WikiD

ANGUS PEARSON

4th Year Project Report
Artificial Intelligence and Computer Science

SCHOOL OF INFORMATICS

UNIVERSITY OF EDINBURGH

2017

Acknowledgements

I'd like to thank my supervisor *Paul Anderson* for providing support and a long tether for this project. Similarly, I'd like to thank the members and administrators of *The Tardis Project* for all the time they dedicate to keeping the system running (most of the time).

Also, the staff at Greggs, Forrest Road, Edinburgh, for opening on Sundays.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Angus Pearson*)

4TH YEAR PROJECT REPORT

ARTIFICIAL INTELLIGENCE AND COMPUTER SCIENCE
SCHOOL OF INFORMATICS
UNIVERSITY OF EDINBURGH

Abstract

Across both small enterprise and large, multi-datacentre deployments, systems administrators are tasked with ensuring datacentre configuration, documentation and operation are all consistent with one-another. Every, connection, cable, asset or otherwise must be documented within what is referred to as *Data Centre Infrastructure Management, Monitoring & IP Address-space Management* (*DCIM & IPAM*) tools.

As the size of a deployment grows, use of *DCIM & IPAM* tooling and automated, *Prescriptive Configuration* becomes increasingly necessary for Administrators to have a correct understanding of operations within the datacentre. However, these tools have control over only that which is configurable under them: transient effects and unconfigurable properties, insofar as software configuration, are outwith the scope of *both* prescriptive configuration languages and documentation. The modern System Administration toolkit trio is completed by the addition of Infrastructure *performance monitoring and analysis* tools.

Should inconsistencies arise between these *desired* states prescribed by *DCIM & IPAM* and the *operational*, actual state deployed to the datacentre, mistruths can easily trigger errors, wasted time, cause network problems, lead to further misconfigurations and create other issues. As such, a need arises for a *fourth* class of tool within the Systems Administration space, giving the ability to automatically verify consistency between all of these *desired* prescribed states against the *operational* state and use this knowledge to pro-actively fix misconfigurations before they cause real errors.

This work presents a methodology for and implementation of such a verification system, called 'WikiD' – probably pronounced *'Wicked'* or *'Wiki Dee'* – wherein a series of abstract models of configurational relations are automatically discovered and populated. These models represent a graph of beliefs held about devices in the system. We can then employ graph consistency checking, i.e. *Partial Homomorphism* to verify that all of these beliefs are introspectively consistent.

CONTENTS

1 *Introduction* 9

1.1 *Project Objectives: What* WikiD *is and isn't* 10

1.2 *Report Structure* 11

1.3 *Contributions* 11

2 *Background* 13

3 *Design* 17

3.1 *Modelling Configurations* 17

3.2 *Constructing Models* 19

3.3 *Consistency Checking* 20

3.3.1 *Worked Example* 21

3.4 *How Well Does The Model Fit the Real World?* 23

4 *Implementation* 25

4.1 *Implementation Aims* 26

4.2 *Architecture Overview* 26

4.3 *Microservices Framework* 27

4.3.1 *Jobs* 27

4.3.2 *Reports* 28

4.4 *The Redis NoSQL Database* 29

4.5 *A Graph Database Implementation with Redis* 30

4.6 *Alternative Graph Databases* 33

4.7 Implemented Services 34
4.8 WikiD *Default Services* 34
4.8.1 Conch 34
4.8.2 Trawler 35
4.8.3 Watchdog 36
4.9 WikiD *Service Integrations for Tardis* 36
4.9.1 DNS Discovery Service 36
4.9.2 Netbox 39
4.9.3 Switch 41
4.9.4 Sentry 41
5 Results & Evaluation 45
5.1 Evaluation of Performance on Tardis 45
5.2 Mock Consistency Check 46
5.2.1 Discussion 47
5.3 Remarks on Production Readiness 48
6 Conclusion 49
6.1 Introspection & Limitations 49
6.2 Future Work 50
6.3 Final Conclusions 50
Bibliography 51
Appendix 57

1

INTRODUCTION

Configuration of large, complex systems has remained a difficult task since the early days of the computer and networked multi-user system. A myriad of different options and architectures present many pitfalls and challenges to the humble SysAdmin.

The need for rigour in Administration should come as no news, with studies from over a decade ago directly attributing lost revenue to network downtime, which was found to be primarily caused by Human Error (62%)[1].

[1] Zeus Kerravala. As the value of enterprise networks escalates, so does the need for configuration management. *The Yankee Group*, 4, 2004

As it stands, the good practice in System Administration is to use some form of *prescriptive* configuration management and to extensively monitor and document the system. *Prescriptive Configuration* is a configuration system where devices in the deployment receive their configurations from a centralised configuration manager. We can also assert that documentation be *authoritative*, meaning it represents the *desired*, correct system state, our *base truth*.

However, configuration management systems can only span the domain of that which is software configurable, and can struggle in heterogeneous environments with many varieties of hardware, operating system requirements. Some of the most critical configurations in a datacentre that govern network behaviour, security and availability of services are buried within equipment with esoteric firmware, configurable only through a specialised interface such as *SNMP* (which seems to use the word *'Simple'* ironically) or not at all. Further to this, there are important *hardware* configurations, too – these things cannot be controlled by configuration systems, and after all, how a network fabric is laid up and interconnected is complex and *has* to be correct, else Network Cthulhu wreaks havoc. Neither of these properties can be conventionally configured with a configuration management system.

One could argue there's little actual difference whether documentation serves as the source of *'correct configuration'* or not, but it is an important distinction to make when considering our *Design Philosophy* – That documentation should serve as the source of *desired*

system state, and that the *operational* state should seek faithfully to implement this desire. With this convention, when architecting our system, we first document the intended layup, then deploy in actuality. Consequently, any inconsistencies between the *operational* state and the *desired* are automatically errors in the *operational* state.

1.1 *Project Objectives: What* WIKID *is and isn't*

The existing infrastructure Administraion triad of configuration, documentation and monitoring tools provides a strong arsenal. What WIKID and this project aims to do is reinforce, rather than supersede or replace, these existing capabilities already in widespread use, by forming an amalgamation of the information they store and checking their collective beliefs are all in agreement.

WIKID's inception is a result of my personal experiences and frustrations administering *The Tardis Project*, as well as conversations with other administrators on the project, at *The University of Edinburgh's School of Informatics* and *HUBS* (*High-Speed Universal Broadband Services C.I.C*).

WIKID is also more of a methodology and framework for doing such an analysis, rather than a tool tied into the specific design choices and requirements of the *Tardis Project* systems it was tested on. A significant amount of design consideration went into ensuring generalisability.

