to type annotations,

Kitsune uses a DSL (Domain-Specific Language), *xfgen*, to customize the traversal, rather than customization via perfaction functions in C-strider. The reason for this difference is that Kitsune does not have a run-time representation of types. We find C-strider's approach much simpler and adaptable: *xfgen* is both over-specific to DSU and hard to extend to new applications. Third, C-strider's use of run-time types improves performance, because arrays and generics can be implemented by simply making a new type (with mktyp_) and using the standard visit function. In contrast, Kitsune uses a very complex closure system to traverse arrays and generics, and the extra levels of indirection cause the slowdown we saw in Figure 2.13. Finally, C-strider supports parallel traversal, while Kitsune is single-threaded.

Type-directed programming and annotations: In this work, we used annotations to convey the actual type of heap objects to make type-safe traversal possible. These type annotations are inspired by Deputy [24, 121] and Cyclone [51]. Cyclone is a type-safe variant of C that also uses programmer-supplied annotations (in addition to an advanced type system, a flow analysis, and run-time checks). Deputy uses dependent type annotations, e.g., to specify the tag field for a union or to identify existing pointer bounds information.

Hinze and Löh [49] compare various approaches to datatype-generic programming and generate code based on type definitions. C-strider also generates code over all of the datatypes in the C program's heap, but the exact code generated differs depending which types the program defines.

Garbage collection: Conservative garbage collectors [20] also traverse the entire heap, treating everything that looks like a pointer as a pointer. In contrast, C-strider's heap traversals must be exact rather than conservative. For example, if we "conservatively" treat an **int** as a function pointer that happens to have the same value, we would serialize and deserialize it incorrectly (because the function pointer could be at a different address when deserialized in another process).

Applications: As mentioned in the introduction to this chapter, there are several systems that implement the particular traversals we explored, but in an ad hoc way.

Dynamic Software Updating. DSU systems must traverse the heap to transfer state from one version of a program to another. We already compared to Kitsune above. Many other DSU systems [65, 71] also implement state transformation. Similar to state transformation, hot-swapping object instances [16] allows objects to be switched to another implementation while the system is running, seamless to the user. C-strider could be extended to perform a similar function during traversal.

Serialization. In this work, we implemented serialization and deserialization for the heap. This could be extended to a general checkpoint-and-restart system [33, 62, 85] by keeping some additional information about registers, thread-local data, and the stack. One benefit of the C-strider approach, compared to full program checkpoint-and-restart, is that C-strider serialization and deserialization is under program control and can be used piecemeal on portions of the heap. For example, it could be used to serialize a particular data structure in lieu of coming up with a new file format for that data structure.

Heap Profiling. Heap profilers can be used to determine how much memory a program uses, locate memory leaks, and find functions that do large amounts of allocation. Many popular heap profilers focus on function allocation granularity [35, 41, 115] based on calls to malloc rather than providing type-level granularity like Mihalicza et. al [68] and C-strider.

Heap assertions. Several researchers have developed systems for checking heap assertions. GC assertions [6] piggyback on top of the Java garbage collector to check a wide range of heap properties. DEAL [81] implements a language for heap assertions that can express combinations of reachability conditions. QVM [14] also checks heap properties, but using *heap probes* to keep overhead low by controlling the frequency of the checking and by sampling. PHALANX [117] extends QVM by adding checks for reachability; PHALANX also parallelizes the queries. Dynamic shape analysis [52] summarizes the pointer relationships of data structures and reports errors when invariants are violated. These systems all run on top of a garbage collector and virtual machine. In contrast, C-strider's heap assertion checking works on C, which has no garbage collector or VM.

2.7 Conclusion

We have presented C-strider, which is, to our knowledge, the first general-purpose, type-aware heap traversal framework for C. C-strider analyzes a target program and generates traversal code for it, which can be run serially or in parallel. The traversal invokes programmer-supplied callbacks as it visits different lo-

cations, passing along appropriate type information. These callbacks implement, or customize, a service. We have experimented with several services, including (de)serialization, dynamic software updating, heap profiling, and heap assertion checking. Where needed, the programmer can augment the standard C types with additional information about array sizes and container types using special annotations. For other cases, like tagged unions, programmers can write arbitrary C code to customize the traversal itself. We found that writing services and customizing traversals with C-strider generally requires a small amount of code, and that performance is reasonable and scales appropriately with heap size.

Author Contributions: This chapter was published in Software: Practice and Experience [96]. I was lead author on that paper and performed all of the programming, development, and experiments, and led in the writing and overall design. My collaborators, Michael Hicks and Jeffrey S. Foster, contributed to the writing and to refining the ideas and the design of this work.

Chapter 3: KVolve: Evolving NoSQL Databases Without Downtime

NoSQL databases, such Redis [89], Cassandra [11], and MongoDB [2], are increasingly the go-to choice for storing persistent data, dominating traditional SQL-based database management systems [15, 42]. NoSQL databases are often organized as *key-value stores*, in that they provide a simple key-based lookup and update service (i.e., "no SQL"). While these databases do not provide a formal language for specifying a schema, applications attach meaning to the format of the keys and values stored in the database. Keys are typically structured strings, and values store objects represented according to various formats [67], e.g., as Protocol Buffers [39], Thrift [12], Avro [10], or JSON [50] objects.

Database schemas change frequently when applications must support new features and business needs; for example, multiple schema changes are applied every week to Google's AdWords database [79]. Applications that use NoSQL databases also evolve data formats over time, and may require modifying objects to add or delete fields, splitting objects so they are mapped to by multiple keys rather than a single key, renaming of keys or value fields, etc. When changes are not compatible with the old version of an application, a straightforward way to deploy them in the field would be to shut down the running applications; migrate each affected object

in the database from the old format to the new format; and then start the new versions of the applications.

Performing these changes *online* is more challenging. The parsers for Thrift, Protocol Buffers, and Avro provide some support for format changes, e.g., by skipping unknown fields or by attempting to translate data from the writer schema into the reader schema [57]. This leaves the task of updating each object in the database (e.g., by iterating over all of its keys [90]). For large amounts of data, this can create an unacceptably long pause. For example, Wikipedia was locked for editing during the upgrade to MediaWiki 1.5, and the schema was converted to the new version in about 22 hours [120]. Developers could avoid shutting down the application by making the new format backward-compatible with the old format, but this would impose a significant constraint on the future evolution of the application. It may also be possible to grant applications read-only access to the old database while the migration takes place, but applications that have even occasional writes will suffer.

One way to avoid downtime and database locking is to migrate data *lazily*. When the application accesses an object in the old format, the object is converted to the new format on-the-fly. Thus, the only update-time pause is due to updating the application—the far longer pause due to migrating the data is now amortized over the new version's execution, causing slower queries immediately after the update but no full stoppage. Disruption due to shutting down and restarting the application can be reduced using techniques like DSU or by switching between parallel versions on-line, as in parallel AppEngine [38].

Currently, the task of lazy data migration falls on the developer: applications

are written to expect data in both old and new formats and to migrate from to the new format when the old one is encountered [17, 83, 87, 109]. This approach adds work for the programmer and results in code that mixes application and format-maintenance logic. Since there is no guarantee that all data will ultimately be migrated, the migration code expands with each format change, becoming more confusing and harder to maintain.

This chapter presents KVolve[1], an extension to the popular Redis NoSQL database, as a solution to the problem of evolving a NoSQL database online. To simplify development, KVolve presents the logical view to applications that data is at the newest version of the format. Rather than convert all data at once, keys and values are converted as they are accessed in a way that requires almost no changes to applications—they simply indicate the data version they expect when they connect to the database, and they are permitted to proceed if their expected version and the logical view's version match. To avoid downtime, applications can use a DSU tool such as Kitsune in conjunction with KVolve to update the client-side code along with the server-side data. Commands submitted to Redis by a client that has just upgraded the application code trigger data conversions automatically. These happen in KVolve code that interposes between the client request and the normal request handling by Redis. To track its progress, KVolve attaches a version identifier to the value of each entry, converting only those keys/values that are out of date. Conversions must be written by the developer, and are specified as functions from the old key/value format to the new one. KVolve allows the conversion functions

[1]KVolve stands for *Key-Value store evolution*.

to perform additional calls to Redis, enabling the conversion function to access or modify other keys if necessary. However, KVolve does not support conversions in which a new key/value is a format of several old key/values; this is necessary to maintain the required logical view. Many applications we have considered easily satisfy this restriction.

Experiments with our implementation show that KVolve imposes essentially no overhead during normal operation. Using the standard Redis benchmark that repeatedly queries the database (the worst case as far as application performance is concerned), we observe the overhead to be in the noise. We also find that laziness significantly reduces the update-time disruption. We applied KVolve to upgrade redisfs [56], a full-scale file system that uses Redis as the back end. The upgrade in question (to version 0.7), required a significant data format change: keys representing file directories were renamed, and data containing the file contents were compressed. While KVolve took care of the changes to the database, we used Kitsune to dynamically update the redisfs driver. Doing so allowed us to seamlessly maintain the file system mount point and other in-memory state. The result was zero downtime compared to 11 seconds in the case where we migrated the keys eagerly. We also tested our approach on a social networks program called Amico [7], and showed that we could reduce the pause time for renaming all 792,711 keys from 100 seconds in the eager case to almost zero seconds with KVolve.

In summary, we make three contributions:

- We define a general approach to updating data formats in NoSQL databases

lazily. For large databases, this approach gives the appearance of an instant update, as the applications can start using the new data format without delay, and it amortizes the cost of migration over the normal execution. Additionally, this approach does not limit the client-side interactions (e.g., to read-only execution) and requires very few changes to applications. (Section 3.1.)

- Our implementation with the Redis NoSQL database requires minimal changes to Redis itself: it is implemented in a modular way and does not interfere with Redis command processing. (Section 3.2.)

- Experiments show that KVolve adds essentially no overhead during normal operation, with the corresponding benefit of very little performance degradation during the lazy update rather than a full stoppage, especially compared to the eager approach. We also show how KVolve can be integrated with Kitsune to perform end-to-end upgrades with no downtime (Section 3.3.)

In Section 3.4, we also present a Python client-side implementation of KVolve, Py-KVolve. Py-KVolve assists users in writing their updates by providing a DSL (Domain-Specific Language) and does not require modifications to Redis, but has several drawbacks, especially performance.

To the best of our knowledge (see Section 3.5), KVolve represents the first general-purpose solution to the problem of evolving a NoSQL database without downtime. We plan to make our code freely available.

3.1 Overview

This section presents an overview of KVolve, describing its various pieces and how they fit together using a simple example.

3.1.1 Background on Redis and NoSQL databases

NoSQL databases are distinguished from traditional *relational database management systems* (RDBMSs), in supporting a very simple, lightweight interface. Our focus is on a variant referred to as *key-value stores* which, as the name implies, focus on mapping keys to values. We chose to work with the key-value store Redis [89], one of the most widely used key-value databases [106]. The main operations in Redis, in particular, are: `GET` k, which returns the value v to which k maps in the database (or "none" if none is present); and `SET` k v, which adds (or overwrites) the mapping $k \rightarrow v$ in the database.

Redis supports variations of these operations (e.g., setting values only if no prior version exists, or defining mappings that will time out). It also supports additional data structures (the main data structures include strings, sets lists, hashes, and sorted sets) and their corresponding operations, such as appending an item to a list. KVolve supports 36 primary Redis commands and their variations, including support for all of the main Redis data structures. Although we have not yet implemented all commands for KVolve, we expect this to be straightforward, as discussed in Section 3.2.2.

Many applications store string values that adhere to formats such as JSON [50],

Avro [10], or Protocol Buffers ("Proto Bufs") [39]. In the example in this section, we focus on Redis databases that store JSON objects. JSON defines four primitive types: numbers, strings, booleans, and null. It also defines two container types: arrays, which are an ordered list of values of the same JSON type; and objects, which are an unordered collection of values of any JSON type, with field labels. Our basic approach should apply to other formats as well, and it would arguably be easier to do so, as Avro and Proto Bufs also define a notion of schema that could be analyzed to understand the effect of a change.

A common programming practice for key-value stores is the use of *namespaces*, which conceptually divide up the kinds of objects stored in the database. Redis does not provide native support for namespaces, but rather advises their use by convention: Keys should have the format $n{:}k$ where n identifies the namespace, and k identifies the proper key name. The assumption is that objects in namespace n will all have the same type. Of course, n might be further partitioned into sub-namespaces, as required. We also support having a global namespace, treating all of the keys in the database as the same type.

3.1.2 Example data format change

As an example, consider an online store (borrowed from an example by Sadalage and Fowler [87]). Such an application may store purchase orders in its database. Keys have the format $\mathsf{order}{:}n$ where order is the namespace, and n is a unique number, representing an invoice number. Such keys map to values that describe a

```
{  "_id": "4BD8AE97C47016442AF4A580",
   "customerid": 99999,
   "name": "Foo Sushi Inc",
   "since": "12/12/2012",
   "order": {
      "orderid": "UXWE-122012",
      "orderdate": "12/12/2001",
      "orderItems": [
         { "product": "Fortune Cookies",
           "price": 19.99 }
      ]
}  }
```

(a) JSON object, version 0

```
      "orderItems": [
         { "product": "Fortune Cookies",
           "fullPrice": 19.99,
           "discountedPrice": 16.99 }
      ]
```

(b) Update to orderItems field, for version 1

Figure 3.1: Evolving a purchase order object

purchase, formatted as JSON as shown in Figure 3.1a.

Suppose we wish to upgrade the application to support differentiated pricing, which necessitates changing the data format as follows: rename the field price to fullprice , and insert a new field named discountedPrice that is a markdown from the original price. The updated orderItems array (the last element of the JSON object) for the example, that adheres to the new format, is shown in Figure 3.1b.

To support migrating from the old to the new version of the application, Sadalage and Fowler suggest that the programmer can modify the new version's code to essentially handle both formats, and migrate data from the old format to the new one when it is encountered.

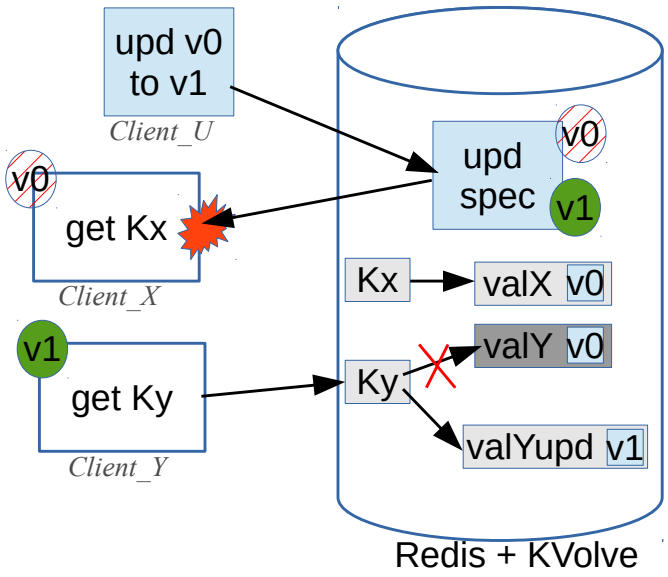

Figure 3.2: KVolve architecture.

This approach is efficacious but creates extra effort for the application programmer, as they have to write code that deals with the differing versions, and this code must reside in the application indefinitely. Sadalage and Fowler also do not explicitly consider the complications of concurrent clients, which could read/upgrade the data at the same time and produce inconsistencies. KVolve avoids this problem by ensuring that the key will be updated exactly once and that subsequent client requests will only read up-to-date data.

3.1.3 Database upgrades using KVolve

KVolve is a minimal extension to standard Redis with some update-enabling elements. (The implementation inner workings are described in detail in Section 3.2.2.) KVolve's architecture is depicted in Figure 3.2, with its update elements depicted in blue. In particular, the figure shows (at the top) *Client_U* installing an

update specification in the database, signaling that the database's contents need to be upgraded. And it shows two clients that are interacting with Redis, *Client_X* and *Client_Y*, trying to retrieve key *Kx* and key *Ky* respectively. The connection order is as follows: *Client_X* connects first at *v0* and runs for a while. After the update is installed by *Client_U*, *Client_Y* connects at *v1*.

An update specification indicates the expected, current version identifier (in the figure, it is *v0*); the new version identifier (here, *v1*); and code that can be used to migrate data from the current version to the new version (not shown). This code is written in C with the help of some template functions, described in detail in Section 3.1.4.

When a client connects, it declares what data version(s) it expects for each namespace it will use. KVolve uses this information to ensure that all clients are connected at only the current version of the namespace's schema, and does not allow clients to connect that request outdated versions. When a version is declared for a specific namespace, KVolve automatically adds the version number to values so that KVolve may track what data it must lazily migrate when an update is requested. The version information also allows KVolve to help orchestrate a switchover to the newest application version by notifying out-of-date clients when/if they fall behind the logical version of the data they are using.

The bottom of the figure shows *Client_Y* interacting with the database after the update and illustrates a lazy update taking place. Each requested value is checked by KVolve to see if an upgrade is necessary. Now, just after an update, there will be no keys at the newest version—all of the data is still at the last (or

even older) versions. As such, KVolve performs an update on the value *valY* before having Redis return the value to the client at *v1* with the value *valYupd*. The newly updated version information and value is also stored in Redis, and the old version of the key is freed.

In the figure, we see that *Client_X* is still at *v0*. When the update is submitted, KVolve closes the connection to all outdated clients, and they will not be permitted to reconnect with the old version of the software. If we consider our purchase order example, we can see that the new version of the data will not work for older clients: They will be expecting JSON objects to contain field price but the data will (eventually) be stored under fields discountedPrice and fullPrice , instead. As such, the old client cannot safely access the data any longer, and must be terminated.

When data is updated in a backward-incompatible manner, clients designed to use that data will have been re-coded to use the new format. As such, when the old version is notified, it can start a new-version replacement that connects to the database. (To continue, *Client_X* must upgrade to *v1*.) The new version then migrates the data as it accesses it, as described above. This approach simplifies application development as each application version can assume a particular data version, and KVolve ensures data is migrated as it is needed. In fact, the only changes required to an application to support lazy data updates are to (a) change the call to connect to Redis to also declare the expected version of the data, and (b) to gracefully terminate and upgrade to the new version when the connection is closed. Graceful termination involves saving any final state locally if necessary, as database access is no longer allowed. These changes amount to only a few lines of

code for the entire application.