I AIM TO FIRST DEVELOP AN ABSTRACT MODEL for representing a broad range of system configurations and properties. Following on from that, I aim to implement a tool and system for automated validation of configuration information, that takes cross-source information as input. These inputs may be sourced from a wide array of different parts of the infrastructure management whole. The tool should assist System Administrators in the tasks of managing configuration and pro-actively mitigating errors. We'll then evaluate and discuss the possible benefits and also limitations of this tool.

The primary components of this tool will be the *Data Import* layer that constructs the model from an external data source, feeding into a *Model Checking middleware*, finding and sending issues to an *Output* layer that the end users (System Administrators) will interact with.

1.2 Report Structure

I'll begin this project report by providing a background on the current state of Systems Administration, motivations behind and need for a cross-configuration validation tool such as WikiD in Ch.2. Following on from that, the model WikiD uses to represent configurations and stateful information about a computer system is presented in Ch.3, as well as the algorithms and behaviours of *Consistency Checking*, the primary function of WikiD.

Ch.4 covers the implementation of the WikiD system in code, covering the system architecture and software components belonging to WikiD in general as well as code that was built specifically for running the system on *The Tardis Project*.

I'll then evaluate and discuss the efficacy and usefulness of the tool in Chapters 5 & 6, presenting further work and potential improvements also.

1.3 Contributions

Design & Implementation of WikiD

Implementation of *pywikid*, a Python module that provides common mechanisms, structures, data stores, message passing interface, global locking, task queue and worker logic, self-management and error recovery for all services built within WikiD.

This *pywikid* service code, and a Graph Database implementation (Ch.4 §4.5) represents around 50% of the implementation work.

Redis Graph DB

Graph Database library built as an extension to Redis [2]. Implementation is described in detail in Ch.4 §4.5, motivations Ch.3 §3.1.

[2] Salvatore Sanfilippo (antirez) and Redis Labs Inc. Redis. URL `http://redis.io`. Redis is an open source in-memory data structure store. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial hashes

Deployment of WikiD *to The Tardis Project's Servers*

Involved implementation of the *pywikid* service instances used within *The Tardis Project*, comprised of integrations with our documentation system *NetBox*, DNS server, *HP ProCurve* Managed Network Switches and the *Sentry* error tracking service, as examples of possible WikiD instances.

Deployment to *The Tardis Project* involved rolling out a *Redis* database cluster, and distributing the WikiD Services within the deployment.

Library patch for redis-py-cluster

Python3 compatibility patch (Pull Request #175) into the *redis-py-cluster*[3] library, fixing encoding issues between the binary encoded responses from the server and Python3's native UTF-8 strings. The library is the officially recognised Python binding for clusterised Redis.

[3] Johan Andersson (Grokzen). Redis-py-cluster. URL `https://github.com/Grokzen/redis-py-cluster`. A Python Client for Redis Cluster

Previously Python3 developers were constrained to using display-character only data in Redis, but now can user arbitrarily encoded binary data as supported by Redis and the Python2 bindings.

The patch was made available in Release 1.3.4

Library patch for redis-py-cluster & redis-py

Support for `PUBSUB CHANNELS`, `PUBSUB NUMSUB` and `PUBSUB NUMPAT` commands added to both *redis-py*[4] (Pull Request #819), the officially recognised Python library for interfacing to Redis, and *redis-py-cluster* (Pull Request #181).

The *redis-py-cluster* patch was included in Release 1.3.4, the *redis-py* patch will be made available in the next release (Tentatively, 2.10.6)

[4] Andy McCurdy. Redis-py. URL `https://github.com/andymccurdy/redis-py`. A Python Client for Redis

2

BACKGROUND

So what do we mean by *Prescriptive Configuration & Documentation*, in practice? Most end users, in both the *Microsoft Windows* and *UNIX/Linux* worlds are unlikely to have any experience with, or need for, orchestrated configuration management. But, it is fairly easy to see that as the number of systems you have to manage increases, doing everything manually would quickly become tedious and eventually be practically impossible.

A set of scripts is probably sufficient for managing a small deployment of servers, but that is a pretty barebones and hacky solution. This method also brings a whole host of it's own problems and certainly wouldn't scale well [1]. Thus, *Configuration Languages & Management Systems* [2,3] come into play. There are more than a few of these to choose from - among them are *Ansible, LCFG, L3, Puppet* and *Salt*, each with a slightly different approach but the same end goal of providing a way to automate the distribution (i.e. *prescription*) of configuration directives out to a set of devices. It is fair to say that use configuration management systems does away with the sundry tasks of configuring new installs, keeping packages updated and managing local network configuration.

Infrastructure Management tools provide a documentation-driven method for addressing this remaining space of unprescriptable configuration. *DCIM & IPAM* tools such as *RackTables*[4] and more recently *NetBox*[5] provide structured documentation for many aspects of site architecture, including server rack utilisation, *VLAN*[6] prefixes and configuration, allocation of utilities including physical cabling, IP addresses, as well as tenancy information.

The third and last component of most large-scale Infrastructure deployments are monitoring and analytics tools. As with both Configuration and Infrastructure Management, there are many good choices available to SysAdmins. For Small & Medium Enterprise and

[1] Stephen Quinney. System configuration: An end to hacky scripts? *UKUUG*, 2008. URL `http://www.lcfg.org/doc/sysconfig_ukuug_2008.pdf`

[2] Paul Anderson. Towards a high-level machine configuration system. In *LISA*, volume 94, pages 19–26, 1994

[3]
Note
A *Configuration Language*, as referred to within this work, is a Configuration Management System that employs *either* a domain specific language, or a standardised markup language such as *XML* or *YAML* with a rigidly defined schema / *DTD*.

[4] Denis Ovsienko, Aaron Dummer, Alexey Andriyanov, and Arnaud Launay. Racktables. URL `http://racktables.org/`. A system for documenting hardware assets, network addresses, rackspace, network configuration and more

[5] DigitalOcean Inc. Netbox. URL `https://netbox.readthedocs.io/en/latest/`. Tool to help manage and document computer networks and infrastructure

[6]
Note
In Computer Networking, a *VLAN*, or Virtual Local Area Network is any broadcast domain that is partitioned and isolated in a computer network at the data link layer (OSI layer 2)
(IEEE 802.1Q-2011)

right up to the infrastructure giants like *Amazon Web Services*, *Facebook* and *Google*, monitoring tools are employed in an attempt to keep network behaviour and system health in check. Almost all, if not all monitoring tools in widespread usage take a *symptomatic* approach to diagnosing system issues; They keep an eye on bandwidth utilisation, load balancing, system load, memory pressure and a plethora of other metrics.