3.1.4 Describing data updates

The programmer must write *update functions* that will convert the old version of a key and/or value to the new version. These functions will be applied on demand by KVolve, but we want to present the logical view to the application that the data was converted eagerly. To ensure this illusion, the new key or value should be a function of *only* the old key or value, and not any other data in the database, since there is no guarantee about the order this data will be changed.

An example update function is given in Figure 3.3. The old key (a string) and value (binary data) are passed in by reference, and the function will update them to the new versions via these references. In this case, the body of the function uses the Jansson library [59] to implement the change to the purchase order example from Figure 3.1 described in Section 3.1.2; the last two lines update the value (the key is not changed). This function is installed by an initialization function called when the update is loaded in KVolve:

```
kvolve_upd_spec("order","order", 0, 1, 1, test_fun_updval );
```

This indicates that the `order` namespace doesn't change, from version 0 to version 1, while the `test_fun_updval` should be called for each key in the namespace `order`.

Namespaces can be changed without requiring an update function. For example, in the Amico program described later in Section 3.3.3, the keys are renamed from the namespace prefix of `amico:followers` to the namespace prefix of

```
1  void test_fun_updval(char ** key, void ** value,
2                                    size_t * val_len){
3    json_t *root, *arr, *ele, *price;
4    int i; double pval;   json_error_t error;
5    root = json_loads((char*)*value, 0, &error);
6    arr = json_object_get(
7            json_object_get(root, "order"), "orderItems");
8    for(i = 0; i < json_array_size(arr); i++){
9       ele = json_array_get(arr, i);
10      price = json_object_get(ele, "price");
11      json_object_set(ele, "discountedPrice",
12                 json_real( json_real_value(price)-3.0));
13      json_object_set(ele, "fullPrice", price);
14      json_object_del(ele, "price");
15   }
16   *value = json_dumps(root,0);// Set the updated value
17   *val_len = strlen(*value);// Set the updated val length
18 }
```

Figure 3.3: Example update function for JSON

amico: followers : **default**. To describe this update, the update-writer should specify:

```
kvolve_upd_spec("amico: followers", "amico: followers : default", 1, 2, 0);
```

where the version numbers are 1 and 2, and the 0 indicates that there are no functions to manipulate the value.

3.2 KVolve Implementation

We designed KVolve to be as modular as possible and to avoid disrupting Redis's normal command-processing control flow. This section describes our implementation of KVolve as an extension to Redis.

3.2.1 Design goals

When designing KVolve, we wanted it to be general-purpose, efficient, and easy to maintain. We initially built an application-side wrapper written in Python, which has the virtue that no changes to Redis are required at all [95], which will be presented in Section 3.4 and Appendix A. But this wrapper is limited to Python Redis clients, and imposes a fair bit of overhead (5–8% on normal operations). We then experimented with a network proxy, written in C, that intercepts Redis-application communications. This approach also requires no changes to Redis, and supports applications written in any language. But we still found it to be slower than we'd like (about 3% overhead), and tricky to write, especially when trying to ensure transaction atomicity.

In the end, we designed KVolve as a separate library that is compiled into Redis itself. Our changes add a `vers` version field to the Redis `robj` data structure, which represents data values stored in the database. They also hook the command processing logic to support lazy migration. In total, 7 lines of Redis (v2.8.17) needed to be changed: 2 lines of code to add and initialize `vers` to a default value, 2 lines of code to add a function call into the KVolve library right before Redis processes commands, and 3 lines of code to allow the `vers` data to be serialized and deserialized to and from disk using the normal Redis serialization commands. This approach has proved to be the most efficient, most general, and easiest to get right, thanks to Redis's single-threaded design. That is, Redis serializes the commands it receives from clients, processing one in its entirety before moving to the next. As such, added

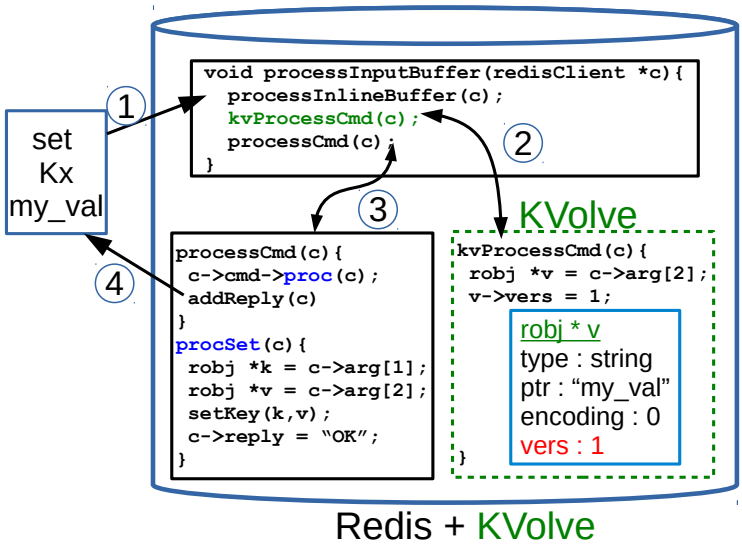

Figure 3.4: Control flow for Redis and KVolve

KVolve processing can be easily made atomic with respect to normal processing of a command. Finally, we believe our current implementation should be easy to maintain: The data structure we modified to add a version field has not changed significantly for several years, in particular, between versions 2.0.0 (2010) and 3.0.2 (2015).

KVolve currently supports 36 Redis commands and all of the main Redis data structures (string, set, list, hash, sorted set). The commands we implemented for our prototype were chosen after examining roughly 30 programs on GitHub that used Redis to see which Redis commands were most commonly used.

3.2.2 KVolve implementation overview

KVolve works by preprocessing commands coming in from the client before passing them along to Redis, as depicted in Figure 3.4. In Step 1, the client issues

the command set Kx my_val. In Step 2 kvProcessCmd, KVolve's hook, is called to preprocess the command (the dashed green box is the KVolve library). Once the KVolve preprocessing is complete (which might involve changes to data's contents and version field), control returns to normal Redis. In Step 3, Redis's processCmd function calls the function pointer shown in blue (which depends on the choice of command—here it is procSet because the client requested set), and this adds the affected object to the database, including any changes to the version field set during KVolve's processing. Finally, in Step 4 Redis responds to the client's request, acknowledging to the client that it successfully executed the set command.

All of KVolve's operations are informed by a data structure it maintains called the *version hash table*. This hash table is constructed from the update specifications submitted to KVolve. (It is not stored in the persistent DB, so if the database goes down, then the update specifications will have to be resubmitted when it comes back up, to reproduce it.) The hash table maps a namespace[2] to a record that notes the namespace's current and previous versions, as well as update functions that move data from one version to the next. It also tracks a list of identifiers for the client connections that have declared interest in the current version of this namespace. Because a logical namespace can change its representation (e.g., from foo: to foo:bar:), the record also tracks prior and subsequent namespace representations (which can themselves be looked up in the version hash to get the associated information). The version hash table maintains a history of all updates over time, as keys that are

[2]As described in Section 3.1.1, namespaces are commonly used in key-value stores and conceptually divide up the kinds of objects in the database.

rarely accessed may require updates far into the future. (Alternatively, we could force-update stale keys using a background thread and purge old updates when all key-value pairs have applied an update. However, we find that stored update information takes up very little space relative to the rest of the database.)

When KVolve processes a command, it parses out the entire namespace prefix from the command's key (based on the rightmost colon in the key name) and looks it up in the version hash table. If a match is found, the lookup returns the relevant record. If no match is found and there are other colons in the key substring, KVolve will recursively search for the next longest matching prefix, starting at the next rightmost colon. If still no match is found (as in the case where the database administrator does not wish to track any versions or implement updates), KVolve returns control to Redis without further action.

After the namespace lookup has completed, KVolve passes a pointer to the record containing the namespace information along with the parsed client request to the appropriate KVolve preprocessor function, such as `kvolve_get` for the `get` command. We describe these next, by type.

3.2.3 String types

Redis supports five main data structures. Strings are the only type of data structures that are not containers, in that the value consists of a structure storing a single string. For that reason, we discuss string types separately.

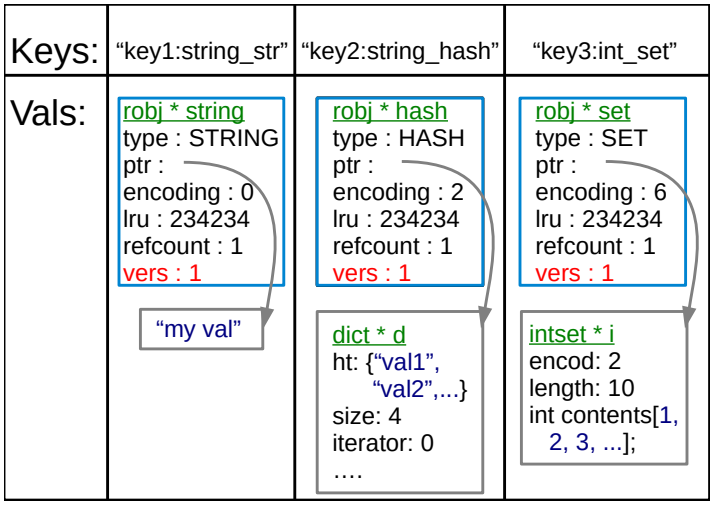

Figure 3.5: Storing different data types

Getting strings: If the client request involves getting a string data structure, KVolve must first prelookup the existing robj value structure in the database to get the version information. An example of the key-value pair for string types is shown for key1: string_str in the first column of Figure 3.5. This action retrieves a pointer to the actual object structure that is stored in the database, so any modifications that KVolve makes to this object will be automatically stored in Redis. Note that this requires an additional database lookup by KVolve, on top of the one that Redis will do later when it does its own lookup to handle the client request. However, this is an O(1) operation and does not incur excessive overhead relative to the other operations that KVolve must already perform, and does not interfere with normal Redis operations.

If the version field of the robj (in this example the version is 1 shown in red for key1: string_str in Figure 3.5) is current for the namespace of the key, or if the key is not present under the current or any former namespace (and therefore no robj exists

for the key), then KVolve returns control to Redis and does no further processing. If the version is not current, either in the current namespace or a former namespace, KVolve will update the key and value as necessary. All of the necessary information to perform the update (the update functions themselves, and the meta-data about which namespaces and versions the updates apply to) is stored in the version hash table, and KVolve uses that information to apply the update as follows:

- In the case of a namespace change, KVolve uses the update information from the version hash table to perform the key rename, leaving the value untouched.
- In the case of a value change, KVolve calls any applicable user-supplied functions and applies them to the value, starting from the oldest needed update and working forward to the current version. After all of the update functions have been applied, KVolve stores the updated value in the `robj` (which is a pointer to the actual structure stored in the database) and updates the version string to match the current version by setting the field in the `robj`.

If both actions (namespace and value change) are necessary, KVolve will perform both. KVolve then returns control flow to Redis, and when Redis performs its own `get`, it will retrieve and return the newly updated key to the client.

Setting strings: If the client request involves setting a string data structure, KVolve first checks to see if the request has any flags that would prevent the value from getting set. These flags, `XX` or `NX`, respectively, specify to only set the key if it already exists, or only set the key if it does not already exist. If necessary,

KVolve does a lookup in the database to determine if the key exists, indicating if the value will be set for the requested key. If the value will not be set due to the flags, KVolve does nothing and returns control to Redis. In this set command, and also for any such command where Redis will be adding the robj to the database, Redis deletes the old robj and replaces it with the new one from the parsed client's request. Therefore, all that KVolve must do is set the most current version string in the robj for the namespace of the key. (Remember that there is no need to attempt to update the value in the key, because the client's provided value is guaranteed to be at the up-to-date version.)

If this set occurs after a namespace change, KVolve must delete the old value for the key to ensure that deprecated key versions are not unnecessarily bloating Redis. For example, in a change to redisfs (presented in Section 3.3.2), an old namespace key was named skx:/ but after an update, the new namespace postfixes DIR such that the key is now named skx:DIR:/. If the user were to set the key skx:DIR:/root before getting (and updating) it, this would leave the old key skx:/root still in the database. Therefore, KVolve must check to see if the existing version under the old namespace exists, and if it does, delete it. It does this by first checking if the namespace had any previous namespace changes. If not, it does nothing. If so, it checks and deletes the old key if necessary. At this point, KVolve returns control to Redis, and Redis adds the robj structure to the database, which also contains the updated version string to be retrieved later if necessary.

3.2.4 Sets, hashes, lists, and sorted sets

The other Redis data structures are containers of sub-values. Although the exact substructure is different for each type, the basic implementation is the same, so we describe them together. The base of Redis containers are all robj structures, and they store the actual data in other ways to maximize efficiency of both space and speed. Figure 3.5 shows an example in columns two and three of value robjs that contain a hash of strings and a set of integers respectively. Because not all sub-values (individual set members, individual list items, etc) are stored directly in a robj structure, it is not possible to store the version information for the individual sub-values without further modifying Redis. Therefore KVolve must store the version information in the robj of the set/list/etc container robj itself, and KVolve must keep *all* members of the sub-values at the same version.

Getting elements from container types: The process for doing a get on one of the container elements is exactly the same as for the string type described in Section 3.2.3, except that if necessary, the update is performed for all of the sub-elements. KVolve uses a Redis-provided iterator to access and update the sub-elements as necessary. If a container type has a namespace change, the rename process is exactly the same as it is for string types, as the keys are the same regardless of value type.

Setting container types: The set process for container types is essentially equivalent to the get process described in the previous paragraph. For example, suppose

key foo maps to the set of strings {'s1', 's2', 's3' } at version 1. Then say an update to version 2 replaces the letter s with the letter t for each element in the set. If a command, post-update, wishes to add string 't4' to foo's set, then the version in the set's robj must be updated, along with the other values in the same set. This is done by performing a get to update the sub-elements, changing the value for foo to be {'t1', 't2', 't3' }. KVolve will also update the version in the robj to be 2, and then return control to Redis, which will add 't4' to the set. This process is similar for lists, sorted sets, and hashes: when any sub-element is accessed, KVolve iterates over and updates all the other sub-elements.

3.2.5 Installing an update

We conclude with a few more details about how updates are installed in KVolve.

The programmer compiles the update specifications into a shared object file that she can direct KVolve to load. In particular, the shared object file will contain the update functions (e.g., Figure 3.3), and a function kvolve_declare_update, which consists of a sequence of calls to kvolve_upd_spec that define the per-namespace changes defined by the update (again, see Section 3.1.4 for examples).

A client (or system admin) can load an update into KVolve by issuing a (repurposed Redis) command that includes the path to the location of the shared object. KVolve will load the shared object from the indicated location, which results in the kvolve_declare_update being run (it is declared as a "constructor" for the shared ob-

ject, so it will be automatically called immediately after the shared object is loaded). For each call to kvolve_upd_spec (...) that is made, KVolve verifies that the specified old namespace exists at the specified old version, and that the specified new version does not exist (which could happen if a different client already loaded the update). In the case of a namespace change, KVolve verifies that the specified new namespace does not exist. If all these checks pass, the code is registered and loaded into the version hash table, so that it will be applied to any keys matching the namespace and version.

At this point, KVolve will close the connection to all clients using the old version of the namespace(s). (Clients not using the updated namespace(s) will not be affected.) Redis provides a client id for each connection, and as mentioned these ids are stored in the version hash table as each client connects. Note that this does not automatically kill the client's application, it just closes the connection and the client will not be allowed to reconnect until it is upgraded to the new version.

3.3 Experimental Results

This section considers the performance impact of KVolve, during normal operation and during an update. Our experimental results are summarized as follows:

- Using the standard benchmark that is included with Redis, we found that KVolve adds essentially no overhead during normal operation, and we determined that storing the version and update information in Redis adds only about a 15% overhead in space.

- We updated the redisfs file system which included renaming some keys and compressing some data stored in keys, and found the operating overhead to be in noise, and the pause time to be close to zero as opposed to 12 seconds for an offline migration of the data.

- We updated the Amico social network system and found no added overhead, with a pause time of close to zero as opposed to 87 seconds for an offline migration of the data.

All experiments were performed on a computer with 24 processors (Intel(R) Xeon(R) CPU E5-2430 0 @ 2.20GHz) and 32 GB RAM with GCC 4.4.7 on Red Hat Enterprise Linux Server release 6.5. All tests report the median of 11 trials, and communication was via localhost with ~.03 ms latency.

3.3.1 Steady state overhead

First we report the steady state overhead for KVolve reported by Redis's included benchmark, *Redis-bench* . Redis-bench acts as a client that repeatedly issues commands to Redis. The default settings for Redis-bench are with 50 clients, with 10,000 repetitions of a single operation at a time (only 1 request per round trip), and with a single key (getting or setting a single key multiple times). However, Redis-bench allows many different configurations. For a longer benchmark, we increased the number of operations to 5 million operations and for a more realistic benchmark we performed these operations over 1 million keys, leaving the rest of the default settings alone. We ran this experiment over localhost which had a latency

	Redis		No NS, KVolve			With NS, KVolve			&Prev NS, KVolve		
	time	siqr	time	siqr	OH	time	siqr	OH	time	siqr	OH
String Get	58.83	(0.25)	58.08	(0.11)	**-0.77%**	58.46	(0.26)	**-0.12%**	59.51	(0.85)	**1.67%**
Set Pop	58.20	(0.10)	58.25	(0.20)	**0.09%**	57.72	(0.31)	**-0.82%**	58.09	(0.21)	**-0.19%**
List Pop	58.47	(0.41)	58.93	(0.68)	**0.79%**	58.71	(0.15)	**0.41%**	59.34	(0.47)	**1.49%**
String Set	63.52	(1.04)	63.66	(0.70)	**0.22%**	65.39	(1.49)	**2.49%**	64.82	(0.35)	**2.05%**
Set Add	61.52	(0.77)	61.11	(0.24)	**-0.67%**	61.25	(0.97)	**-0.44%**	62.06	(0.71)	**0.88%**
List Push	59.49	(0.66)	59.55	(0.56)	**0.10%**	60.02	(0.74)	**0.89%**	60.87	(0.94)	**2.32%**

Table 3.1: Redis-bench with single instructions for Redis vs KVolve (times in seconds, median of 11 trials)

	Redis		No NS, KVolve			With NS, KVolve			&Prev NS, KVolve		
	time	siqr	time	siqr	OH	time	siqr	OH	time	siqr	OH
String Get	9.73	(0.30)	9.75	(0.27)	**0.21%**	9.96	(0.25)	**2.36%**	9.93	(0.20)	**2.06%**
Set Pop	9.88	(0.41)	9.52	(0.35)	**-3.64%**	10.04	(0.51)	**1.62%**	9.87	(0.44)	**-0.10%**
List Pop	9.60	(0.32)	9.71	(0.25)	**1.15%**	9.55	(0.43)	**-0.52%**	9.63	(0.27)	**0.31%**
String Set	13.77	(0.26)	14.56	(0.18)	**5.74%**	14.56	(0.32)	**5.74%**	14.48	(0.33)	**5.16%**
Set Add	13.97	(0.57)	14.09	(0.32)	**0.86%**	14.35	(0.71)	**2.72%**	14.67	(0.30)	**5.01%**
List Push	14.22	(0.39)	14.38	(0.25)	**1.13%**	14.40	(0.41)	**1.27%**	14.48	(0.36)	**1.83%**

Table 3.2: Redis-bench with 10 pipelined instructions for Redis vs KVolve (times in seconds, median of 11 trials)

Program	**Max RSS**
Redis, empty	7.7MB
Redis, 1M 10-byte values	112.1MB
KVolve, empty	7.7MB
KVolve, 5 namespaces, 1M 10-byte values	128.6MB

Table 3.3: Max resident set size (RSS)

of ~.03ms. We chose three types of `get` operations (string gets, set pops, and list pops) and three types of `set` operations (string sets, set adds, and list pushes), as these were part of the default benchmark operations test suite.