Yet, this approach to system monitoring has some flaws. Because these tools monitor symptomatic properties of systems, the best they can do is tell administrators that there *is* a problem – but cannot necessarily provide high fidelity information as to possible root causes, or when the issue was introduced and by what action. Take for example, an administrator accidentally assigning a new machine a range of IP addresses that were already taken, instead of a single one. This isn't particularly hard to do with a malformed CIDR expression.[7,8] Firstly, this misconfiguration won't cause breakage immediately, as the new routing will take a short amount of time to propagate. This is a misconfiguration in the newly provisioned machine – but the manifestation through the lens of network monitoring tools could look like a number of things, as the results of an IP conflict are unpredictable, intermittent and wide-reaching, and can be particularly bad if one of the stolen IPs previously belonged to networking hardware. In the worst case, it could cause a complete denial of service within the LAN.

[7] Yakov Rekhter and Tony Li. Rfc-1518: An architecture for ip address allocation with cidr. Technical report, 1993

[8] Vince Fuller, Tony Li, Jessica Yu, and Kannan Varadhan. Rfc-1519: Classless inter-domain routing (cidr): an address assignment and aggregation strategy. Technical report, 1993

Similarly, capturing the utilisation of available servers is indicative only of the load on the system. If a server is pegged at 100% CPU utilisation, is that because the site is seeing abnormally high traffic and its simply not large enough to handle incoming traffic, or because a recent configuration change broke caching? These are not questions that traditional monitoring tools can directly answer, as they just provide evidence for SysAdmins to hypothesise about, though in this example the answer would hopefully be obvious.

Surely it would be better to try and spot these errors in configuration rather than wait for them to cause breakage?

IT SEEMS INTUITIVE THAT in an increasingly large & complex deployment, the greater volume of information being stored in documentation and configuration directives increases the probability of an error entering the system, irrespective of the pragmatism of the administrators. Added to this, newer technologies in modern networking such as *SDN* [9] and the continuous fuzzing of the *OSI model* (Open Systems Interconnection model) and continued widespread uptake of virtualisation and containerisation introduce further complexity. This has lead to some companies employing brute-force strategies to make sure their datacentres stay up – *Facebook* monitor for network issues by sending a huge volume of network 'fuzz' through their datacentre fabric to verify routes aren't dropping pack-

[9]
Note

SDN: *Software Defined Networking*, a dynamical approach to Computer Networking wherein many network behaviours, such as packet routing are abstracted away from hardware and moved into software.

ets and that switches are operational.

At the other extreme, in a particularly small deployment there may not even be a dedicated SysAdmin running the system – and the smaller scale permits more lax practices. These smaller systems can appear chaotic and even adversarial, with multiple administrators deploying and modifying containers, servers and services.

This is probably true of *The Tardis Project* [10], where all the System Administrators work voluntarily on the servers and services. Coordination between administrators in this setting is not guaranteed, and a reliable source of information pertaining to the whole system's state isn't always available. It is much easier for an administrator to provision a new virtual machine, assign it an IP then get on with whatever it was that they wanted to do, rather than provision the machine, make a note of it in the VM management console, document the new IP address assignment, remember to edit the reverse DNS zone as well as the forward and add the machine to the Wiki.

This environment certainly motivates an automated configuration validator, though as will be seen there's also a strong case for a configuration validator like WikiD in Small, Medium and Large enterprise.

[10] University of Edinburgh The Tardis Project. The tardis project. URL `https://wiki.tardis.ed.ac.uk`. The Tardis Project is a computing facility, run and maintained by students of The University of Edinburgh

3

DESIGN

So, with WIKID we're trying to identify misconfigurations and mistruths within different representations of a datacentre or IT deployment. This presents and interesting problem if we want to remain as generic and as adaptable as possible, instead of tailoring a solution to a specific set of existing tools. We also need to be careful, as the semantics and limitations of whatever model for representing configurations and dependencies we choose will fundamentally constrain what we can then do with the data.

Being able to identify misconfigurations and belief conflicts requires that we ingest configuration information from many heterogeneous sources. Configuration Languages are one possible source, as well as documentation and organically discovered information, but it should be stressed that WIKID has a wide range of potential uses.

3.1 Modelling Configurations

At its root, a configuration directive is a *typed relation*, that assigns some entity to another entity. By *typed* I mean the relation is the assignment of some property; that property is the type.

Take, for example, the requirement that the package 'LaTeX' be installed on a server called *ceilingcat* – this is the assignment of the *Install* relation from the entity *LaTeX* to *ceilingcat*, which can be written $Install_{\ latex \mapsto ceilingcat}$. This relation is directed, meaning that, by definition:

$$Install_{\ latex \mapsto ceilingcat} \neq Install_{\ ceilingcat \mapsto latex} \tag{3.1}$$

In the case of the *Install* relation, clearly the second directive is nonsensical, hence, directivity. We use this left-to-right direction by convention. Not all relations are directed, so to represent these we can

either use a pair of relations going in opposite directions, or choose a convention for direction and stick to it.

Similarly, the same pair of entities may be related with more than one *type* of relation, and that any given node is permitted to have multiple incumbent relations of the same type with the different origins. So, for example, we have a set of relations E:

$$\begin{aligned}\{&Install_{\ latex \mapsto ceilingcat},\\ &Install_{\ emacs \mapsto ceilingcat},\\ &Install_{\ emacs \mapsto bistromath}\} \subseteq E\end{aligned} \tag{3.2}$$

This model for a configuration also extends well beyond software package installation directives; We could model assignments, dependencies and connections also:

$$\begin{aligned}\{&IP\ Address_{\ 193.62.81.4 \mapsto ceilingcat},\\ &IP\ Address_{\ 193.62.81.5 \mapsto bistromath},\\ &Interface\ 02_{\ coreswitch \mapsto ceilingcat},\\ &Interface\ 42_{\ coreswitch \mapsto bistromath}\} \subseteq E\end{aligned} \tag{3.3}$$

Eq.3.3 shows assignments of canonical names to IP Addresses in a zone file, and physical interface connections between a switch and two servers, *ceilingcat* and *bistromath*. The relational behaviours are very similar for both the IP address assignment and the interface connection, but the actual meaning with respect to the real world is quite different.

Using this abstraction of binary, typed relations, it seems we can model the majority of possible states prescribed by system configuration directives. It follows fairly quickly from this use of typed binary relations that we can construct a *Relational Graph* of everything in our relations set, E.

Given that a potentially disconnected, directed multigraph G is defined as having a set of edge tuples, $(x, y) \in E$ and a set of vertices V and a set of relation types T, we have the following definitions:

A relation that is undirected between x and y is equivalent to two directed relations, $R_{\ x \mapsto y}$ and $R_{\ y \mapsto x}$. By convention, the relation $R_{\ x \mapsto y}$ means x provides y with R.

At this point it's important to point out the qualification criteria for being a vertex in this graph. Broadly speaking, the vertices are simply entities we care about, be they physical servers, or virtual machines, pieces of software, facts. Broadly, any *noun* can be a vertex. Similarly relational edges can be seen as *verbs* – install, connect, provide, depend. This flexibility enables the input of information from a very broad range of sources.

$$G = \{V, E, T\} \quad \text{(Graph Definition)} \tag{3.4}$$

$$\forall\, R_{\,x \mapsto y} \in E \;:\; x, y \in V \quad \text{(Relational Graph)} \tag{3.5}$$

$$\forall\, R_{\,\_ \mapsto \_} \in E \;:\; R \in T \quad \text{(Relation Types)} \tag{3.6}$$

$$R_{\,x \mapsto y} \in E \not\Rightarrow R_{\,y \mapsto x} \in E \quad \text{(Directed Graph)} \tag{3.7}$$

$$\forall\, x = x', y = y' \;:\; R_{\,x \mapsto y} \equiv R_{\,x' \mapsto y'} \quad \text{(Relation Uniqueness)} \tag{3.8}$$

$$R_{\,x \mapsto y}, R_{\,z \mapsto y} \in E \not\Rightarrow x = z \quad \text{(Graph Multiplicity)} \tag{3.9}$$

Figure 3.1: Graph Definition Semantics of the WIKID abstract model. Here, we take the symbol '_' to mean an unbound free variable.

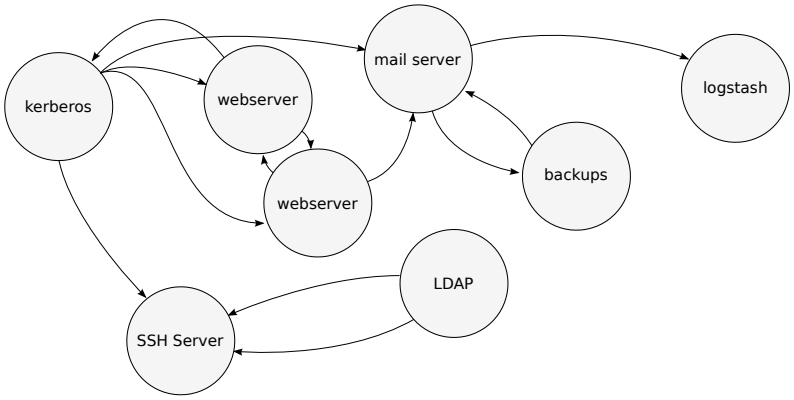

Figure 3.2: Illustrative example of a *Directed Multigraph* showing an arbitrary set of relational edges and vertices.

3.2 *Constructing Models*

For each source of information available to us we build a *partial model* for the *partial state* it encodes. Depending on the source, this may be a *desired* systems state (e.g. from configuration language directives, documentation) or *operational* state (e.g. garnered from inspecting deployed configuration on devices or network behaviours).

Each model can be built asynchronously, meaning the rate at which they are populated and update can depend entirely on the change frequency of the input source, rather than some global schedule. Each model has no interdependence on any other model. Actually building this model with real data adds complications, as uniquely identifying entities can be challenging, especially given our definition of what an entity is allowed to be is very broad.

We'll see more about this problem in §3.3, but for the intents and purposes of vertex matching between graphs, we'll just require that their names are equal. More complex matching would of course be possible, but is outwith the scope of this project. In this implementation, each model builder had to deal with name matching itself before populating the graph.

3.3 Consistency Checking

Given two model graphs, $G = \{V, E, T\}$ and $H = \{V', E', T'\}$, we can check that the relations they represent are consistent by checking that they are *partially homomorphic*. Symmetrically, if we can find a counterexample to the consistency hypothesis then by definition the graphs are *not consistent*. Looking for a homomorphism similar to a *subgraph isomorphism*, but differs in that neither graph must strictly be a subset of the other, and a *homomorphism* allows for compactification of vertices by the morphism, which is therefore *injective* but not *surjective*.

For two graphs G and H to be *fully* homomorphic means that there exists a mapping $f : A \mapsto B$ such that

$$(x, y) \in E \;\Rightarrow\; (f(x), f(y)) \in E' \tag{3.10}$$

$$x,\ y \in V \;\Rightarrow\; f(x),\ f(y) \in V' \tag{3.11}$$

$$T \;=\; T' \tag{3.12}$$

A partial homomorphism is a weaker notion, where we are only concerned with edges between mutual vertices of mutual types. We do not need to find a mapping that goes from all edges in E to all edges in E' *and* from all vertices in V to V', but one f' such that for any two vertices that exist in both graphs, edges between them in one also exist in the other (shown more concretely Eq.3.18).

Using a *set comprehension*, these relationships and the common edges E_c plus common vertices V_c between G and H with the graph model can be expressed as:

$$G = \{V, E, T\} \qquad \text{(Graph Definition)} \tag{3.13}$$

$$H = \{V', E', T'\} \qquad \text{(Graph Definition)} \tag{3.14}$$

$$f' : V \mapsto V' \qquad \text{(Partial Homomorphism)} \tag{3.15}$$

$$V_c = V \cap f(V') \qquad \text{(Common Vertices)} \tag{3.16}$$

$$T_c = T \cap T' \qquad \text{(Common Relations)} \tag{3.17}$$

$$\begin{aligned} E_c = \{R_{\,x \mapsto y} \mid\ & x, y \in V_c, \\ & R_{\,x \mapsto y} \in E, \\ & R_{\,f'(x) \mapsto f'(y)} \in E', \\ & R \in T_c\} \end{aligned} \qquad \text{(Common Edges)} \tag{3.18}$$

For the two graphs to be considered *strongly* consistent it must also be the case that there are absolutely no contradictory relations in one and not the other. If these do exist then we have a counterexample to the consistency hypothesis and have thus shown inconsistency. To check for this, we construct two *difference* (diff) graphs $\hat{G}$ and $\hat{H}$:

$$\hat{G} = \{V_c, \{R_{\ x \mapsto y} \mid R_{\ x \mapsto y} \in E - f'(E'),\ R \in T_c\}, T_c\} \tag{3.19}$$
$$\hat{H} = \{V_c, \{R_{\ x \mapsto y} \mid R_{\ x \mapsto y} \in f'(E') - E,\ R \in T_c\}, T_c\} \tag{3.20}$$