Table 3.1 shows the steady state overhead of this experiment. We show unmodified Redis in column 2 for comparison and broke the overhead into separate categories: KVolve with no namespace declared (causing KVolve to return immediately for each key) in column 3, KVolve with a single namespace declared (causing a hash lookup and a version check for each key) in column 4, and KVolve with a previous namespace declared but no previous keys at the old namespace (causing a hash lookup, a version check, and a string concatenation to look for a non-existent previous key) in column 5. Each sub-column of Table 3.1 shows the total time for the test, the SIQR to show the variance, and the overhead as a comparison against unmodified Redis. We ran this benchmark many times with various configurations (multiple namespaces, less or fewer keys, less or fewer clients, etc) and found that the overhead varied generally around ±3%, with no consistent pattern between any of the tests, even repeated tests with the exact same setup. The numbers presented

in the table show some negative and some positive overhead, reflecting this variation. Notice that the SIQR numbers show that the variance is relatively high, as high as 1.49s for setting strings with KVolve and a namespace, shown in the sixth row of the fourth column.

Table 3.2 shows a modification of the original overhead experiment, using a pipeline to feed 10 instructions into each round trip to Redis-bench over localhost. This reduced the I/O overhead, putting more emphasis on KVolve operations. We found that these numbers showed a bit more overhead, and allowed us to bound the overhead at 5.74% for 10 subsequent pipelined instructions. This test demonstrated that although there is some overhead added by KVolve, for the non-pipelined version and most commonly-used scenario (Table 3.1), the overhead is mostly buried in I/O and very low overall. (In our test programs, described next, Amico pipelined at most 3 instructions per round trip, and redisfs did not use pipelineing.)

In addition to time overhead, KVolve incurs some additional memory overhead due to tracking the version information. Table 3.3 shows the maximum resident set size as reported by *ps*. Empty, Redis and KVolve take up about the same amount of size in memory. With 1 million keys each mapping to 10-byte values and with 5 namespaces declared, KVolve takes up about 16.5MB (~15%) more memory than unmodified Redis. This includes the extra version field (4 bytes) on each value structure, the amount of space it takes to store the version lookup information and hash table, and any extra padding that may be automatically added to the additional structures.

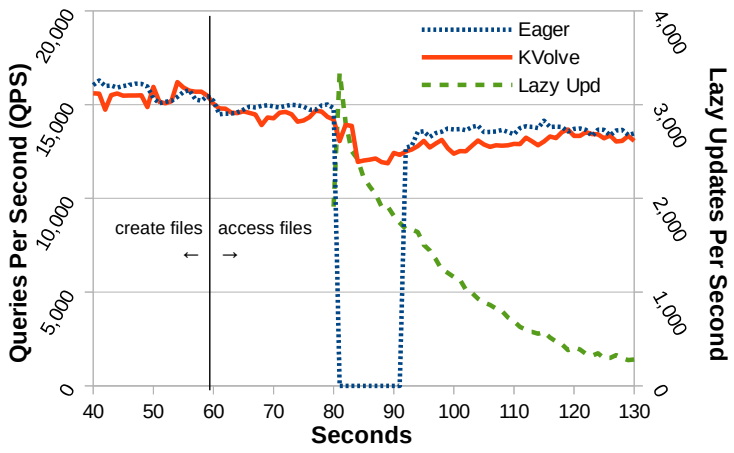

Figure 3.6: Lazy vs. eager updates for RedisFS

3.3.2 Redisfs

Redisfs [56] uses Redis as the backend to the FUSE [112] file system. The inode information, directory information, and all file system data are stored in Redis. On startup, FUSE mounts a directory with Redis as the backend, and a user can perform all of the normal operations of a file system, with the data silently being stored in Redis. Redisfs has 8 releases, ~2.2K lines of C code each. In redisfs.5, released March 4th, 2011, file data is stored in a Redis as a binary string with no compression, and the directory keys have the format `skx:/path/todir`. In redisfs.7, released March 11th, 2011, file data is compressed using `zlib`, and directory keys have the format `skx:DIR:/path/todir`. (Note that redisfs.6 contained an error and was retracted, so we use versions .5 and .7.) This change makes it impossible to view the directories or any of the files using redisfs.7 for any files created using redisfs.5.

In all versions, the inode data is stored across about 12 Redis keys with meta information such as modification time, creation time, file size, and also the data

itself. All file system information is represented in redisfs with four namespaces: the skx:/ namespace to represent the directories (which is updated to skx:DIR/ in redisfs.7), the skx:NODE namespace to represent the inodes (some of which is updated to add compression in redisfs.7), and also skx:PATH for paths to directories and skx:GLOBAL to track internal structure, neither of which are updated. To make redisfs compatible with KVolve, we added the same 6 lines of code in both versions which consisted of an additional call to Redis on start-up to declare that we would be using those 4 namespaces at either version .5 or .7, along with a few additional lines of error handling to make sure that the namespaces were properly set. No additional changes were made to redisfs for KVolve.

We performed an update from redisfs.5 to redisfs.7, both by migrating the keys offline (referred to as the Eager version), and with KVolve to automatically rename the directory keys as they are accessed and to add compression to the files as they are accessed. In addition to updating redisfs with KVolve, we also used Kitsune, whole-program update software for C, to allow us to also dynamically update redisfs along with the data so that the users experience no downtime; the switchover from .5 to .7 is completely seamless. Normally, killing redisfs.5 and restarting at redisfs.7 also causes the mount point to be unmounted then remounted (causing the user to have to switch back into the mounted directory after remount), but with Kitsune, the mount point is not disrupted during the switchover. We used the file system benchmark PostMark [54] to generate a workload for redisfs, creating an initial 10,000 files ranging from 4-1024 bytes in 250 subdirectories plus the root directory, for a total of 251 directories. We ran PostMark outside the root directory mount

point, accessing the files via full path name to avoid having to change directories due to the restart for the Eager (non-KVolve/Kitsune) version.

Figure 3.6 shows the results of the redisfs experiment. After about 60 seconds, PostMark switched from creating the new files to reading from or appending to existing files. Both KVolve and the Eager version had a very similar average Queries Per Seconds (QPS), as shown by the solid and finely dashed lines which correspond to the left y-axis. At 80 seconds, we killed redisfs.5. For KVolve, we used Kitsune to dynamically update to redisfs.7 without pause, maintaining the mount point so that the benchmark never lost access to the files or the directory structure, and KVolve continued to process queries throughout the update. For the Eager version, we halted all traffic to Redis and migrated the data, performing the renames and compression as necessary. In this update, not all of the keys needed to be updated, only the 251 directory keys that needed to be renamed and the 10,000 data keys that needed to be compressed. However, the database contained 123,002 total keys, and the to-be-updated keys were searched for in the database, adding to the pause time. This offline update process took about 12 seconds, as shown in Table 3.4.

In addition to showing the QPS lines, the green widely-dashed line in Figure 3.6 shows the number of lazy updates per second for KVolve, corresponding to the right y-axis. Immediately after the update, this number burst to about 3,000 keys per second, and quickly trailed off as keys were lazily updated. KVolve renamed the 251 directory keys, updated the version tag on all 112,752 keys in the skx:INODE namespace, and added compression to the 10,000 keys in that namespace that contained file data.

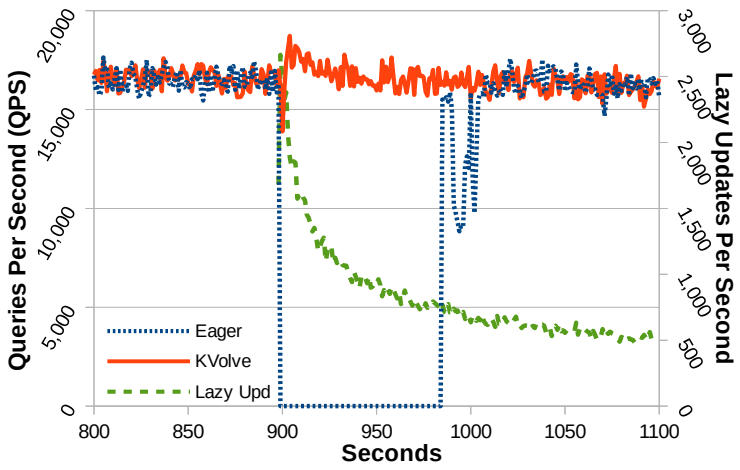

Figure 3.7: Lazy vs. eager updates for Amico

Overall the impact on the update experienced by redisfs was minimal, as the QPS dipped only slightly right after the update before it quickly returned to full speed around the 120 second mark of the experiment. After the update, the overall QPS was slower for both KVolve and redisfs because the files must be compressed and decompressed as they were accessed.

3.3.3 Amico

Amico [7] maps relationships in the style of a social network, defining a set of users and the relationships between them. Amico provides an API that allows queries over a data set of users: a user may be following or be followed by any number of other users. Amico is backed by Redis, has 10 versions created between 2012-2013, and is written in ~200 lines of Ruby code. Amico version 1.2.0, released Feb 22, 2012, stores these relationships in 5 different types of Redis keys with the following prefixes: amico: followers , amico: following , amico:blocked, amico: reciprocated , and amico:pending. In version 2.0.0, released Feb 28, 2012 (the next consecutive

version after 1.2.0), the developers added the concept of a "scope" so that there could be different graphs stored in Redis with prefixes to keep them separate, such as "school" network and a "home" network. The default name for the scope is "default", such that all of the keys are prefixed with amico: followers :**default** for example. This change makes databases created with Amico 1.2.0 incompatible with Amico 2.0.0. To make Amico work with KVolve, we only changed the same 4 lines of code in each version to declare the namespaces right after Amico connects to Redis.

For this experiment, we used the LiveJournal data set from the SNAP [61] library. The LiveJournal data set has 4,847,571 nodes and 68,993,773 directed edges defined by ordered node id numbers *A follows B* such as 186032 2345471, which we shuffled into two separate files for reading in a random order. To create a workload, we started two programs with calls to the Amico 1.2.0 API: one program to read from the first random file and add nodes to the Amico network, and one program to read from the second random file and perform queries over nodes in the network such as querying if USER A followed USER B or querying the number of followers of USER A. After letting the programs run for 900 seconds (15 minutes), the Redis database was filed with 792,711 keys containing nodes and edge data.

At the 900 second mark, as shown in Figure 3.7, we stopped both of the Amico 1.2.0 programs. For the Eager case (finely dashed line), we then updated all 792,711 keys, renaming them to have the **default** scope prefix in all of the key names. This migration took about 87 seconds as shown in Table 3.4. In addition to the pause, the Eager case shows a continued disruption until around the 1,000 second mark. After the migration was complete, we started the same writer/reader programs, this time

	Pause (s)	**Update Events**
Amico	87s	792,711 : rename
redisfs	12s	10,000 : compress, 251 : rename (123,002 total keys in database)

Table 3.4: Offline update pause times

using the Amico 2.0.0 API. For the KVolve case (solid line), we immediately started the two Amico 2.0.0 programs after the update so that the keys could be lazily migrated. Right at the update point, there is a ~2K drop in the QPS, before a brief spike and a return to the original rate. The widely-dashed green line corresponds to the right y-axis and shows the number of lazy updates that take place each second. Because this is a very large data set, many of the keys are not accessed immediately, taking full advantage of laziness. Although the lazy updates continue at a rate of about 500 per second at the 1,100 second mark, this does not significantly impact overall queries per second, as shown by the solid line maintaining a similar QPS before and after the update.

3.4 A Python client-side version of KVolve: Py-KVolve

This section describes the basics of Py-KVolve, a Python client-side implementation of KVolve. In contrast to KVolve, Py-KVolve exists as a wrapper sitting at a Python client, intercepting Redis API calls, rather than at Redis, intercepting Redis commands. One of the benefits of Py-KVolve is that it presents a DSL to assist with

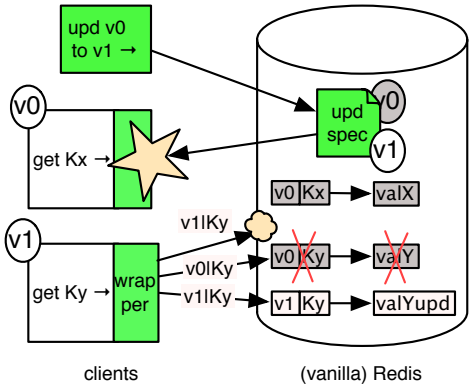

Figure 3.8: Py-KVolve architecture.

writing the update functions. Py-KVolve also has the benefit of not needing any modifications to Redis. However, Py-KVolve has some major drawbacks, including the large amount of overhead (4.95%-8.6% overhead in general use) and the limitation of only working with Python Redis clients. This section presents only the basics of Py-KVolve. Additional details, including the DSL specifics, are presented in full in Appendix A.

3.4.1 DB upgrades using Py-KVolve

The basic components of Py-KVolve are similar to KVolve, with some large implementation differences. Py-KVolve's architecture is depicted in Figure 3.8. Py-KVolve's client-side wrapper, shown on the clients in green, interposes on the basic Redis commands, i.e., *gets* and *sets*. Unlike KVolve, Py-KVolve's wrapper is an extension of Redis-Py [66], the Python client suggested by the Redis website. And unlike KVolve's update function, the update code for Py-KVolve is written in our DSL, illustrated in the next section and described in detail in Section A.1.1.

In contrast with KVolve, all of the update functionality occurs entirely on

the *client-side*. The wrapper used by clients provides roughly the same API as a standard Redis client, but performs three extra functions. The wrapper adds version information to *keys*, rather than values, as shown in Figure 3.8. (This is because Py-KVolve does not have access to the underlying database structures, and keys always consist of only a simple string, whereas the values may vary greatly.) The wrapper is also responsible for the lazy migration of old data it encounters. Finally, the wrapper orchestrates a switchover to the newest application version when/if it falls behind the logical version of the data it is using.

Like KVolve, the only changes required to an application to support lazy data updates are to (a) change the call to connect to Redis to also declare the expected version of the data, and (b) to gracefully terminate when signaled to do so. These changes amount to only a few lines of code for the entire application.

Updating data using Py-KVolve's DSL: To make it easier for programmers to migrate data that has changed format, we developed a DSL for specifying the required changes. Our language supports changes to key names, as well as modifications to the contents of JSON objects; for the latter we support adding, deleting, renaming, and updating fields.

For the purchase order schema change example from Figure 3.1, the corresponding update is expressed in Figure 3.9a. The **for** statement indicates that all keys matching the regular expression `order:*` should be migrated from version `v0` to version `v1`. Py-KVolve does not presume any particular versioning scheme; strings like `v0` and `v1` are chosen by the developer. Py-KVolve checks that the names corre-

```
1 for order:* v0→v1 {
2   INIT ['order', ['orderItems'], 'discountedPrice'] {
3     $out = round($base['price'] * 0.7, 2)
4   }
5   REN ['order', ['orderItems'], 'price']→
6       ['order', ['orderItems'], ' fullprice ']
7 };
```

(a) DSL code specifying transformation

```
1 def group_0_update_order( rediskey , jsonobj ):
2   e = jsonobj.get('order').get('orderItems')
3   assert(e is not None)
4   for f in e:
5     assert(f is not None)
6     f['discountedPrice'] = round(f['price'] * 0.7,2)
7     f[' fullprice '] = f.pop('price')
8   return ( rediskey , jsonobj )
```

(b) Generated Python code

Figure 3.9: Specified update to purchase order objects

spond to a single chain of version updates, meaning that the developer cannot write an update for v0 to v1, and then later write an update for v0 to v2.

The body of this statement defines the changes to the values. First, the INIT on Line 2 indicates that a field with path order.orderItems.discountedPrice should be added to the JSON object. This path is defined using a list, where elements like orderItems in brackets indicate the JSON field is itself a list, so the update should be applied to each element of that list on the designated sub-path. The code on Line 3 initializes the new field to 0.7 times the value of the field price, rounded to two decimal places. This code is simply Python code with some DSL-specific variables contained in it, in this case $base and $out. The former is used to refer to the parent field of the one specified in the initialized list; here, therefore $base[' price '] refers to

the price field in the existing object that is a sibling of the newly added field. The $out special variable refers to the added field; here, we are therefore writing to the discountedPrice field. Line 5 indicates that field price should be renamed to fullprice . Again, because orderItems is a list, this rename will be applied to all entries in the list.

DSL programs are compiled into normal Python code that is then stored with the update specification in the database, for later use by the Py-KVolve wrappers. For our example, the generated code is given in Figure 3.9b. This function is called for each object whose key matches the specification; the updated object is returned (along with the potentially updated key, though in this case the key is unaffected). The DSL greatly simplifies the update writing process. The DSL is possible in Py-KVolve because the implementation language was fixed to Python and leaned heavily towards the data having JSON values. (A C-based DSL for KVolve is left to future work described in Section 3.6.2.) The DSL is presented in full in Appendix A.

3.4.2 Implementing lazy updates

This section describes the basic mechanics of Py-KVolve. Unlike KVolve where all updating is handled within single-threaded Redis, with Py-KVolve there may be multiple clients attempting to update the data. To enforce correctness, the Py-KVolve client must issue several Redis commands for each client query, and these extra commands are a major source of overhead in this implementation. (Additionally, re-writing the key string is the other main source of overhead, also described in

this section.) We describe the basic wrapper operations for GET and SET operations.

Get operations: As mentioned in the previous section, keys passed to both GET and SET operations are prepended with version information. In particular, each key k is rewritten to $ver|k$ where ver is the expected version of data indexed by k. For example, the key key:123 might be rewritten to v1|key:123 if the current data version for namespace key is v1. Thus, when data is initially stored in the database using SET operations (discussed below), the data's version is indicated by its key.

When the program calls GET k, the wrapper prepends the current version to k. If this call succeeds it immediately returns the (up-to-date) value. We expect this to be the common case. If the lookup fails, there are two possible reasons: either (a) there simply is no data having the given key, at any version; or (b) there exists an earlier version of the data that must be migrated forward.

To see whether we are in the first case, the wrapper constructs all possible prior names of the key k; we generate all names because there could be very old data in the database (though only ever at one version). For now we ignore the fact that key names may evolve over time (due to **for** namespace commands), and therefore assume that older versions of the key simply differ in the prepended version. The list of names is ordered from newest to oldest, e.g., as ['v1|key:123', 'v0|key:123']. The wrapper then does a bulk-lookup of these keys using Redis' MGET command, returning a list of corresponding values, with None for each key for which no value exists. If the list is all Nones then we are in case (a), and the wrapper can return None. If another client migrated the data in the meantime, the first element in the

list (for the newest version) will be non-None, and so we can return that. Otherwise, we will see something like ['None', '{... JSON value...}'], meaning there is a key at an earlier version, but not the current one; i.e., case (b).

To handle case (b), the wrapper begins a kind of lightweight transaction, called a *pipe*, and sets a *watch* on the new-version key. If the key is set or deleted by another thread during the transaction, it will abort. The wrapper then calls MGET again for the newest key and the non-None key previously discovered. Assuming that both are as they were prior to starting the transaction, the wrapper will generate a new version of the key and/or value using the relevant DSL transformation. (If the object is multiple versions behind, the transformations are composed oldest to newest.) Finally, the wrapper will store any new value (perhaps at a new key name), and delete the old value. Or, if only the key is renamed, it will simply rename the existing key. Then the transaction terminates. If successful, because the new key was not modified by another thread during the transaction, the updated value is returned; otherwise, the wrapper goes to the beginning and tries again, at which point it will find the key has been deleted or that the new version is available.

Set operations: As with a GET, the wrapper for SET first prepends the version to the requested key. Then it performs the Redis call GETSET with the versioned-key, and the value. This call returns any prior version of the value mapped to be this key. If GETSET returns anything, then we know this data was previously migrated to the current version, so no additional versions of the data are present. On the other hand, if it returns nothing, then there are two cases: either (a) there was no

data previously stored for this key, or (b) there was no data *at the current version*, but there could be at an older version. In case (a) we are done; in case (b) we need to delete the old data. To do this, we generate a list of all possible prior keys (following the same procedure as in the analogous case for GET) and delete all data associated with these keys.

3.4.3 Py-KVolve steady state overhead

This section considers the performance impact of Py-KVolve during normal operation. (Additional results are presented in Section A.3.) We find that Py-KVolve adds 5–9% overhead during normal operation. This overhead derives largely from the extra code in the wrapper that is used to lazily migrate data.

Because Redis-bench uses a baked-in client and does not allow us to use our own Py-KVolve client, we recreated the operations of the benchmark as closely as possible. The benchmark uses 50 clients performing 20,000 requests each for a range of operations; for SETs and GETs the operations use a single key with a dummy value matching the one used by Redis-bench. This experiment is similar to the experiment performed with KVolve described in Section 3.3.1 with results in Table 3.1.

Table 3.5 shows the steady state overhead for Py-KVolve compared using unmodified Redis-Py. We show the results for benchmarks involving only GETs (columns 2 and 3) and only SETs (columns 4 and 5). We consider two cases: when all accesses are to keys already present in the database ("no miss", columns 2 and 4), and when some keys are missing ("15% miss", columns 3 and 5). For

	GET (only) no miss	GET (only) 15% miss	SET (only) no miss	SET (only) 15% miss
Py-KVolve	52.36s	53.83s	53.14s	54.04s
Redis-Py	49.89s	50.63s	50.03s	49.76s
Overhead %	4.95%	6.33%	6.21%	8.60%
Overhead (s)	2.47s	3.20s	3.11s	4.28s

Table 3.5: Comparing running time of Py-KVolve and Redis (times in seconds)

the "no miss" cases, we start with a pre-populated Redis database with 20,000 keys (`'key:00000' − 'key:19999'`), with 50 clients (launched as threads) accessing a random key in the same range, for a total of 20,000 accesses per thread. For the tests with misses, we started with 17,000 keys, and the clients used the full key range, missing 15% of the keys from the original database. Prior to populating the keys in the Py-KVolve tests, we also performed a dummy update (basically, incrementing the version of all of the data, but not changing its format) to force Py-KVolve to check for non-existing keys at prior versions in the miss cases.

The 4.95% overhead for `GET`-"no miss" is due entirely to the cost of prepending the version onto the key. For `GET`s with misses, Py-KVolve incurs slightly more overhead (at 6.33%) because it may require additional database accesses to determine whether the miss is due to a genuinely missing value, or because an old value needs to be migrated; note that performance would be worse if not for the use of sentinel values as described in Section A.2.2). For `SET`s, Py-KVolve incurs a bit

more overhead because it performs a GETSET and then checks the return value. In the non-miss case, Py-KVolve returns because it will receive a non-None value, and has overhead at 6.21%. In the miss case, Py-KVolve incurs additional overhead by attempting to delete old versions of the key, raising the overhead to 8.6%.

Even the best-case scenario for Py-KVolve (4.95% - 6.21% with no misses) is much higher than the results for KVolve (-0.77% to 2.49% as shown in Table 3.1), with for string GETs and SETs. (KVolve is not impacted by a miss or hit, so they are not measured separately.) Because of this overhead, Py-KVolve should be used only when the user is not able to modify Redis or strongly prefers the DSL to write update functions.

3.5 Related Work

We now consider work related to KVolve. In the realm of relational databases, the evolution of an application's schema is characterized by the changes to the `CREATE TABLE` statements used to instantiate the schema in subsequent versions of the application. In practice, complex schema changes often require taking the application offline or locking the database tables, such as the update to Wikipedia that held a write lock for 22 hours [120]. Prior research has proposed supporting non-blocking schema changes by accepting out of date copies of database objects [110] or by implementing changes on-the-fly using triggers [86] or log redo [63]. Additionally, several professional tools can perform `ALTER TABLE` operations in a non-blocking manner [21, 73, 76, 99, 108]. Because these tools focus only on the database, the

changes implemented must be backward compatible to avoid breaking the application logic. To avoid this limitation, the Imago system [31] proposed installing the new version in a parallel universe, with dedicated application servers and databases, which allowed it to perform an end-to-end upgrade atomically. This can be achieved in practice by deploying parallel AppEngine [38] applications, at multiple versions. However, this approach duplicates resources and exposes the new version to the live workload only after the data migration was completed.

In contrast, the F1 database from Google implemented an asynchronous protocol [79] for adding and removing tables, columns and indexes, which allows the servers in a distributed database system to access and update all the data during a schema change and to transition to the new schema at different times. This is achieved by having stateless database servers with temporal schema leases, by identifying which schema-change operations may cause inconsistencies, and by breaking these into a sequence of schema changes that preserve database consistency as long as servers are no more than one schema version behind. Google's Spanner distributed key-value store [26] (which provides F1's backend) supports changes to the format of keys and values by registering schema-change transactions at a specific time in the future and by utilizing globally synchronized clocks to coordinate reads and writes with these transactions. These systems do not address changes to the format of Protocol Buffers stored in the F1 columns or Spanner values [87] or inconsistencies that may be caused by interactions with (stateful) clients using different schemas [32].

Schema evolution in NoSQL databases is less well understood, as these databases

do not provide a data definition language for specifying the schema. There is no true limit to the format that the data may be stored in. However, many applications attach meaning to the format of the keys and values stored in the database, and these formats may evolve over time. For example, the values often correspond to data structures serialized using JSON [50], or are described with tags in XML [25], or scientific data formats, or are stored in a binary format like Thrift [12], Protobufs [39], or Avro [10]. The latter formats have schema-aware parsers, which include some support for schema changes, e.g. by skipping unknown fields or by attempting to translate data from the writer schema into the reader schema [57]. However, orchestrating the actual changes to the data and the application logic is entirely up to the programmer.

One approach to defining schema changes defines a declarative schema evolution language for NoSQL databases [97]. This language allows specifying more comprehensive schema changes and enables the automatic generation of database queries for migrating eagerly to the new schema. (While the paper also mentions the possibility of performing the migration in a lazy manner, which is needed for avoiding downtime, design and implementation details are not provided.) Another approach uses a DSL for describing data schema migrations for Haskell datatypes [43]. Many other approaches [8, 28, 80, 118] have focused on the problem of synthesizing the transformation code to migrate from one schema version to the next, and the transformation is then typically applied offline, rather than incrementally online. In this work, we are not focusing on the problem of synthesizing the transformation code for now, and in any case it is a much simpler affair in the key-value setting. Rather,

we focus on how to apply a transformation without halting service.

In practice, developers are often advised to handle all the necessary schema changes in custom code, added to the application logic that may modify the data in the database [17, 83, 87, 109]. This approach adds burden on the programmers, results in complex code that mixes application and schema-maintenance logic, does not provide a mechanism for reasoning about the correctness of schema changes performed concurrently with the live workload and often leave outdated entries in the database.

Our work is also related to the body of research on dynamic software updates, described in Section 1.1.2. However, with the exception of a position paper [30], these approaches focus on changes to code and data structures loaded in memory, rather than changes to the formats of persistent data stored in a database.

3.6 Future Work

We plan to release our code and make it freely available. We also have some additional ideas for follow-on work, described in this section.

3.6.1 Distributed data updates

KVolve demonstrated the ability to dynamically and lazily update the data stored in a single instance of Redis. However, newer versions of Redis introduced Redis Cluster (released with version 3.0.0 in April 2015), which is a distributed implementation of Redis. Unlike Redis' original strategy of having multiple proxies

or slaves create redundancy, Redis Cluster is a fully distributed implementation that can scale to several thousand instances and allows nodes to fail and rejoin the cluster. This clustering is not specific to Redis and has previously been implemented in several other large database systems, such as in MongoDB [2]. Additionally, the common practice of database sharding, where some of the data may be located on physically separate nodes also adds complication and further distribution throughout the database system.

This could present a challenge to orchestrating an update across all instances of Redis (or other similar database system). Currently we use a single instance of Redis as a central coordinator to store and orchestrate the updates. An interesting research problem would be to investigate how to coordinate and bound the update notification throughout the cluster without using a central point of update coordination. We could attempt to expand on existing distributed coordination algorithms and also carefully define the bounds of consistency, making sure that no safety violations occur, such as accessing a backward-incompatible schema value.

3.6.2 Automatic generation of transformation functions in NoSQL updates

Our initial implementation of KVolve, described in Section 3.4, included a DSL facilitating the generation of schema transformation functions for JSON. The current implementation of KVolve detailed in this chapter provides an extreme overhead improvement (from 5-8% overhead in our original implementation to virtually

no overhead), but it has the drawback that the transformation functions must be written in C. This is because now the transformation functions are called from within Redis itself, which does not provide any language flexibility.

Ideally, the update-writer would be able to write the transformation function in a language of his choice, or to write the transformation function in a DSL similar to the one presented in Appendix A. Future work in this area would involve investigating how to best implement this functionality, focusing on providing full expressiveness in the ability to describe the update, while making the tool easy for the update-writer to utilize.

3.7 Conclusion

This chapter has presented KVolve, a general approach to evolving a NoSQL database without downtime. KVolve adapts Redis to migrate data as it is accessed, reducing downtime that would otherwise result during a data upgrade, and minimizing required changes to applications. We find that KVolve imposes essentially no overhead when not performing an update, and minimal overhead when performing an update.

Author Contributions: This work was presented in a paper [94]. I was lead author on that paper and performed all of the programming, development, and experiments, and led in the writing and overall design. My collaborators, Michael Hicks and Tudor Dumitraş, contributed to the writing and to refining the ideas and

the design of this work.

Chapter 4: Morpheus: Safe and Flexible Dynamic Updates for Software-Defined Networks

Software-defined networking (SDN) controllers are complex software systems that must simultaneously implement a range of interacting services such as topology discovery, routing, traffic monitoring, load balancing, authentication, access control, and many others. Like any non-trivial system, SDN controllers must be periodically updated to add features, improve performance, and fix bugs. However, in many networks, downtime is unacceptable, so updates must be deployed *dynamically*, while the network is in operation and in such a way as to minimize disruptions.

In general, dynamic updates differ from static ones in that while modifying the *program code* they must also be concerned with the current *execution state*. In SDN, this state can be divided into the *internal state* stored on controllers (e.g., in memory, file systems, or databases), and the *external state* stored on switches (e.g., in forwarding rules). A key challenge is that updated code may make different assumptions about state—e.g., using different formats to represent internal data structures or installing different rules on switches. This challenge is exacerbated in SDN, where state does not reside at a single location but is instead distributed

across multiple controllers and switches.

Existing approaches: Most SDN controllers today employ one of two strategies for performing dynamic updates, distinguished by how they attempt to ensure correct, post-update execution state.

• In *simple restart*, the system halts the old controller and begins executing a fresh copy of the new controller. In doing so, the internal state of the old controller is discarded (except for persistent state—e.g., stored in a database), under the assumption that any necessary state can be reconstructed by the new controller after the restart. One simple strategy for recovering internal state is to delete the forwarding rules installed on switches so future network events are sent to the controller, which can use them to populate its state. This behavior is available by default in open-source SDN platforms such as POX [4] and Floodlight [1].

• In *record and replay*, the system maintains a trace of network events received by the old controller. During an update, the system first replays the logged events to the new controller to "warm up" its internal state and then swaps in the new controller for the old one. By giving the new controller access to the events that were used to generate the internal and external state for the old controller, it is possible to avoid the issues that arise with less direct mechanisms for reconstructing state. Record and replay has been used effectively in several previous systems including HotSwap [116], OpenNF [36], and a recent system for managing middleboxes [104]. A related approach is to attempt to reconstruct the state from the existing forwarding rules on the switches, rather than from a log. According to private discussions with SDN operators, this approach is often adapted by proactive controllers that

do not make frequent changes to network state (e.g., destination-based forwarding along shortest paths).

Unfortunately, neither approach constitutes a general-purpose solution to the dynamic update problem: they offer little control over how state is reconstructed and can impose excessive performance penalties. Simple restarts discard internal state, which can be expensive or impossible to reproduce. In addition, there is no guarantee that the reconstructed state will be harmonious with the assumptions being made by end hosts—i.e, existing flows may be routed along different paths or even to different destinations, breaking socket connections in applications. Record and replay can reproduce a harmonious state, but requires a complex logging system that can be expensive to run, and still provides no guarantees about correctness—e.g., in cases where the new controllers would have generated a different set of events than the old controller did. Reconstructing controller state from existing forwarding rules can be laborious and error prone, and is risky in the face of inevitable switch failures. We illustrate these issues using examples in Section 4.1.

Our approach: Update by state transfer: The techniques just discussed *indirectly* update network state after an update. This chapter proposes a more general and flexible alternative: to properly support dynamic updates, operators should be able to *directly* update the internal state of the controller, as a function of its current state. We call this idea *dynamic update by state transfer*.

To support this dynamic update technique, controllers must offer three features: (1) they need a way of making their internal state available; (2) they need a way of initializing a new controller's state, starting from (a possibly transformed

version of) the old controller's state; and (3) they need a way to coordinate behavior across components when updates happen, to make sure that the update process yields a consistent result. This basic approach has been advocated in prior DSU work described in previous chapters.

Update by state transfer directly addresses the performance and correctness problems of prior approaches. There is no need to log events, and there is no need to process many events at the controller, whether by replaying old events or by inducing the delivery of new ones by wiping rules. Moreover, the operator has complete control over the post-update network state, ensuring that it is harmonious with the network—e.g., preserving existing flows and policies. The main costs are that the network programmer must write a function (we call it μ) to initialize the new controller state, given the old controller state, and the controller platform must provide protocols for coordinating across distributed nodes.

Fortunately, our experience (and that of DSU generally) is that for real-world updates this function is not difficult to write and can often be partially automated. The changes to the controller platform needed to support state initialization and coordination between nodes adds complexity, but they are one-time changes and are not difficult to implement. Moreover, the cost of coordinating controller nodes is likely to be reasonable, since the distributed nodes that make up the controller are likely to be relatively small and either co-located on the same machine or connected through a fast, mostly reliable network.

Morpheus: A controller with update by state transfer: We have implemented our approach in Morpheus, a new distributed controller platform based

on Frenetic [9,34,82], described in Section 4.2. Our design is based on a distributed architecture similar to the one used in industrial controllers such as Onix [58] and ONOS [18]. We considered modifying a simpler open-source controller such as POX or Floodlight [1, 4], but decided to build a new distributed controller to provide evidence that update by state transfer will work in industrial settings.

Morpheus employs a NIB, basic controller replicas, and standard applications for computing and installing forwarding paths, each running as separate applications. Persistent internal state is stored in the NIB, which can be accessed by any application. When an application starts (or restarts, e.g., after a crash) it connects to the NIB to access the state it needs, and publishes updates to the state while it runs. Applications coordinate rule deployments to switches via controller replicas, which use NetKAT [9] to combine policies into a unified policy, and can use consistent updates [82] to push rules to switches. The NIB utilizes KVolve, presented previously in Chapter 3, (Specifically, Morpheus uses a variant of KVolve, a Python client version of KVolve that is described in Section 3.4, along with the pros and cons of the Python client version.)

Supporting update by state transfer requires only a few additions to Morpheus's basic design, described in Section 4.3. The relevant state is already available for modification in the NIB, so we just need a means of modifying that state to work with the new versions.

We also need to coordinate the update across the affected applications. To see why this is important, consider a situation in which we have several routing application replicas, each responsible for a subset of the overall collection of switches.