If $\hat{G}[E] \neq \emptyset \wedge \hat{H}[E] \neq \emptyset$ (both diff graphs have some edges, i.e. aren't empty) then the graphs are inconsistent – both hold relations of a common type the other does not, with reference to the same vertices. In the weaker case, where $\hat{G}[E] \neq \emptyset \vee \hat{H}[E] \neq \emptyset$ (either diff graph has edges) this might indicate missing information in one of the graphs, which we shall call a *Partial Inconsistency*

3.3.1 *Worked Example*

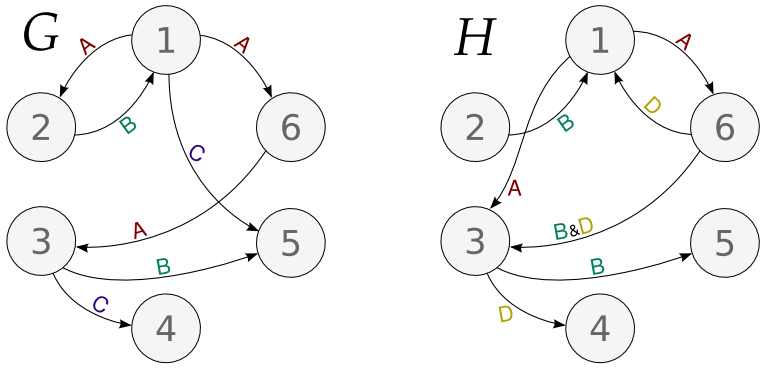

Figure 3.3: Two graphs, G and H with 6 common vertices and some common edges.

Consider two graphs, G and H, as shown Fig.3.3. We'll run through a consistency check on these graphs, which you can already see visibly disagree on some relations. The formal definitions for these two graphs in our model are given in Eq.3.21 and Eq.3.22:

$$\begin{aligned} G &= \{V, E, T\} \\ V &= \{1,2,3,4,5,6\} \\ E &= \{(A_{\ 1 \mapsto 2},\ A_{\ 1 \mapsto 6},\ A_{\ 6 \mapsto 3}, \\ &\quad B_{\ 2 \mapsto 1},\ B_{\ 3 \mapsto 5},\ C_{\ 1 \mapsto 5},\ C_{\ 3 \mapsto 4}\} \\ T &= \{A, B, C\} \end{aligned} \tag{3.21}$$

$$\begin{aligned} H &= \{V', E', T'\} \\ V' &= \{1,2,3,4,5,6\} \\ E' &= \{A_{\ 1 \mapsto 3},\ A_{\ 1 \mapsto 6},\ B_{\ 2 \mapsto 1},\ B_{\ 3 \mapsto 5}, \\ &\quad B_{\ 6 \mapsto 3},\ D_{\ 3 \mapsto 4},\ D_{\ 6 \mapsto 1},\ D_{\ 6 \mapsto 3}\} \\ T' &= \{A, B, D\} \end{aligned} \tag{3.22}$$

Looking at these graphs, we can see that they have common types, T_c containing $\{A, B\}$. Edges of type C only appear in graph G, and similarly those of D are only in H. These edges aren't important for the purposes of consistency checking, so they're discarded.

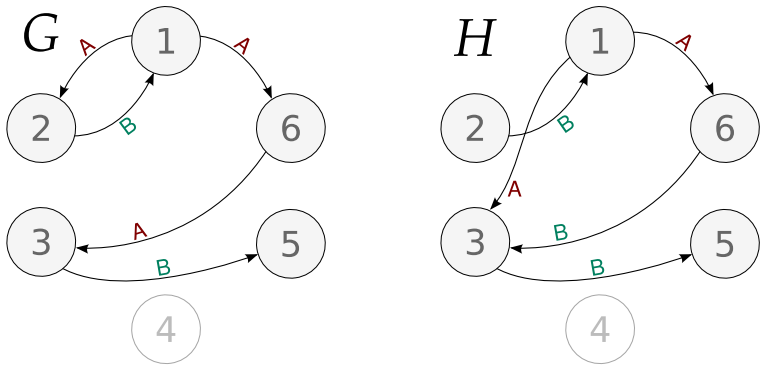

Figure 3.4: G and H again, with uncommon edge types evicted. Vertex 4 is now disconnected from the rest of the graph.

FIG.3.4 shows the reduced G and H graphs with only edges of common types remaining. Note that Vertex 4 is now completely disconnected from the graph as it has no inbound or outbound edges.

We can now build the difference graphs $\hat{G}$ and $\hat{H}$:

$$\hat{G}_E = \{A_{\ 1 \mapsto 2},\ A_{\ 6 \mapsto 3}\} \tag{3.23}$$

$$\hat{H}_E = \{A_{\ 1 \mapsto 3},\ B_{\ 6 \mapsto 3}\} \tag{3.24}$$

As neither $\hat{G}$ nor $\hat{H}$ are empty, these two graphs are *strongly* inconsistent, i.e. both hold a belief that is counterfactual to the other. The causal relation is the A departing 1, with G believing $A_{\ 1 \mapsto 2}$ while H believes $A_{\ 1 \mapsto 3}$. There is also another, *partial* inconsistency. Both graphs claim to know relation types A and B, but each has a relation the other does not – there is partial information between the two. These two inconsistencies are shown graphically, FIG.3.5

And therefore we have shown G and H to be strongly inconsistent, and found all the counterfactual relations between them.

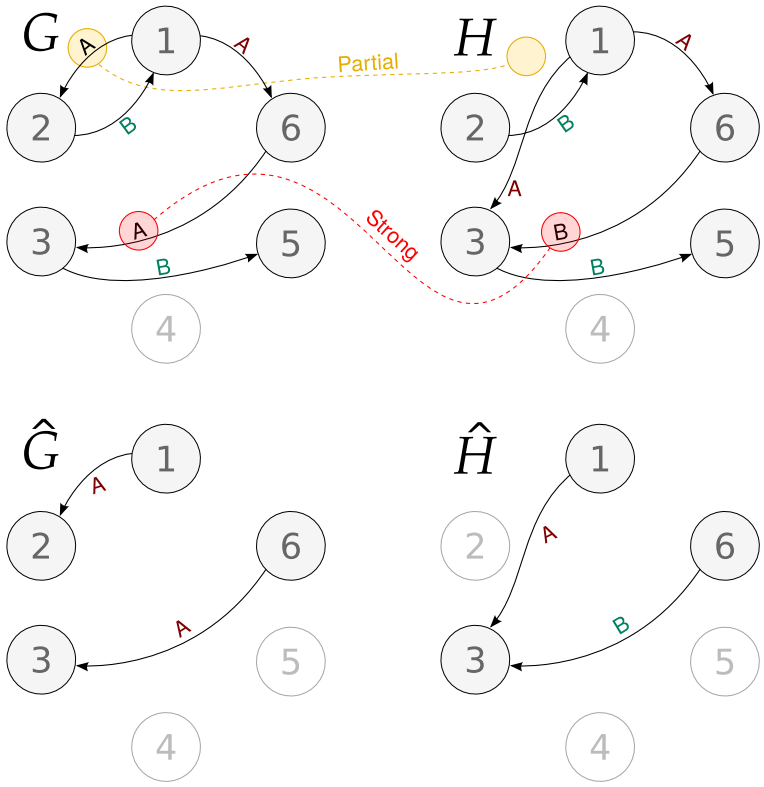

Figure 3.5:
top: Inconsistencies between G and H highlighted
bottom: The difference graphs $\hat{G}$ and $\hat{H}$