Now suppose we wish to deploy a dynamic update that changes which paths forward traffic through the network. It is clear we must update all of the replicas in a coordinated manner, or else some of the replicas could implement old paths and others implement new paths, leading to anomalies including loops, black holes, etc. Morpheus's simple coordination protocol operates in three steps: (1) *quiescence*—the affected applications are signaled and paused before the update begins; (2) *installation*—the μ function is registered with the NIB for the purposes of transforming the state; and (3) *restart*—the updated applications are restarted, using μ to update the NIB state (in a coordinated way). After the state is updated, they send updated policies to the controller replicas which compose them and generate rules to install on switches.

Using Morpheus we have written several applications, and several versions of each, including a stateful firewall, topology discovery, routing, and load balancing. Through a series of experiments, described in Section 4.4, we demonstrate the advantages of update by state transfer, compared to simple restarts and record-and-replay. In essence, there is far less disruption, and no incorrect behavior. We also find that the μ functions are relatively simple, and an investigation of changes to open-source controllers suggests that μ functions for realistic application evolutions would be simple as well.

Summary: This chapter's contributions are as follows:

- We study the problem of performing dynamic updates to SDN controllers and identify fundamental limitations of current approaches.

- We propose a new, general-purpose solution to dynamic update problem for SDNs—*dynamic update by state transfer*. With this solution, the programmer explicitly transforms old state to be used with the new controller, and an accompanying protocol coordinates the update across distributed nodes.
- We describe a prototype implementation of these ideas in the Morpheus system.
- We present several applications as case studies as well as experiments showing that Morpheus implements updates correctly and with far less disruption than current approaches.

Next, we present the design and implementation of Morpheus (§4.1-4.3), our experimental evaluation (§4.4), and a discussion of related work and conclusion (§4.5-4.7).

4.1 Overview

This section explains why existing approaches for handling dynamic updates to SDN controllers are inadequate in general, and provides detailed motivation for our approach based on *state transfer*.

4.1.1 Simple restart

As an example, suppose the SDN controller implements a stateful firewall, as depicted in Figure 4.1. The topology consists of a single switch connected to trusted internal hosts and untrusted external hosts. Initially the switch has no forwarding rules, so it diverts all packets to the controller. When the controller receives a

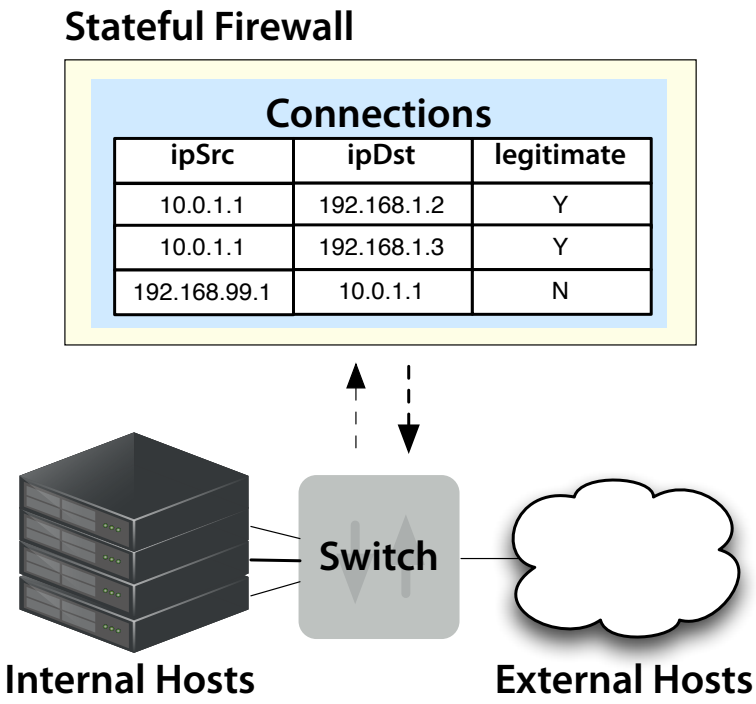

Figure 4.1: Example application: stateful firewall.

packet from a trusted internal host, it records the internal-external host pair in its (internal) state and punches a hole in the firewall so the hosts can communicate. Conversely, if the controller receives a packet from an external host first, it logs the connection attempt and drops the packet.

Now suppose the programmer wishes to update the firewall so that if an external host tries to initiate more than n connections, then it is blacklisted from all future communications. With simple restart, the old controller would be swapped out for a new controller that contains no record of connections initiated by internal and external hosts. In the absence of any other information about the state prior to the update, the controller would delete the rules installed on the switch to match

its own internal state, which is empty. This leads to a correctness problem:[1] If the external host of an active connection sends the first few packets after the rules are wiped, then those packets will be interpreted as unsolicited connection attempts. The host could easily be blacklisted even though it is merely participating in a connection initiated previously by an internal host.

This problem stems from the fact that the old controller's state is discarded by the simple restart. In this example, it could be avoided by storing key internal state outside of the controller process's memory—e.g., in a separate *network information base* (NIB), as is done in controllers such as Onix [58]—and indeed, we do exactly this in our Morpheus controller. However, in general, safe dynamic updates require more than externalized state, as we discuss in Section 4.1.3—e.g., in the case that the new version expects the state in a new format and multiple controllers share this state.

4.1.2 Record and replay

At first glance, record and replay seems like it might offer a fully automatic solution to dynamic controller updates. The HotSwap (HS) system [116], a noteworthy example of this approach, records a trace of the events received by the old controller and replays them to the new controller to "warm up" its internal state before swapping it in. For the stateful firewall, HS would replay the network events for each connection initiated by an internal host and so would easily reconstruct the

[1] It may also be disruptive: if unmatched traffic is sent to the controller, then the new controller will essentially induce a DDoS attack against itself as a flood of packets stream in.

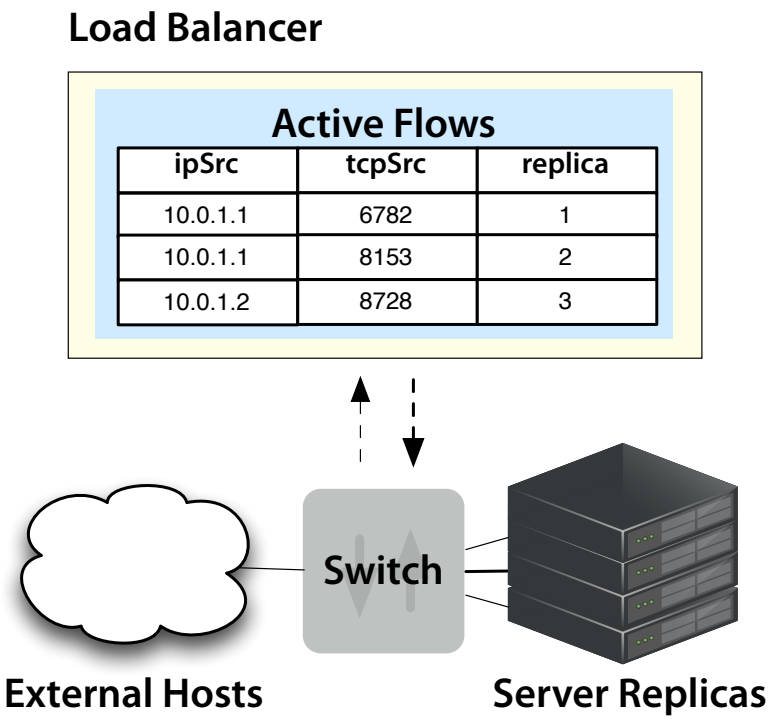

Figure 4.2: Example application: load balancer.

set of existing connections, avoiding the problems with the simple restart approach. Moreover, because record and replay works with events drawn from a standard API like OpenFlow, it is fully "black box"—the implementation details of the old and new controllers are immaterial.

Unfortunately, record and replay has several limitations that prevent it from being a full solution to the dynamic update problem. One obvious issue is overhead: in general, unless the system has prior knowledge of the new controller's functionality (which it will not, in general), the system will have to record (and replay) all relevant events that contributed to the network's current state. Doing this can be prohibitively expensive in a large, long-running network.

Another issue is that the recorded trace may not make sense for the new

controller, and therefore replaying it may result in an incorrect state, for the following reasons. The new controller may, in general, behave differently than the old one—e.g., it may install different forwarding rules on switches. As such, if the new controller had been used from the start, these rules might have caused a different set of network events to be generated than those that were actually recorded. Such events could have been induced *directly* due to different rules (e.g., because they handle fewer or more packets compared to the old rules) or they might have been induced *indirectly* (e.g., because the new rules elicit different responses from the hosts that are communicating via the network). Establishing that an update is correct under these circumstances is extremely difficult in general.

To illustrate, consider the example of a server load balancer, as depicted in Figure 4.2. The topology consists of a single switch with one port connected to a network of external hosts and another n ports connected to back-end server replicas. Initially, the switch has no rules, so all packets are diverted to the controller. Upon receiving a new connection from an external host, the controller picks a server replica (e.g., uniformly at random) and installs rules that forward traffic in both directions between the host and the selected server. The controller also records the selected server in its internal state (e.g., so it can correctly repopulate the forwarding rules if the switch drops out and later reconnects).

Now suppose the programmer wishes to dynamically deploy a new version of the controller where the selection function selects the least loaded server and also puts a cap c on the number of open connections to any given server, and refuses connections that would cause a servers to exceed that cap. During replay, the

new controller would receive a network event for each existing connection request. However, it would remap those connections to the least loaded server instead of the server previously selected by the old controller. In general, the discrepancy between these two load balancing strategies will break connection affinity—a different server replica may receive the ith packet in a flow and reset the connection.

Attempting to reconstruct the controller state from querying the switch state could also be problematic. Although the new controller would have the information needed to generate forwarding rules that preserve connection affinity, writing the controller to retrieve this information is potentially laborious, error-prone work for the programmer. And it may require modifications to the code; e.g., if the new controller uses statically allocated data structures to keep track of active flows (something that is possible due to the cap c), it may be incorrect to exceed the cap. We would prefer a solution that is simpler and more systematic.

4.1.3 Solution: update by state transfer

This chapter proposes a different approach to the SDN dynamic update problem. Rather than attempting to develop fully automated solutions that handle certain simple cases but are more awkward or impossible in others, we propose a general-purpose solution that attacks the fundamental issue: *dynamically updating the state.* The above approaches attempt to *indirectly* reconstruct a reasonable state, but they lack sufficient precision and performance to fully solve the problem.

Our approach, which we call *update by state transfer*, solves the dynamic up-

date problem by giving the programmer *direct* access to the running controller's state, call it σ, along with a way enabling the new controller with an existing state, call it σ', such that the new state can be constructed as a function, call it μ, of the old state so that $\sigma' = \mu(\sigma)$. In addition, our approach requires a means to signal the controller that an update is available so that it can *quiesce* prior to performing the update. This mechanism ensures that σ is consistent (e.g., is not in the middle of being changed) before using μ to compute σ'.

Consider the problematic examples presented thus far. For both the firewall update and the load balancing update, the state transfer approach is trivial and effective: setting the μ function to a no-op (i.e., identity function) grandfathers in existing connections and the new semantics is applied to new connections. Pleasantly, for the load-balancing update, any newly added replicas will receive all new connection requests until the load balances out.

Another feature of update by state transfer is that it permits the developer to more easily address updates that are backward-incompatible, such as the load balancer with a cap c discussed above. In these situations, the current network conditions may not constitute ones that could ever be reached had the new controller been started from scratch. With state transfer, the operator can either allow this situation temporarily by preserving the existing state, with the new policy effectively enforced once the number goes below the cap. Or she can choose to kill some connections, to immediately respect the cap. The choice is hers. By contrast, prior approaches will have unpredictable effects: some connections may be reset while others may be inadvertently grandfathered in, unbeknownst to the controller.

In addition to its expressiveness benefits, update by state transfer has benefits to performance: it adds no overhead to normal operation (no logging), and far less disruption at update-time (only the time to quiesce the controller and update the state). The main cost is that the network service developer needs to write μ, which will not always be a no-op. For example, if we updated a routing algorithm from using link counts to using current bandwidth measurements, the controller state would have to change to include this additional state. Fortunately, according to our experience (and that of a substantial body of work in the related area of *dynamic software updating*), μ tends to be relatively simple, and its construction can be at least partially automated.

4.2 Morpheus Controller

To provide a concrete setting for experimenting with dynamic SDN updates, we have implemented a new distributed controller called Morpheus, implemented in Python and based on the Frenetic libraries [9, 34, 82]. Our design follows the basic structure used in a number of industrial controllers including Onix [58] and ONOS [18], but adds a number of features designed to facilitate staging and deployment of live updates. This means that it should be easy to adapt update techniques developed in the context of Morpheus for use in other controllers as well. We should note that our aim is to support updates to the applications running on the controller, but not necessarily the controller itself. In the future, we plan to investigate extensions that will also support updates to the controller infrastructure—e.g., migrating

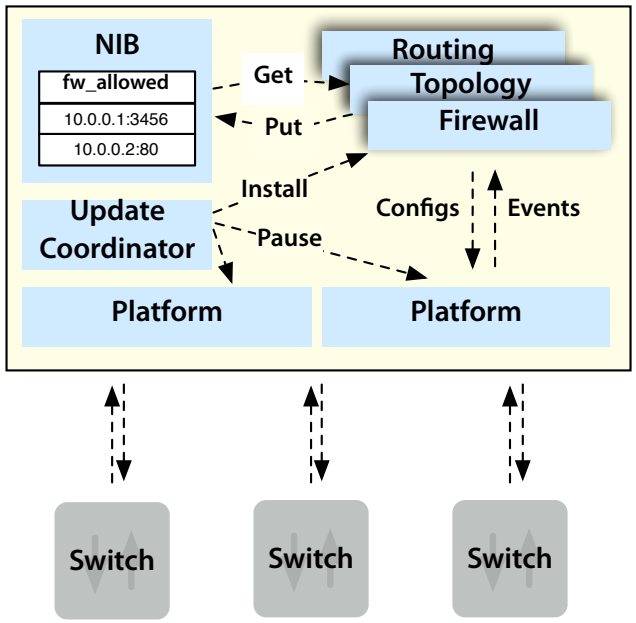

Figure 4.3: Morpheus architecture.

to a new protocol for communicating with SDN switches.

4.2.1 Architecture

Morpheus's architecture is shown in Figure 4.3. The controller is structured as a distributed system in which nodes communicate via well-defined message-passing interfaces. Morpheus provides four types of nodes:

- platform nodes (PLATFORM), which are responsible for managing low-level interactions with SDN switches and interfacing with applications,
- a network information base (NIB), which provides persistent storage for application state,
- an update coordinator (UPDC), which implements distributed protocols for staging and deploying updates, and

- application nodes (TOPOLOGY, ROUTING, etc.), which implement specific kinds of functionality, such as discovering topology or computing shortest paths through the topology.

Each node executes as a separate OS-level process, supporting concurrent execution and isolation. Processes also make it easy to use existing OS tools to safely spawn, execute, replicate, kill, and restart nodes.

4.2.2 Components

We now describe Morpheus's components in detail.

Platform: The most basic components are PLATFORM nodes, which implement basic controller functionality: accepting connections from switches, negotiating features, responding to keep-alive messages, installing forwarding rules, etc. The PLATFORM nodes implement a simple interface that provides commands for interacting with switches:

- `event()` returns the next network event,
- `update(pol)` sets the network configuration to `pol`, specified using NetKAT [9],
- `pkt_out(sw,pkt,pt)` injects packet `pkt` into the network at `sw` and `pt`,

as well as commands for synchronizing with the UPDC during dynamic updates:

- `pause()` temporarily stops propagating configurations to the network, and
- `resume()` resumes propagating configurations.

When multiple Morpheus applications are operating, the PLATFORM nodes make every network event available to each application by default. If needed, filtering can be applied to prevent some applications from seeing some network events. Likewise, the policies provided by each application are combined into a single network-wide policy using NetKAT's modular composition operators [9]. For scalability and fault tolerance, Morpheus would typically use several PLATFORM nodes that each manage a subset of the switches. These nodes would communicate with each other to merge their separate event streams into a single stream, and similarly for NetKAT policies. For simplicity, our current implementation uses a single PLATFORM node to manage all of the switches in the network.

Network Information Base: Morpheus applications store persistent state in the NIB. The information in the NIB is guaranteed to be preserved across application executions, thereby avoiding disruption if individual applications stop and restart. The NIB provides a simple interface to a NoSQL key-value store, which can be used to store persistent state.[2] Morpheus's store is currently based on Redis [89] and although it currently uses a single node, Redis supports clustering for better scalability.

Data stored in the NIB is divided among conceptual *namespaces*, organized according to the applications that use it. For example, a firewall application might store information in the NIB in the `fw_allowed` namespace about which hosts are currently allowed. An application may access data in multiple namespaces, where it

[2]Obviously, applications may also maintain their own in-memory state for efficiency reasons, but this state is lost on restart.

might be the conceptual data owner for one, but a consumer of another. For example, our TOPOLOGY application discovers the structure of the network by interacting with the PLATFORM nodes, and stores the topology persistently in the `topology` namespace.

Redis does not support namespaces directly (some other NoSQL databases do) so we encode the namespace as a prefix of the keys under which we store a application's data values. Many Morpheus applications also use Redis' built-in publish-subscribe mechanism to handle frequently changing data. For example, TOPOLOGY publishes a notification to a channel any of the keys in the topology namespace changes, and ROUTING subscribes to this channel and updates its routing configuration appropriately when it receives a notification that the topology has changed.

Applications: Applications running on top of Morpheus follow a common design pattern. Upon startup, they connect with the NIB to retrieve any relevant persistent state. The application then adds to, and retrieves from, the persistent store any other necessary data depending on its function. For example, TOPOLOGY discovers and stores hosts, switches, edges, and additional information about the topology in the NIB, and when ROUTING starts up it reads this information and then adds the least-cost paths to each destination. During normal operation, applications are *reactive*: they will process events from the PLATFORM and from other applications (e.g., via the pub-sub mechanism). In response, they will make changes to the NIB state and push out a new NetKAT program via the `update` function on the PLATFORM nodes, which will update in the switches.

Update Coordinator: Because Morpheus has a distributed architecture, dynamic updates require coordination between nodes. Morpheus uses an update coordinator (or UPDC) that manages interactions between nodes during an update. We discuss these interactions in detail in the next section.

4.3 Dynamic updates with Morpheus

Morpheus's design supports dynamic updates by allowing important state to persist in the NIB between versions while providing a way to transform that state when required by an update. To ensure consistent semantics, Morpheus's UPDC node organizes updates to the affected applications using a simple protocol. This section describes this protocol, and then describes some example updates that we have performed.