3.4 How Well Does The Model Fit the Real World?

Inspecting this model we can see that there are a few types of relation it can't represent. There's an implicit assumption that all relations are many-to-many, or that relations have meta-relations with each-other, such as *mutual exclusion*. These relations cannot be represented within this model. We also cannot model *codependencies* easily, where the existence of some relation mandates that another of different type also exist.

However, WikiD does provide a method for performing checks over these graph models after all, so it would seem acceptable to say that any such relations, like mutual exclusion, required arity or codependencies can be satisfied by making this a check over the graph.

For the sake of argument, we could imagine there could exist a *Turing Complete* Configuration Language. Trivially, we could distribute and then execute a computer program on the nodes to be configured, and we'd have such a system. The effects of this system would then be *irrepresentable* as a graph of objects and relations. Luckily configuration languages and configuration management systems in practice to fall into a more constrained behaviour space, and this is largely a product of what they're attempting to do: provide

and then enact a description of a system state.

There are two main arcs within the configuration language space, being *declarative* languages, that specify a *desired* state and then attempt to match the system's *operational* state to this representation; And *imperative* languages that instead provide a set of instructions to be executed serially in the hope that the system will end up in the correct state.

Some configuration languages provide non-deterministic directives (either probabilistic or random) – these abilities are useful if we wish to balance load on nameservers, for example. It is impossible to know what exact value is taken ahead of time, but bearing in mind that we're attempting to verify the faithfulness of the resultant system state to the configuration we can either ignore the effects of this kind of directive as we assume they're never wrong, or simply check that the end state matches a set of acceptability criteria.

SALT, ONE PARTICULAR configuration language/system[1] uses the *YAML* file format for configuration directives. These form a tree of associations of values to keys. We know from graph theory that all trees are special cases of and therefore representable as graphs. A simple Salt *state file* could look like:

[1] https://saltstack.com/

```
install pywikid:            # Name this state
    pkg.installed:          # Package Installed directive
        - name: pywikid     # Package name
```

This fairly trivially translates to our representation, once applied to a machine called *beeblebrox*:

$$pkg.installed_{\ pywikid \mapsto beeblebrox}$$

The following case is a little more tricky, bearing in mind we do not have a concept of negative relations:

```
remove vim:                 # Name this state
    pkg.removed:            # Package Removed directive
        - name: vim         # Package name
```

Where the package `vim` to be installed on the machine but this directive be in the configuration file, this would be flagged by WIKID as a *partial inconsistency* because both graphs claim knowledge about installations, yet the install directive is missing from the Salt-based graph. *But* there doesn't seem to be much point in us modelling what *isn't* installed on a system, which is a near infinite number of software packages.

This serves as a simple example but demonstration of how we fit the model to external information sources.

4

IMPLEMENTATION

At a high level, WikiD is a distributed system of interconnected *microservices* and a distributed *Redis Cluster*[1] NoSQL data store. This system design has some key properties that are inherently useful for WikiD: it is headless, as it has no core controlling or coordinating node. Services are free to join and part from the cluster as they wish.

Any system monitoring tool that is hypersensitive to intermittent connectivity obviously looses usefulness in a crisis where servers may be down. Building distributability, redundancy, parallelism and replication into the system by design prevents availability from depending on a single node. In a sufficiently large deployment, a significant portion of WikiD nodes could be down and the system as a whole would remain fully available as a result of this distribution and redundancy. Also it is entirely possible to run WikiD on a single server.

Within WikiD a pipeline is formed between the three types of service and their interactions. *Importer Type* Services collect data, populate and update models then inform the *Checker Type* services of changes. The Checkers then evaluate whether any of these changes and new data represent an issue, and if so publish *issue reports* to a queue. The last stage is *Exporter Type* services, which wait on the reports queue, and export issues published by the Checkers to external platforms such as mailing lists and ticketing systems.

A Python module, called *pywikid* is used as the common basis of all components of WikiD. Within *pywikid* are a microservice framework (§4.3), a Graph Database & Redis API (§4.5 & 4.4) and also some of the default, core functions without which WikiD's main features wouldn't work (though, one would be free to extend these or swap them out for functionally equivalent services).

[1] Salvatore Sanfilippo (antirez) and Redis Labs Inc. Redis. URL `http://redis.io`. Redis is an open source in-memory data structure store. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial hashes

4.1 Implementation Aims

I aimed to produce *production ready* code that adhered to a coding standard and was well documented. Given Python3 is the chosen implementation language, I've elected to adhere (mostly) to the *PEP8*[2] Python Coding Style Guide, and to use *'Pythonic'* features like callable objects, generators and iterators.

The *pywikid* module's classes source code is automatically documented by the *Sphinx*[3] code documentation generator. Class and member declarations use so-called *doc comments* that are then pulled out and rendered as HTML or LaTeX human readable documentation. A copy of the generated documentation is included in the APPENDIX.

Given the architecture as will be described shortly, the core WIKID services are implemented as a Python3 module that is then installed as a dependency for all the non-default WIKID services. This helps keep WIKID extensible and source code *DRY*, as well as isolating each service from the others.