4.3.1 Update protocol

To deploy an update, the operator provides UPDC with the following update specification:

- New versions of the affected applications' code
- A state transformation function μ that maps the existing persistent state in affected namespaces into a format suited to the new application versions.

As a convenience, the application programmer can write μ in a DSL (Domain-Specific Language) we developed for writing transformers over JSON values (inspired by Kitsune's *xfgen* language [46]), illustrated briefly in Sections 4.3.2 and 4.3.3. This

language's programs are compiled to Python code that takes an old JSON value and produces an updated version of it.[3] Alternatively, the user can write μ using standard Python code.

Given the update specification, UPDC then executes a distributed protocol that steps through four distinct phases: (i) application quiescence, (ii) code installation and state transformation, (iii) application restart, and (iv) controller resumption.

1. Quiesce the applications: UPDC begins by signaling the applications designated for an update. The applications complete any ongoing work and shut down, signaling UPDC they have done so. (A *configurable* timeout is used to forcibly shut down applications that do not exit gracefully, but this should be used as a last resort as this may cause an application to lose some state.) At the same time, UPDC sends the list of applications to the PLATFORM, which will temporarily suppress any rules updates made by those applications, which could be stale. Once all applications have exited, and the PLATFORM has indicated it has begun blocking the rules, Morpheus has reached *quiescence*.

2. Install the update in the* nib*: Next, UPDC installs the administrator-provided μ functions at the NIB. The NIB verifies that these functions make sense, e.g., that if the request is to update for namespace `nodes` from versions `v3->v4`, then the current NIB should contain namespace `nodes` at version `v3`. All transformations will be applied *lazily* (as described in the introduction to Chapter 3), as part of in step 4.

[3]While the programmer currently must write μ, automated assistance is also possible [46, 78].

3. Restart the applications: Now UPDC begins the process of resuming operation. UPDC signals the new versions of the affected applications to start up. These applications connect to the NIB, and the NIB ensures that the applications' requested version matches the version just installed in the NIB. The applications then retrieve relevant state stored in the NIB, and compute and push the new rules to the PLATFORM. The PLATFORM receives and holds the new rulesets. It will push them once it has received rules (or otherwise been signaled) from *all* of the updated applications, to ensure that the rules were generated from consistent software versions. Once the PLATFORM has received rules from all updated applications, it will remove the old rules previously created by the updated applications and install the new rules on the switches.

4. Resume operation: At this point, the update is fully loaded and the applications proceed as normal. As the applications access data in the NIB, any installed μ function is applied lazily. In particular, when an application queries a particular key, if that key's value has not yet been transformed, the transformer is invoked at that time and the data is updated.

The rest of this section describes some example updates we have implemented in Morpheus for a stateful firewall, and for TOPOLOGY and ROUTING applications.

4.3.2 Update example: Firewall

We developed three different versions of a stateful firewall, and defined updates between them.

- FIREWALL$\leftharpoondown$ permits bidirectional flows between internal and external hosts as long as the connection is initiated from the inside. When the controller sees an outbound packet from internal host S to external host H, it installs forwarding rules permitting communication between the two.

- FIREWALL$\rightleftharpoons$ acts like FIREWALL$\leftharpoondown$ but only installs the rules permitting bidirectional flows after seeing returning traffic following an internal connection request. (It might do this to prevent attacks on the forwarding table originating from a compromised host within the network.)

- FIREWALL$\rightleftharpoons\odot$ adds to FIREWALL$\rightleftharpoons$ the ability to time out connections (and uninstall their forwarding rules) after some period of inactivity between the two hosts.

FIREWALL$\leftharpoondown$ defines a namespace `fw_allowed` that keeps track of connections initiated by trusted hosts, represented as JSON values:

```
{ "trusted_ip": "10.0.0.1",
  "trusted_port": 3456,
  "untrusted_ip": "10.0.0.2",
  "untrusted_port": 80 }
```

Updating from FIREWALL$\leftharpoondown$ to FIREWALL$\rightleftharpoons$ requires the addition of a new namespace, called `fw_pending`; the keys in this namespace track the internal hosts that have sent a packet to an external host but have not heard back yet. Once the return packet is received, the host pair is moved to the `fw_allowed` namespace. For

this update, no transformer function is needed: all connections established under the FIREWALL↽ regime can be allowed to persist, and new connections will go through the two-step process.[4]

Updating from FIREWALL⇌ to FIREWALL⇌⊙ requires updating the data in the `fw_pending` and `fw_allowed` namespaces, by adding two fields to the JSON values they map to, `last_count` and `time_created`, where the former counts the number of packets exchanged between an internal and external host as of the time stored in the latter. Every N seconds (for some N, like 3), the firewall application will query the NIB to see if the packet count has changed. If so, it stores the new count and time. If not, it removes the (actual or pending) route.

In our DSL we can express the transformation from FIREWALL⇌ to FIREWALL⇌⊙ data for the `fw_allowed` namespace as follows:

```
for fw_allowed:* ns_v0->ns_v1 {
  INIT ["last_count"] {$out = 0}
  INIT ["time_created"] {$out = time.time()}
};
```

This states that for every key in the namespace, its corresponding JSON value is updated from version `ns_v0` (corresponding to FIREWALL⇌) to `ns_v1` (corresponding to FIREWALL⇌⊙) by adding two JSON fields. We can safely initialize the `last_count` field to 0 because this is a lower bound on the actual exchanged

[4]We could also imagine moving all currently approved connections to the pending list, but the resulting removal of forwarding rules would be unnecessarily disruptive.

packets, and we can initialize `time_created` to the current time. Both values will be updated at the next timeout. In general, our DSL can express transformations that involve adding, renaming, deleting field names, modifying any data stored in the fields, and also renaming the keys themselves. The DSL is detailed in full in the previous chapter.

The above code will be compiled to Python code that is stored (as a string) in Redis and associated with the new version. The existing data will be transformed as the new version accesses it via the NIB accessor API. When the new version of the program retrieves connection information from the NIB, the transformation would add the two new fields to the existing JSON value shown earlier in this section:

key: `fw_allowed:10.0.0.1_3456_10.0.0.2_80`

value:
```
{ "trusted_port": 3456,
  "untrusted_port": 80,
  "trusted_ip": "10.0.0.1",
  "untrusted_ip": "10.0.0.2",
  "last_count": 0,
  "time_created": 1426167581.566535 }
```

4.3.3 Coordination: Routing and Topology

In the above example, the firewall is storing its own data in the NIB with no intention of sharing it with any other applications. As such, we could have killed the application, installed the update, and started the new version. However, when

multiple applications share the same data and its format changes in a backward-incompatible manner, then it's critical that we employ the update protocol described in Section 4.3.1, which gracefully coordinates the updates to applications with shared data.

As an example coordinated update, recall from Section 4.2 that our ROUTING and TOPOLOGY applications share topology information stored in the NIB. In its first version, TOPOLOGY merely stores information about hosts, switches, and the links that connect them. The ROUTING application computes per-source/destination routes, assuming nothing about the capacity or usage of links. In the next version, TOPOLOGY regularly queries the switches for port statistics and stores the moving average of each link's bitrate in the NIB. This information is then used by ROUTING when computing paths. The result should be better load balancing when multiple paths exist, between hosts.

Updating from the first to the second version in Morpheus requires adding a field to the JSON object for edges, to add the measured bitrate. The transformer μ simply initializes this field to 1, indicating the default value for traffic on the link as follows:

```
for edge:* ns_v0->ns_v1 {
  INIT ["weight"] {$out = 1}
};
```

As such, the initial run of the routing algorithm will reproduce the existing routes because all initial values will be the same, ensuring stability. Subsequent ROUTING

computations will account for and store the added usage information and thus better balance the routes.

4.4 Experiments and Evaluation

In this section, we report the results of experiments where we dynamically update several canonical SDN applications: a load balancer, a firewall, and a routing application. We implement three dynamic update mechanisms: state transfer using Morpheus, simple restart, and record and replay. In all cases, state transfer is fast and disruption-free, whereas the other techniques cause a variety of problems, from network churn to dropped connections. We ran all experiments using Mininet HiFi [44], on an Intel(R) Core(TM) i5-4250U CPU @ 1.30GHz with 8GB RAM. We report the average of 10 trials.

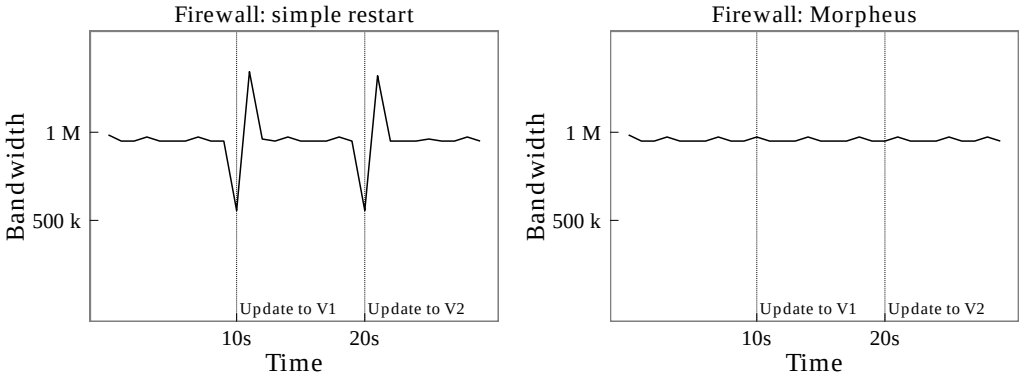

Figure 4.4: Firewall update

4.4.1 Firewall

Figure 4.4 illustrates a dynamic update to the firewall, described in Section 4.3.2, from FIREWALL$\leftharpoondown$ to FIREWALL$\rightleftharpoons$ and then to FIREWALL$\rightleftharpoons\odot$. The figure shows the result of simple restart (where all data is stored in memory and lost on restart) and state transfer (where data is stored in the NIB). We do not depict record and replay, which happens to perform as well as state transfer for this example (as per Section 4.1.2).

For the experiment, we used a single switch with two ports (with 1 MBPS bandwidth) connected to two hosts. One host is designated the client inside the firewall and the other is the server outside the firewall. Using iperf, we establish a TCP connection from the client to the server. The figure plots the bandwidth reported by iperf over time. In both experiments, we update to FIREWALL$\rightleftharpoons$ after 10 seconds and FIREWALL$\rightleftharpoons\odot$ after 20 seconds.

Using simple restart, the figure shows that bandwidth drops significantly during updates. This is unsurprising, since a newly started firewall doesn't remember existing connections. Therefore, FIREWALL$\rightleftharpoons$ and FIREWALL$\rightleftharpoons\odot$ first block all packets from the server to the client, until the client sends a packet, which restores firewall state. In contrast, Morpheus doesn't drop any packets because state is seamlessly transformed from one version to the next.

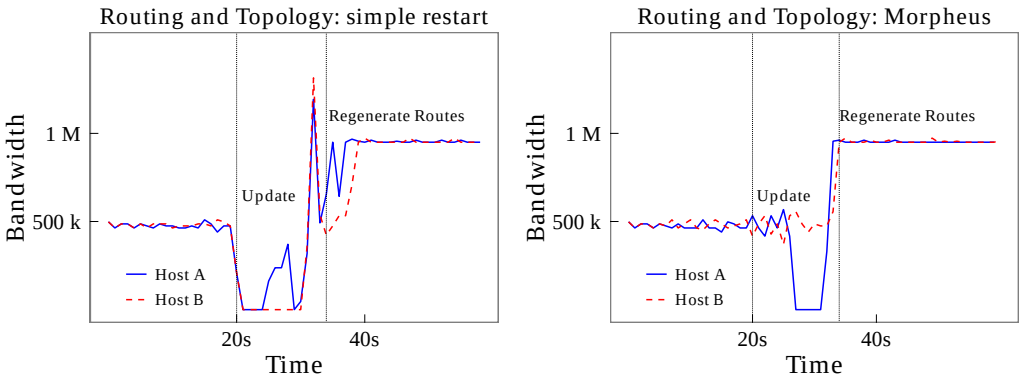

Figure 4.5: Routing and topology discovery update

4.4.2 Routing and Topology

Figure 4.5 shows the effect of updating routing and topology applications (described in section 4.3.3), where the initial version uses shortest paths and the final version takes current usage into account. The experiment uses four switches connected in a diamond-shaped topology with a client and server on either end. Therefore, there are two paths of equal length through the network. The client establishes two iperf TCP connections to the server.

Initially, both connections are routed along the same path because the first version of TOPOLOGY and ROUTING pick the same shortest path. The links along the path are 1MBPS, therefore each connection gets 500KBPS by fair-sharing. After 20 seconds elapse, we update both applications: the new version of TOPOLOGY stores link-utilization information in the NIB and the new version of ROUTING using this information to balance traffic across links. After the update, each connection should be mapped to a unique path, thus increasing link utilization and the bandwidth reported by iperf.

start	*apps exit*	*restart begins*	*rout push*	*topo push*	*platform resume*
0.00s	0.05s	0.11s	1.67s	1.68s	1.70s

Table 4.1: Update quiescence times for TOPOLOGY and ROUTING (median of 11 trials)

Using simple restart, both connections are disrupted for 10 seconds, which is how long TOPOLOGY takes to learn the network topology. Until the topology is learned, routing can't route traffic for either connection. Morpheus is much less disruptive. Since the state transfer function preserves topology information, the new ROUTING module maps each connection to a unique path. The connection that is not moved (Host B) suffers no disruption and gracefully jumps to use 1MBPS bandwidth. The connection that is moved (Host A) is briefly disrupted as several switch tables are updated. Even this disruption could be avoided using a consistent update [82].

Table 4.1 breaks down the time to run the update protocol for this update. It takes .05s for both TOPOLOGY and ROUTING to receive the signal to exit at their quiescent points and shut down, and for the PLATFORM to also receive the signal and pause. At .11s, both applications restart, begin pulling from the NIB, and begin performing computations. At 1.67s and 1.68s respectively, the ROUTING and TOPOLOGY applications send their newly computed rules to the PLATFORM. The PLATFORM holds on to the rules until it ensures it has received the rules from both apps, and then PLATFORM pushes both sets of rules to the switches and unpauses.

This entire process takes 1.70s, with most of the time taken by simply restarting the application (as would be required in the simple case anyway). In general, the amount of time to update multiple applications safely will vary based on number of applications, the amount of state to restore, and the type computations to be performed to generate the rules, but the overhead (compared to a restart) seems acceptable.

4.4.3 Load Balancer

Figure 4.6 shows the effect of updating a load-balancer that maps incoming connections to a set of server replicas. For this experiment, in addition to the simple restart and Morpheus experiments, we also report the behavior of record-and-replay which consists of recording the packet-in events and replaying them after restart. After 40 seconds, we bring an additional server online and update the application to also map connections to this server. To avoid disconnecting clients, existing connections should not be moved after the update.

As shown in the figure, both simple restart and record-and-replay cause disconnections, whereas state transfer causes no disruptions, since the state is preserved. As discussed in Section 4.1.2, replaying the recorded packet-ins will cause the three connections to be evenly distributed across the three servers. Similarly, for the simple restart, the connections will be evenly distributed when the clients attempt to reconnect. Therefore, one connection is erroneously mapped to the new server mid-stream, which terminates the connection.

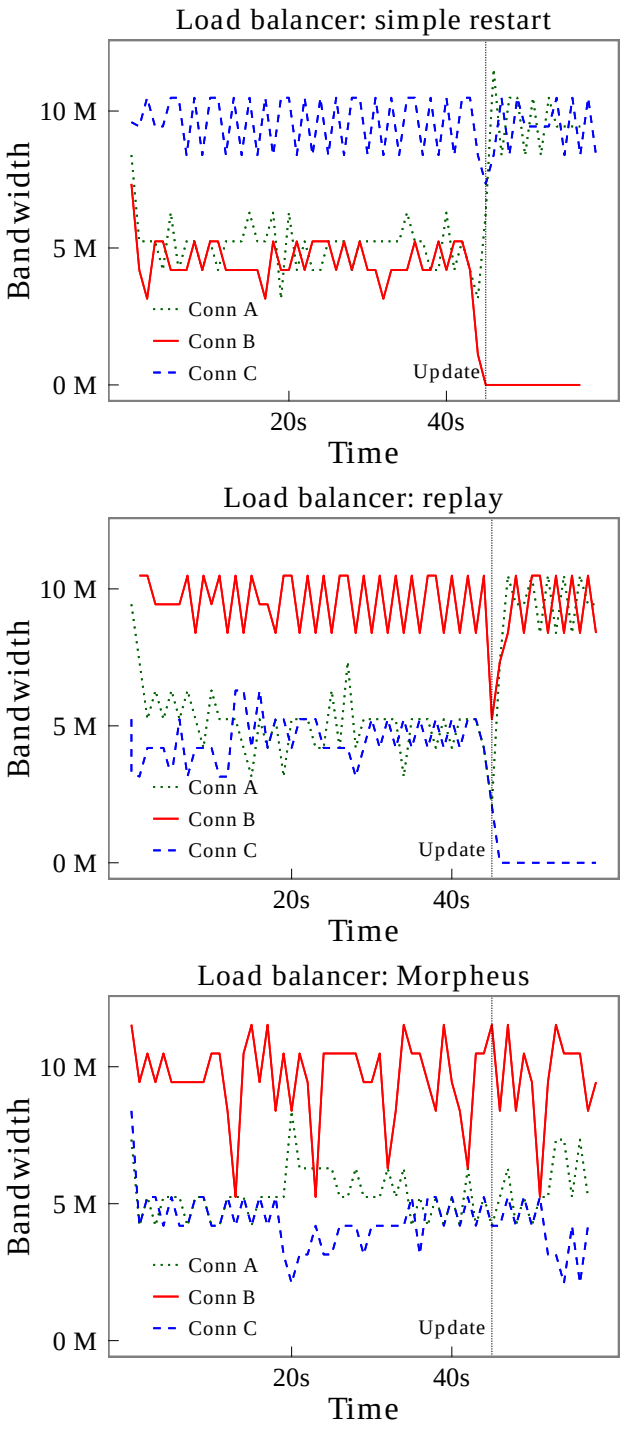

Figure 4.6: Load balancer results

4.4.4 Programmer effort

Starting from a Morpheus application/service, there are two main additional tasks required to enable dynamic update: writing code to quiesce an application prior to an update, and writing a μ transformer function to change the state. In this subsection we discuss both tasks, showing that both are straightforward.

Quiescence: The application developer must write code to check for notifications from the NIB that an update is available, and if so to complete any outstanding tasks and gracefully exit. These tasks would include storing any additional state in the NIB and/or notifying external parties. For all of our examples, this work was quite simple, amounting to 8 lines of code.

Transforming the state: Writing the function μ to transform the state was also straightforward. For FIREWALL, as described in Section 4.3.2, we wrote 4 lines of DSL code to initialize new fields to desired values so that the fields could be read with the correct data. Similarly for our applications TOPOLOGY and ROUTING, as described in Section 4.3.3, we wrote 3 lines of DSL code to initialize the weight field to a default value. For the LOAD BALANCER, no μ function was necessary, as no state was transformed, only directly transferred to the new version of the program.

We also looked at the application histories of other controllers to get a sense of how involved writing a μ function might be for updates that occur "in the wild." In particular, we looked at GitHub commits from 2012–2014 for OpenDaylight [3] and POX [4] applications. We examined applications such as a host tracker, a topology manager, a Dijkstra router, an L2 learning switch, a NAT, and a MAC blocker.

Several of the application changes consisted only of updates to the application logic, such as multiple changes to POX's IP load balancer in 2013. For them, no μ would be necessary. We also found that many of the application changes involved adding state, or making small changes to existing state. For example, an update to OpenDaylight's host tracker on November 18, 2013 converted the representation of an `InetAddress` to a `IHostId` to allow for more flexibility and to store some additional state such as the data layer address. To create this update, the administrator would write μ to initialize the data layer address for all stored hosts, if known, or add some dummy value to indicate that the data layer address was not known. An update to POX's host tracker on June 2, 2013 added two booleans to the state to indicate if the host tracker should install flows or should suppress ARP replies. To create this update, the administrator would write μ to initialize these to `True` in the NIB. To sum up, while the size of μ scales with the size of the change in state being made, in practice, we found that the effort to write μ is minimal.

4.5 Related Work

Morpheus represents the first general-purpose solution to the problem of dynamically updating SDN controllers (and by extension, updating the networks they manage). We argued this point extensively in Section 4.1, specifically comparing to alternative techniques involving controller restarts and record and replay (exemplified by the HotSwap system [116]). In this section we provide comparison to other work that provides some solution to the dynamic update problem.

Graceful control-plane updates: Several previous works have looked at the problem of updating control-plane software. In-Service Software Upgrades (ISSU) [23, 53] minimize control-plane downtime in high-end routers upon an OS upgrade by installing the new control software in parallel with the old one, on different blade and synchronizing the state automatically. Other research proposals go even further and allow other routers to respond correctly to topology changes that affect packet forwarding, while waiting for a peer to restart its control plane [102, 103]. In general, most routing protocols have mechanisms to rebuild their state when the control software (re)starts (cf. [70, 91]), e.g., by querying the state of neighboring routers.

The key difference between these works and Morpheus is that Morpheus aims to support unanticipated, semantic changes to control-plane software, possibly necessitating a change in state representation, whereas ISSU and normal routing protocols cannot.[5] In addition, Morpheus is general-purpose (due to its focus on SDN), and not tied to a specific protocol design (e.g., a routing protocol).

Distributed Controllers: Distributed SDN controller architectures such as Onix [58], Hyperflow [114], ONOS [18] or Ravana [55] can create new controller instances and synchronize state among them using a consistent store. Morpheus's distributed design is inspired by the design of these controllers, which aim to provide scalability, fault-tolerance and reliability, and can support simple updates in which the shared state is unchanged between versions (and/or is backward compatible).

[5]Cisco only supports ISSU between releases within a rolling 18-month window [23]. Outside of this window, a hard-reset of the control-plane has to be done.

However, to the best of our knowledge these systems have not looked closely at the controller upgrade problem when (parts of) the control program itself must be upgraded in a semantics-changing manner, especially when the new controller may use different data structures and algorithms than the old one. Morpheus handles this situation using the update protocol defined in Section 4.3, which quiesces the controller instances, initiates a transformation of the shared store's data according to the programmer's specification (if needed), and then starts the new controller versions. We believe this same approach could be applied to these distributed controllers as well.

Dynamic Software Upgrades: The approach we take in Morpheus is inspired by DSU discussed in the previous chapters. Most prior DSU work has focused on updating a running process ("bringing the new code to the old (but transformed) data") whereas for Morpheus the same effect is achieved by starting a new process with the relevant state ("bringing the old (but transformed) data to the new code"). While prior DSU work has considered the problem updating network software generally [98] (including for "active" networks [47]), ours is the first to apply a general-purpose solution to (distributed) software-defined network controllers in particular.

4.6 Future Work

Discussing and implementing Morpheus generated several avenues of additional ideas for follow-on work. This section presents some of those ideas.

4.6.1 Rollback in SDN controller updates

One common question that we receive when discussing this research is about the possibility of "rolling back" an update if the network starts behaving incorrectly after an update has taken place. Even though an update may be tested with test data by staging a dry run of the update on a set of test controllers, in practice, updates may result in unexpected or incorrect behavior. Having the ability to rollback a dynamic update in the case of problems would give network administrators more confidence in choosing to use dynamic updates, and also add more confidence in general about the entire update process.

One way to view rollback is as a reverse upgrade, essentially "upgrading" backwards to the previous version. This operation would involve transforming the state backward, reversing whatever transformation function was applied. In the case of rolling back mid-update, we would need to track which components had been upgraded and which components had not yet been modified, to avoid unnecessarily applying the reverse transformation. We would also need to research how to best ensure correctness in the rollback, and to clearly define correctness guarantees, similar to the correctness discussed in the next section.

4.6.2 Correctness in SDN controller updates

Another research challenge we could address is how to best determine whether or not our update is correct in our SDN controller. We must first define what "correct" means in our updates. During an update, our goal is to minimize disruption to

the network. Some behavior is obviously wrong and a violation of correctness, such as dropping packets or introducing routing loops. But some updates may be more specific, such as the case when an application is implementing load balancing and an update switches the configuration from two servers to three. If the new program version should be balancing traffic between three servers but still is only using two servers after update, then this behavior violates correctness although traffic may not be noticeably disrupted.

Future work in this area would involve investigating the best method for specifying what a program is supposed to do to define correctness. When a new controller version implements the same semantics as the old version (but perhaps just organizes its internal state differently), then the semantics of the update are clear. But in practice, controller upgrades are going to change the semantics in some way. Thus an important question is how these changes should affect new *and existing* flows.

One potential approach is to take inspiration from previous work on specifying and verifying properties of general-purpose dynamic updates [45]. We could potentially define correctness with respect to a set of NetKAT properties [9], which define what the network does by relating ingress and egress flows or individual events. In future work, we could explore other sorts of correctness notions, based on properties for which the behavior of the old and new versions is not identical but can be related, and based on properties of new features that should only apply to flows or events after the update.

4.7 Conclusion

This chapter has proposed *dynamic update by state transfer* as a general-purpose approach to dynamically update software-defined network controllers. The approach works by providing direct access to the relevant state in the running controller, and initializing the new controller's state as function of the existing state. This approach is in contrast to alternatives that attempt to automatically reproduce the relevant state, but may not always succeed. We implemented the approach as part of Morpheus, a new SDN controller whose design is inspired by industrial-style controllers. Morpheus provides means to specify transformations in a persistent store, and employs an update coordination protocol to safely deploy the transformation. Experiments with Morpheus show that dynamic update by state transfer is both natural and effective: it supports seamless updates to live networks at low overhead and little programmer effort, while prior approaches would result in disruption, incorrect behavior, or both.

Author Contributions: This work is the result of a collaboration between many individuals, resulting in a paper [93] which appears in this chapter. All authors contributed to the overall design and ideas behind this work. Arjun Guha and Joseph Collard wrote all of the test applications (Firewall, Routing and Topology, and the Load Balancer) and helped perform the experiments. Nate Foster, Michael Hicks, and Laurent Vanbever contributed greatly to the writing. I designed the NIB (Network Information Base) and integrated the applications to work with it, wrote

the update DSL and all other aspects related to updating the applications, designed and implemented the update coordinator, contributed to the writing, and helped with the experiments.

Chapter 5: Conclusion

In this dissertation, I have argued that it is possible to build general-purpose frameworks for efficient, on-line data transformation in support of flexible system services, especially dynamic upgrades. I showed that abstracting some of the ideas from DSU work enables the principles to be applicable to a wider variety of systems due to several generalizations. I generalized DSU's notion of in-memory state transformation, allowing broader methods of data transformation and broader types of data that may be transformed. I also built on the concept of state transformation in a process' memory to include transformation of data not necessarily stored in memory, and for services other than DSU.

I presented frameworks that are flexible across a variety of programs and argued that these frameworks can be used transform data in an efficient way that maximizes program availability. Specifically, I first presented C-strider, a generic framework that allows users to customize a heap traversal to perform a specific and reusable service, with both parallel and single-threaded traversals. Next, I presented KVolve, a system that lazily applies on-line data transformations to data stored in Redis, a NoSQL database, allowing us to perform these transformations without impacting Redis' availability. Finally, I presented Morpheus, an approach to dy-

namically update SDN controllers. Morpheus demonstrates the ability to update multiple controller applications without loss of availability and presents a framework for updating the state shared across the applications.

Appendix A: Py-KVolve Details

This appendix presents the details of Py-KVolve, described in Section 3.4. It presents additional details of the overall design of Py-KVolve as it differs from KVolve, provides the DSL (Domain-Specific Language) details (which we used in Morpheus), and provides some additional experimental results.

A.1 Specifying Updates

This section explains how programmers can express database updates using Py-KVolve's DSL, first introduced in Section 3.4.1. We begin by defining the language's syntax and semantics, and then present a series of examples. We also sketch how we compile DSL programs into Python code, using during lazy data migrations.

A.1.1 DSL syntax and semantics

An update specification consists of a sequence of DSL commands, which are processed in order. There are two basic kinds of commands. The first specifies changes to JSON values, and the second generalizes the first, also allowing changes to namespaces.

Directive	**Path**	**Action code**	**Must Return**
INIT	[json path, or empty for entire value]	Yes. (Assign value to `$out`.)	None
UPD	[json path, or empty for entire value]	Yes. (Assign value to `$out`.)	None
DEL	[json path, or empty for entire value]	Yes. (Code to determine what to delete.)	Bool
REN	[json path] → [json path]	No	None

Table A.1: The DSL directives

for *keyglob* $vers_{old} \rightarrow vers_{new}$ {

DIRECTIVE path action-code

...

}

(a) Command to update JSON values

for namespace $ns_{old} \rightarrow ns_{new}$ $vers_{old} \rightarrow vers_{new}$ {

optional-DIRECTIVES

};

(b) Command to update namespaces

Figure A.1: DSL commands

Updating values: The format of commands for updating values is given in Figure A.1a. The *keyglob* is a simplified regular expression (allowing only '*', '?' and '[range]') which identifies the keys of affected values. For example, the keyglob order:may20* might indicate only a range of purchase order invoice numbers are affected. The $vers_{old}$ and $vers_{new}$ strings identify the affected versions of the data.

The body of the **for** contains one or more commands, each beginning with a directive, which is the action to be performed. There are four possibilities, shown in Table A.1: initializing a new field, deleting a field, updating the value contained in a field, or renaming the field. The *path* is an index into the JSON object, such as ['order', ['orderItems'], 'discountedPrice'] as shown on line 2 of Figure 3.9a. In

Token	Meaning
$out	the value of the path inside the [] in the Directive *(this is the value to be updated, initialized, deleted, etc)*
$in	the original value stored in the key (same as $out, but not written to)
$root	the root of the JSON structure.
$base	the same JSON structure, used to address siblings
$dbkey	the database key name currently being processed

Table A.2: The DSL convenience tokens

the REN case the path part contains both a source and destination path.

The *action-code* of the command differs per directive, as shown in the third column of Table A.1. This code consists of a mix of special DSL tokens and Python code. The INIT and UPD directives are similar in that they both must specify the value that should be initialized or renamed. The DEL requires the code to return true or false, indicating whether a given path should, indeed, be deleted. Table A.2 shows the DSL tokens that may appear in action code, which are interpreted specially by our DSL compiler. All of the directives' action code may use any of the tokens, except INIT which may not use $in because an existing value does not exist for an initialized value.

Updating namespaces: As mentioned in Section 3.1.4, another type of update involves changing namespaces. The second DSL command type generalizes the first,

defining a change to the key namespace in addition to any changes to the values of the affected keys. This directive is particularly useful when the DSL writer wishes to change the prefix at the beginning of all of some set of keys. For example, a DSL writer may wish to rename namespace order: to be order:2015: for easier tracking. Any directives in the body of the **for** will apply to the values of any matched keys.

A.1.2 Example updates

Here we present two example updates: one to update JSON fields, the other to update namespaces.

A.1.2.1 Manipulating JSON fields

The purchase order example in Section 3.1.2 showed how to add and rename JSON fields. Building on that example, suppose the next version (v2) expects the contents of field _id to be a decimal, rather than hexadecimal. On line 4 of Figure A.2a, the DSL writer uses $in to access the field's current value, and the action code converts it from hexadecimal, assigning the result to $out. Also suppose the next version avoids using the field since when it is later than orderdate. On line 7 of Figure A.2a, the DSL writer uses the $base variable to access fields higher up in the JSON nesting structure than the field 'since', determining whether the field should be deleted.

```
for key:* v1→v2 {
  UPD ["_id"] {
    if any(c.isalpha() for c in $in):
        $out=int($in, 16) }
  DEL ["since"] {
    if (datetime.strptime($out, "%d/%m/%Y") >
      datetime.strptime($base["order"]["orderdate"], "%d/%m/%Y")):
        return True
    else:
        return False }
};
```

(a) Additional purchase order update (extending Fig. 3.9a).

```
for namespace
  amico:following → amico:following:default v1.2→v2.0;
for namespace
  amico:reciprocated → amico:reciprocated:default v1.2→v2.0;
```

(b) Amico namespace change.

Figure A.2: Example updates

A.1.2.2 Renaming the database keys

As mentioned in Section 3.3.3, Amico [7], a Redis-backed social network, added a *scope* suffix to certain namespaces between version 1.2 and 2.0. In version 1.2, namespaces were of the format amico:following and amico:reciprocated, whereas in version 2.0, they would have forms like amico:reciprocated:*scope*. For backward compatibility, they defined a default scope, default. As such, we can use our DSL to upgrade existing data to use this default scope, as shown in Figure A.2b.

A.1.3 Implementing data transformations

The DSL code is translated into Python functions. In some cases multiple directives are combined into a single function, and in other cases there is one function per directive. These functions have the form shown in Figure 3.9b: they take the current key and value and return a potentially updated key and value. The body of each function is essentially the action code provided in the DSL specification with special variables (like `$out`) replaced by the appropriate accessors. For example, in Figure 3.9b the `$base` variable is replaced with Python variable `e`, and it is then used in the accessor for `$out`, which is the Python code `f['discountedPrice']`. Multiple directives in the same **for** block will apply to the same values, so sometimes we find it efficacious to generate a single function for multiple directives, e.g., when the paths are similar (as is the case in Figure 3.9b). Otherwise we will produce multiple functions but sequence their execution in the `GET` wrapper (Section 3.4.2) in the order in the given file. Our implementation borrows some code from JSON-delta [84].

The **for** `namespace` transformations potentially modify the Redis keys as well. This is largely straightforward, and we leave out further details for lack of space.

A.2 Details of Implementing Lazy Updates in Py-KVolve

This section describes our Py-KVolve implementation in detail. We start by explaining how we install a dynamic update. Then we describe additional information about Py-KVolve's implementation of the `GET` and `SET` operations (that were

first presented in Section 3.4.2). Finally we discuss other commands.

A.2.1 Installing an update

Py-KVolve provides a special "update client" application for initiating an update. The developer invokes it by pointing it at the running Redis and passing the DSL file. Redis stores the current logical version of the data for each namespace *ns* under the special key __VERSION_*ns*. It also stores the compiled DSL (a Python module) and metadata for prior transformations, for each namespace. When the update client connects, it confirms that it is consistent with the existing versions, i.e., it is upgrading from the current logical version to the new one, for the affected namespaces. The update client uses a lightweight transaction to atomically update the logical versions of the namespaces, add the new DSL and metadata, and set a flag __UPDATE_IN_PROGRESS to true.

Next, it kills any clients that are now out of date with any logical namespace version. To do this, it uses the Redis command CLIENT LIST, which provides a list of all connected clients, then it uses the command CLIENT KILL to direct Redis to kill those clients that are potentially using out-of-date data. It identifies such clients by looking at the names of the clients in the list, which are set to include this version information when the clients connect. Once all clients have been killed, the __UPDATE_IN_PROGRESS flag is set to false.

At this point, it is safe for clients to connect. When they do, they will first check that they match the versions of data they are interested in, and then wait for

the __UPDATE_IN_PROGRESS flag to be unset, if it is set. Clients that attempt to connect but are outdated will be refused, receiving a DeprecationWarning.

A.2.2 Additional information about GET operations

Making misses more efficient: The GET algorithm described in Section 3.4.2 works best when queries succeed; determining that a key is definitively not in the database requires two lookups: the one at the newest version (which fails), and then the MGET at all versions. We could forego the first lookup in favor of always doing the second, but we find the second call is much more expensive, especially when there is a large number of possible previous versions.

As an optimization, when the wrapper discovers it is in case (a), it sets the queried key (at the current version) to have a *sentinel value* that indicates the key is morally absent. Then, the wrapper will identify this sentinel on subsequent GETs and return None. This approach trades space for time: storing the extra mapping ensures we have to perform only one lookup for each GET call, rather than two. Informal experiments found that this optimization could cut overhead by a third for workloads in which misses were more common.

Handling namespace changes: We simplified our description above by assuming that all versions of a key k are the same, aside from the prepended version identifier. But a key's namespace can evolve according **for** namespace DSL commands (see Figure A.1b). As such, the generated list of possible keys for the first MGET might need to be something like ['v2|newns:123', 'v1|ns:123', 'v0|ns:123']. To

generate a list like this, we essentially run namespace transformations backwards. In particular, to transform the namespace from v1 to v2 in the above list, we would have had to have had a command:

for namespace ns → newns v1 → v2;

The call GET newns:123 from the application would cause the wrapper to query the key v2|newns:123. When this fails, we run the above transformation backwards, to produce key v1|ns:123. The last key in the above list, v0|ns:123, would arise assuming that no namespace was transformed between versions v0 and v1.