[2] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8–style guide for python code. *Available at ht tp://www.python. org/dev/peps/pep-0008*, 2001

[3] "Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license." *http://www.sphinx-doc.org/en/stable/*

4.2 Architecture Overview

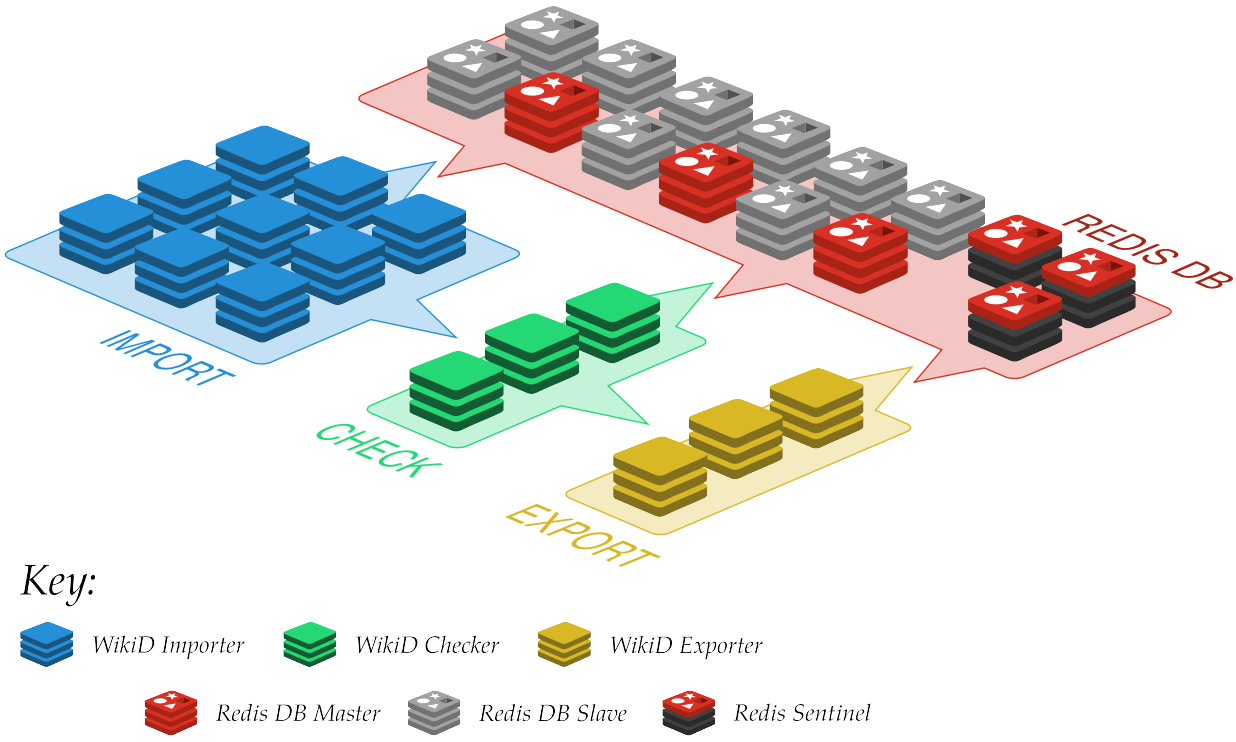

Figure 4.1: Service-level block diagram of the WIKID architecture. Note that the exact number of each system block is purely illustrative and can vary wildly depending on the WIKID setup.

Structurally WIKID is implemented as a three stage pipeline that takes input and constructs a model, then performs checks on the model and passes any detections on to an output stage. All three of these stages interact with the database.

WIKID services and *Redis* (§4.4) instances are distributed across many servers. The normal setup has one or more WIKID Services, one *Redis Master* and a number of *Redis Slaves* per physical machine or VM, and with the *Redis Sentinels* running separately on their own devices, shown FIG.4.2. This layout isn't required though, and every single service could be run on an individual machine, or all could be run on the same. A higher slave-to-master ratio of Redis instances improves read performance and crash tolerance.

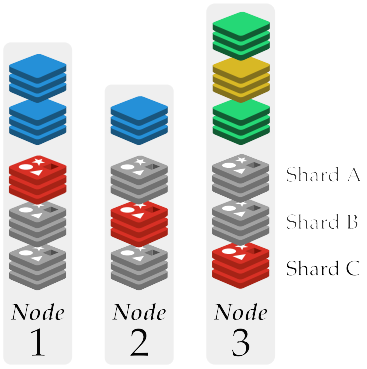

Figure 4.2: Showing a possible allocation of WIKID and *Redis* services to physical machines.

4.3 Microservices Framework

The *Service* class and interface in *pywikid* provides a common microservice implementation. The service interface helps with the core functionalities of communicating with all the other WIKID nodes, job-queueing, daemonisation, reading in service configuration and handling potential crashes & errors.

The microservice design pattern has recently been gaining popularity[4]. With microservices, a clear, well defined boundary exists between different services that communicate using a common interface or protocol. For WIKID this is particularly useful given the *Import* type services will each have to transform their respective input data source into the WIKID Graph Model, meaning each service is likely to have it's own, possibly large, set of dependencies. Using microservices in their own isolated codebases makes managing this much easier when compared to lumping everything together in a more traditional monolithic codebase.

[4] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015

The use of Microservices allows nodes to join and part from the WIKID cluster freely, and for additional services to come online and begin building graphs, performing checks or exporting without the entire cluster needing a restart. This makes managing a WIKID deployment at scale significantly easier, and adding new importers or more redundant nodes a seamless operation.

It is of course important to ensure that all the services agree on their respective communication interface. Microservices also add communication overhead that wouldn't be present in a monolithic application, but by being distributable we can much better parallelise tasks across machines and build in redundancy.

4.3.1 Jobs

A *FIFO job queue* is implemented in the *pywikid.Service* class to facilitate communication between *Import* type services at the start of the WIKID pipeline, and the *Checker* type services in the middle. The jobs interface is relatively simple; a job type and argument are pushed onto the jobqueue by an Importer (the *producer*, in queue jar-

gon). The Checkers sit as *consumers* waiting on the head for a job to be added, which they immediately pop.

Not all Checkers are able to do the same jobs, for instance, *Conch* (which I'll explain in §4.8.1) can only run jobs of jobtype `graphcheck`. If a Checker pops a job from the queue it cannot perform, it *rescinds* it, pushing it back onto the queue's tail, before waiting a short amount of time and attempting to fetch a job again.

4.3.2 *Reports*

Reports are generated by *Checker* type services (thought the reporting interface in *pywikid* is available to all service types). A report has the standard message format:

Member	Description
issue	One of $\{disagree, missing, notice\}$
level	One of $\{fatal, error, warning, info, debug\}$
message	Descriptive string
affected graph	Give the diff graph (CH.3 §3.3) $\hat{G}$ of the graph being inspected
prior graph	Give the diff graph $\hat{H}$, of the prior graph that was found to disagree with $\hat{G}$
commonalities	Common edges, effectively $G_E \cap H_E$ (See §3.3.1)
vertices	Common vertices, $G_V \cap H_V$

Table 4.1: WIKID Standard report structure, as generated by *Check* type services.

Once a *Checker* has found an issue and produced a report, it is dispatched onto the *Reports Queue*. *pywikid* provides a blocking asynchronous method for a *Exporter* type service to sit waiting on a new message to be produced.