A.2.3 Additional information about SET operations

The SET operation, first described in Section 3.4.2, can also be passed flags that modify its behavior. Flag NX instructs SET to only set k to v if k is not already in the database. Our wrapper must therefore check *all versions* are absent before performing this command. Flag XX only performs the set when k is already present; it is handled similarly. Finally, flags EX and PX set an expiration timer for a key-value pair. Supporting this flag follows the basic logic for SET, but requires carrying forward the timeout during migration. The GET wrapper does this by querying the current time-to-live and setting it on the new value when it is migrated.

A.2.4 Other commands

Redis supports many more commands, but our Py-KVolve implementation focused on GETs and SETs, as they are the core feature of applications that use Redis,

in our experience.[1] Supporting most other commands would be straightforward. For example EXIST checks the presence of a key—our wrapper would treat this similarly to GET. Or, APPEND appends to an existing mapped-to value; this would be handled similarly to SET XX. However, we chose to leave full implementation to the standard (non-client) version of KVolve.

A.3 Py-KVolve Update Overhead

This section considers the performance impact of Py-KVolve during an update. (Additional experimental results were presented in Section 3.4.3.) We find that Py-KVolve significantly reduces the pause perceived by clients during an update, compared to a stop-the-world data migration. All experiments were performed on a computer with 24 processors (Intel(R) Xeon(R) CPU E5-2430 0 @ 2.20GHz) and 32 GB RAM, and run with PyPy 2.0.2 with GCC 4.4.7 on Red Hat 4.4.7-3. All tests report the median of 11 trials.

A.3.1 Update pause time using Py-KVolve

To demonstrate update pause time, we performed updates for JSON entries similar the example from Section 3.1.2 using the DSL shown in Figure 3.9a. At start-up, 50 clients connect and begin sending commands to Redis as fast as possible. The commands consist of 50% gets and 50% sets chosen at random. For this experiment, we start with a pre-populated database of 200,000 keys. After 20 seconds, a separate

[1] We also support DEL, which deletes keys (we try to delete the current version, and if no key exists, delete all versions, for good measure).

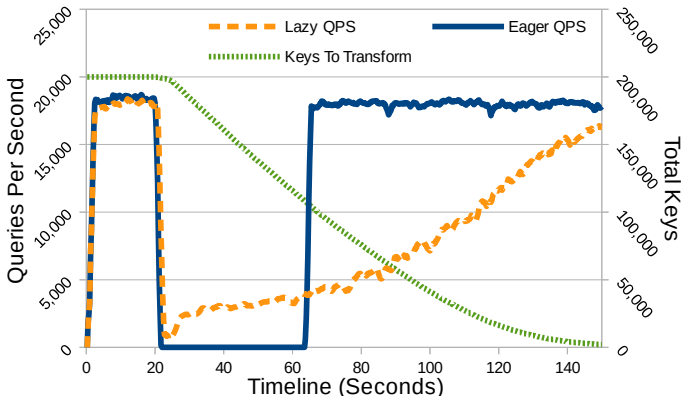

Figure A.3: Lazy vs. eager updates for gets and sets over the full range of 200,000 keys

thread requests an update, and all 50 clients are subsequently disconnected by Redis. Then they all reconnect, specifying the new data version, and proceed with further queries, which will update the data lazily. We compare this against an "Eager" case that simulates stop-the-world updates: rather than reconnect immediately, the clients wait for another thread to signal them after completely migrating the database; only then do they reconnect.

Figure A.3 shows the queries per second (QPS) across a timeline for the experiment, where the clients access all 200,000 keys uniformly at random. At 20 seconds, the QPS for the Py-KVolve case (dashed line) dips briefly when an update thread requests an update, and all 50 clients disconnect and reconnect at the new version. Then the clients begin lazily updating the keys at a degraded rate as the keys are brought up to the current version. The old-version keys are shown in the finely-dashed line, and this number decreases at a constant rate as the keys are uniformly accessed lazily.

The Eager experiment (solid line) shows that all clients are stopped at 20

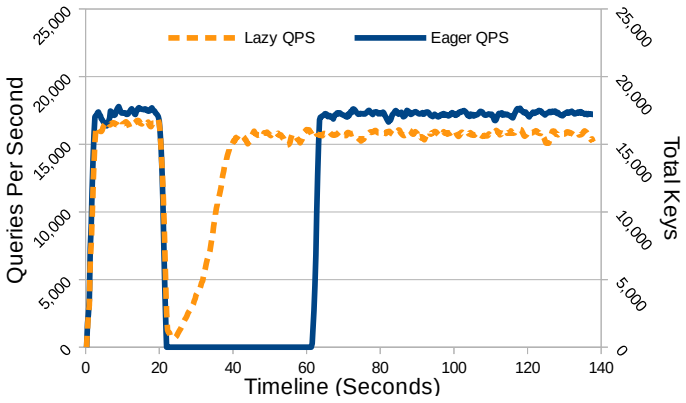

Figure A.4: Lazy vs. eager updates for gets and sets over a 20,000 key subset in a 200,000 key database

seconds at the beginning of the update. After about 40 seconds, the transformation is complete and the clients reconnect when they are signaled by the transformation thread at around 60 seconds on the timeline. Note that this pause time varies widely across runs. The pause time of 40 seconds shown in Figure A.3 was the median of 11 trials, with a SIQR of 18.75 seconds. We think this variation in transforming the Redis database may be due to Python or PyPy, but we are still investigating the exact cause of variation of pause time.

Note that in this experiment, all clients access *all* keys in the entire database in a uniform pattern. This is the worst-case scenario for lazy updating, as opposed to the scenario where a smaller set of keys is constantly accessed and the rest of the database is rarely accessed. Lazy updates benefit the most from the later situation, whereas the Eager case suffers the most from that situation, having to transform old data that is rarely (or never) accessed.

Figure A.4 shows the same experiment as above, except this time, the clients query only subset of 20,000 keys, representing a "hot" set of keys that get queried

repeatedly. This setup allows Py-KVolve to fully capitalize on lazy updates. Now, the Py-KVolve case (dashed line) is able to very quickly return to full speed of execution and the clients experience minimal degradation, whereas the Eager case must still migrate the entire database, as less-frequently used keys might still need to be accessed.

These experiments show that data access patterns and number of database keys can alter the performance of Py-KVolve and lazy updates. In general, we demonstrated that in some cases Py-KVolve can greatly reduce pause time during updates with minimal performance penalty.

Bibliography

[1] Floodlight. http://floodlight.openflowhub.org/.

[2] MongoDB. http://www.mongodb.com/.

[3] OpenDaylight. http://www.opendaylight.org.

[4] Pox. http://www.noxrepo.org/pox/about-pox/.

[5] Adobe. Disable automatic check for updates. https://helpx.adobe.com/creative-suite/kb/disable-automatic-check-updates-cs3.html, 2012.

[6] E. E. Aftandilian and S. Z. Guyer. Gc assertions: Using the garbage collector to check heap properties. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 235–244, New York, NY, USA, 2009. ACM.

[7] Agora Games. Relationships (e.g. friendships) backed by redis. https://github.com/agoragames/amico.

[8] Y. An and T. Topaloglou. Semantic web, ontologies and databases. chapter Maintaining Semantic Mappings Between Database Schemas and Ontologies, pages 138–152. Springer-Verlag, Berlin, Heidelberg, 2008.

[9] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *POPL*, 2014.

[10] Apache. Apache avro. http://avro.apache.org/.

[11] Apache. The apache cassandra project. http://cassandra.apache.org.

[12] Apache. Apache thrift. http://thrift.apache.org/.

[13] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 187–198, New York, NY, USA, 2009. ACM.

[14] M. Arnold, M. Vechev, and E. Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. pages 143–162, 2008.

[15] M. Asay. Nosql databases eat into the relational database market. http://www.techrepublic.com/article/nosql-databases-eat-into-the-relational-database-market/, 2015.

[16] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[17] D. Bellot. Painless data migrations with schema-less nosql data stores and redis. https://github.com/ServiceStack/ServiceStack.Redis/wiki/MigrationsUsingSchemalessNoSql, 2011.

[18] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an open, distributed SDN OS. In *HotSDN*, pages 1–6, 2014.

[19] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[20] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, Sept. 1988.

[21] M. Callaghan. Online schema change for mysql. https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932, 2010.

[22] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew. Dynamic software updating using a relaxed consistency model. *IEEE Transactions on Software Engineering*, 37(5):679–694, 2011.

[23] Cisco Systems. Cisco IOS In-Service Software Upgrade. `http://www.cisco.com/c/dam/en/us/products/collateral/ios-nx-os-software/high-availability/prod_qas0900aecd8044c333.pdf`, 2007.

[24] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] W. W. W. Consortium. Extensible markup language (xml). `http://www.w3.org/XML/`.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[27] N. Cozma. Stop applications from updating automatically in windows. `http://www.cnet.com/how-to/stop-applications-from-updating-automatically-in-windows/`, 2012.

[28] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22(1):73–98, 2013.

[29] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 176–185, New York, NY, USA, 2005. ACM.

[30] A. Deshpande and M. Hicks. Toward on-line schema evolution for non-stop systems. Presented at the 11th High Performance Transaction Systems Workshop, September 2005.

[31] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 18:1–18:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[32] T. Dumitras, P. Narasimhan, and E. Tilevich. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. pages 865–876, Reno/Tahoe, NV, Oct 2010.

[33] A. Ferrari, S. J. Chapin, and A. Grimshaw. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing*, 3(2):63–73, Apr. 2000.

[34] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[35] H. Freyther. memprof. `http://www.secretlabs.de/projects/memprof/`.

[36] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[37] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 279–292, 2013.

[38] Google. Google app engine: Platform as a service. https://cloud.google.com/appengine/docs.

[39] Google. Protocol buffers. https://developers.google.com/protocol-buffers/.

[40] Google. Container engine: Rolling updates. `https://cloud.google.com/container-engine/docs/rolling-updates?hl=en`, 2015.

[41] Google Developer Tools. Heap profiler. `http://google-perftools.googlecode.com/svn/trunk/doc/heapprofile.html`.

[42] G. R. Guide. Market guide for in-memory dbms. http://www1.memsql.com/gartner.html, 2014.

[43] A. Gundry. Coping with change: data schema migration in haskell. http://cufp.org/2015/coping-with-change-data-schema-migration-in-haskell.html, 2015.

[44] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[45] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In *International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2012.

[46] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. *ACM Trans. Program. Lang. Syst.*, 36(4):13:1–13:38, Oct. 2014.

[47] M. Hicks and S. Nettles. Active networking means evolution (or enhanced extensibility required). In H. Yashuda, editor, *International Working Conference on Active Networks (IWAN)*, volume 1942 of *Lecture Notes in Computer Science*, pages 16–32. Springer-Verlag, October 2000.

[48] M. Hicks and S. Nettles. Dynamic software updating. *ACM TOPLAS*, 27(6):1049–1096, 2005.

[49] R. Hinze and A. Löh. Generic programming, now! In *Proceedings of the 2006 International Conference on Datatype-generic Programming*, SSDGP'06, pages 150–208. Springer-Verlag, Berlin, Heidelberg, 2007.

[50] E. International. The json data interchange format. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[51] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. pages 275–288, 2002.

[52] M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 119–128, New York, NY, USA, 2009. ACM.

[53] Juniper Networks. Juniper Networks. Unified ISSU Concepts. `http://tinyurl.com/9wbjzhy`.

[54] J. Katcher. Postmark. Technical Report TR 3022, Network Appliance, 1997.

[55] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *SOSR*, 2015.

[56] S. Kemp. Redisfs. http://www.steve.org.uk/Software/redisfs/.

[57] M. Kleppmann. Schema evolution in avro, protocol buffers and thrift. http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html, 2013.

[58] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*. USENIX Association, Oct. 2010.

[59] P. Lehtinen. Jansson. htp://www.digip.org/jansson/.

[60] R. Lemos. Heartbleed is the gift that keeps on giving as servers remain unpatched. http://arstechnica.com/security/2014/08/heartbleed-is-the-gift-that-keeps-on-giving-as-servers-remain-unpatched/, 2014.

[61] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. https://snap.stanford.edu/data/soc-LiveJournal1.html, June 2014.

[62] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[63] J. Løland and S. Hvasshovd. Online, non-blocking relational schema changes. In Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2006.

[64] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *SIGARCH Comput. Archit. News*, 32(5):211–223, Oct. 2004.

[65] K. Makris and R. A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 31–31, Berkeley, CA, USA, 2009. USENIX Association.

[66] A. McCurdy. Redis-py: Redis python client. https://github.com/andymccurdy/redis-py.

[67] J. Medina. *Nosql 133 Success Secrets - 133 Most Asked Questions on Nosql - What You Need to Know*. Emereo Pty Limited, 2014.

[68] J. Mihalicza, Z. Porkoláb, and A. Gabor. Type-preserving heap profiler for C++. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 457–466, Washington, DC, USA, 2011. IEEE Computer Society.

[69] P. Mosendz. Facebook's overnight outage cost them half a million dollars. http://www.thewire.com/technology/2014/06/the-overnight-facebook-outage-cost-them-half-a-million/373089/, 2014.

[70] J. Moy, P. Pillay-Esnault, and A. Lindem. Graceful OSPF Restart. RFC 3623, 2003.

[71] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM.

[72] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[73] S. Noach. openark kit. https://code.google.com/p/openarkkit/.

[74] OSXDaily. Stop software update from bugging you in mac os x. http://osxdaily.com/2012/11/15/stop-software-update-mac-os-x/, 2012.

[75] M. Payer and T. R. Gross. Hot-patching a web server: A case study of asap code repair. In *PST*, pages 143–150, 2013.

[76] Percona. pt-online-schema-change. http://www.percona.com/doc/percona-toolkit/2.1/pt-online-schema-change.html, 2013.

[77] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM.

[78] L. Pina, L. Veiga, and M. W. Hicks. Rubah: DSU for java on a stock JVM. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 103–119. ACM, 2014.

[79] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, asynchronous schema change in F1. In *VLDB*, 2013.

[80] E. Rahm. Towards large-scale schema and ontology matching. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, chapter 1, pages 3–27. Springer, Heidelberg, 2011.

[81] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the gc compute efficiently?: A language for heap assertions at gc time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 256–269, New York, NY, USA, 2010. ACM.

[82] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[83] D. Rethans. Managing schema changes with mongodb. http://derickrethans.nl/managing-schema-changes.html, 2013.

[84] P. J. Roberts. Json-delta: a diff/patch pair for json-serialized data structures. `http://phil-roberts.name/json_delta/`.

[85] E. Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002.

[86] M. Ronström. On-line schema update for a telecom database. In *ICDE*, pages 329–338, 2000.

[87] P. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012.

[88] Salesforce. Maintenance calendar. `http://trust.salesforce.com/trust/calendar`.

[89] S. Sanfilippo. Redis. http://redis.io/.

[90] S. Sanfilippo. Finally redis collections are iterable. http://antirez.com/news/63, 2013.

[91] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. Graceful Restart Mechanism for BGP. RFC 4724, Jan. 2007.

[92] K. Saur. C-strider. `https://github.com/plum-umd/c-strider`.

[93] K. Saur, J. Collard, N. Foster, A. Guha, L. Vanbever, and M. Hicks. Morpheus: Safe and flexible dynamic updates for SDNs, Mar. 2015.

[94] K. Saur, T. Dumitras, and M. W. Hicks. Evolving nosql databases without downtime. http://arxiv.org/abs/1506.08800v2, 2015.

[95] K. Saur, T. Dumitras, and M. W. Hicks. Evolving nosql databases without downtime (original version). http://arxiv.org/abs/1506.08800v1, 2015.

[96] K. Saur, M. Hicks, and J. S. Foster. C-strider: type-aware heap traversal for C. *Software: Practice and Experience*, 10.1002/spe.2332, 2015.

[97] S. Scherzinger, M. Klettke, and U. Störl. Managing schema evolution in nosql data stores. In *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013)*, 2013.

[98] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, pages 53–65, March 1993.

[99] Serious Business, M. Freels, R. Ravi, and R. Olson. Tablemigrator. https://github.com/freels/table_migrator, 2009.

[100] A. W. Services. Amazon ec2 maintenance help page. `https://aws.amazon.com/maintenance-help/`.

[101] A. W. Services. Aws elastic beanstalk announces rolling updates. `https://aws.amazon.com/about-aws/whats-new/2013/11/11/aws-elastic-beanstalk-announces-rolling-updates/`, 2013.

[102] A. Shaikh, R. Dube, and A. Varma. Avoiding instability during graceful shutdown of OSPF. In *INFOCOM*, 2002.

[103] A. Shaikh, R. Dube, and A. Varma. Avoiding instability during graceful shutdown of multiple OSPF routers. *IEEE/ACM Transactions on Networking*, 14(3):532 –542, june 2006.

[104] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Macciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback recovery for middleboxes. In *SIGCOMM*, 2015.

[105] Skype. Survey finds nearly half of consumers fail to upgrade software regularly and one quarter of consumers don't know why to update software. `http://about.skype.com/press/2012/07/survey_finds_nearly_half_fail_to_upgrade.html`, 2012.

[106] solid IT. Db-engines ranking. http://db-engines.com/en/ranking.

[107] C. Soules, J. Appavoo, K. Hui, D. D. Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. 2003.

[108] SoundCloud. Large hadron migrator. https://github.com/soundcloud/lhm, 2011.

[109] stackoverflow.com. Are there any tools for schema migration for nosql databases? http://stackoverflow.com/questions/1961013/are-there-any-tools-for-schema-migration-for-nosql-databases.

[110] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[111] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: A VM-centric approach. In *PLDI*, 2009.

[112] M. Szeredi. Fuse. http://fuse.sourceforge.net/.

[113] J. Talty. Did twitter's outage cost the economy billions? `http://www.ibtimes.com/did-twitters-outage-cost-economy-billions-731964`, 2012.

[114] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[115] valgrind.org. massif. `http://valgrind.org/info/tools.html`.

[116] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. Hotswap: Correct and efficient controller upgrades for software-defined networks. In *HotSDN*, 2013.

[117] M. Vechev, E. Yahav, and G. Yorsh. Phalanx: Parallel checking of expressive heap assertions. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 41–50, New York, NY, USA, 2010. ACM.

[118] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 584–595. VLDB Endowment, 2003.

[119] L. Walker. Fcc report details major 911 outage as america updates emergency services systems. http://www.newsweek.com/fcc-report-details-major-911-outage-america-updates-emergency-services-systems-278603, 2014.

[120] WikiMedia. Mediawiki 1.5 upgrade. http://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade, 2005.

[121] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.