It is important to note it that no de-duplication is done by WIKID on reports, and that it is highly likely that many of the same report will be produced while an issue exists. This is an intentional behaviour. De-duplication can be a complex problem, plus, depending on the exporter, it might be useful to know about each and every occurrence of an issue. It is therefore left to the *Exporters* to deal with as they wish – we'll see how the *Sentry* service handles this in §4.9.4. To help, reports are *fingerprinted*, with the idea being that with few exceptions, reports of the same fingerprint are the same issue.

$$B = (\hat{G}_T \cup \hat{H}_T) - (\hat{G}_T \cap \hat{H}_T) \qquad \text{(Types of conflict edges)} \qquad (4.1)$$

$$fingerprint = sorted(list(B) + report_{Level} + report_{\hat{G}} + report_{\hat{H}}) \qquad \text{(As strings)} \qquad (4.2)$$

Eq.4.1 shows the algorithm for generating an issue fingerprint.

4.4 *The Redis NoSQL Database*

Redis is an *in memory NoSQL* database server, meaning it does not implement a relational (Structure Query Language) interface, and instead presents a *schemaless* application programming interface and flexible data-model of datastructure primitives, including the usual *hashmaps, sets, sorted sets* and *lists* as well as some more esoteric & specialised structures, such as *bit fields, hyperloglogs* and *geospatial radius hashes*. Redis also provides a *Publisher/Subscriber* (PubSub) interface, which WikiD uses for informational chatter between nodes.

This schemaless design has pros and cons. We must be careful to ensure all our interactions with the DB do what we intend to avoid invalidating our database. Redis hasn't got a schema to check types against, and performing the wrong operation on a key could cause our client to crash.

As Redis stores data in memory, strong data persistence guarantees are not made – Persistence can be configured to never write to disk, to do so only every n key changes, or to asynchronously append to a difflog after every write. The risk of data loss therefore can be very real for a single node Redis setup, and is one of the reasons Redis is rarely seen in a primary database setting, though this is changing as Redis matures.[5]

Redis is often found as a caching layer between an application and a more traditional relational SQL database, and in fairness it is well suited to this role – while SQL Database systems benchmark at not much more than 7000 serial queries per second, whereas Redis can benchmark as high as 70,000 serial queries per second[6,7] this is mostly due to Redis holding the database in RAM, and from the much simpler queries it has to handle.

Redis 3.0.0 added clustering,[8] allowing Redis deployments to scale up from a single master multi-slave setup to a full multi-master, multi-slave system. Theoretically a Redis cluster can be scaled up to around 1000 nodes. Redis Cluster is able to survive netsplit partitions and remain available where the majority of the master nodes are reachable and there is at least one reachable slave for every master node that is no longer reachable. The *Redis Sentinel* is a special instance of a Redis Server whose task is to watch over the cluster and control failovers if a master or slave node should go down. At least three Sentinel instances need to be running for the cluster to be properly protected.

With clusterised Redis, the hashspace is *sharded* across a number of Redis server instances. The hashing algorithm is well defined, and clients are expected to determine which shard a key is in, and therefore which master is responsible for it. No proxying will be done, which prevents excess communication between the Redis cluster nodes themselves. A Redis server will refuse a request for a value

[5] Leena Joshi. Redis usage survey. *Redis Labs*, 2016. URL `https://redislabs.com/blog/the-results-are-in-redis-usage-survey-2016/`. Found that 67% of Redis users used it for data not stored in any other database

[6] Salvatore Sanfilippo (antirez). On redis, memcached, speed, benchmarks and the toilet. URL `http://oldblog.antirez.com/post/redis-memcached-benchmark.html`

[7] http://redis.io/topics/benchmarks

[8] Salvatore Sanfilippo et al. Redis cluster specification. *Redis Labs*, 2014. URL `https://redis.io/topics/cluster-spec`

in a key it does not control. In addition to this, one can read from both slaves and masters, but only write to masters.

WITHIN WIKID, CLUSTERISED REDIS IS USED as the primary datastore in a 3-1 Slave-to-Master ratio, meaning for every master node there are three slave nodes duplicating it's data. A set of three Redis Sentinels watch over the cluster, which for the *The Tardis Project* WIKID deployment was 12 Redis Server instances, 3 master 9 slave. A common database abstraction API is implemented within the *pywikid* module and made available to all services. This API presents utility methods as well as the *Graph Database Implementation* (§4.5) which is used to build the configuration models (CH.3 §3.1).

4.5 A Graph Database Implementation with Redis

Given the model as outlined CH.3 §3.1 uses a graph representation of configuration, WIKID needs a performant and capable Graph Database for storing and retrieving *multiple* graphs. Labelled Directed Multigraphs are one of the more complicated species of Graph from discrete mathematics, where vertices are labelled and edges (the relations) have a direction and a type. These graphs can be difficult to represent well as a datastructure.

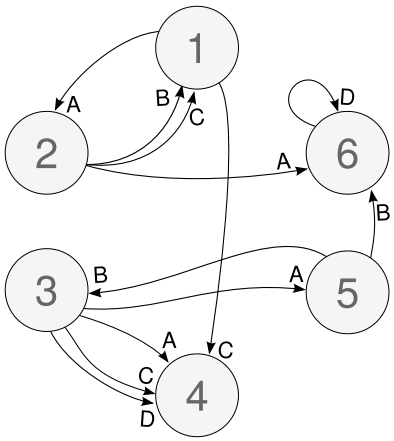

Figure 4.3: An example multidigraph with the distinguishing features of loops, multiple edges, directed edges and labelled edges.

The *Adjacency Matrix* representation of this graph is given EQ.4.3

Consider the graph shown FIG.4.3, which has 6 vertices $\{1, 2, 3, 4, 5, 6\}$ and 12 edges of 4 types $\{A, B, C, D\}$. We can represent this graph as a 6×6 *adjacency matrix*[9] – a datastructure that has been shown to be a succinct datastructure of near-optimal spatial complexity for graph representation[10] – where each element is the set of typed relations connecting the two vertices (CH.3 §3.1) encoded by

[9]
Note

For a *simple* graph with a set of vertices V, the adjacency matrix is a symmetric matrix of booleans, dimension $|V| \times |V|$.

In a *directed* graph, the matrix is no longer symmetric. In a *labelled* graph, we replace the boolean elements with the edge labels, and in a multigraph we replace them with sets or lists of edge labels.

Loops within a graph always occur on the leading diagonal in the adjacency matrix

[10] György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984

the rows and columns, like so:

$$\begin{pmatrix} \emptyset & \{A\} & \emptyset & \{C\} & \emptyset & \emptyset \\ \{B,C\} & \emptyset & \emptyset & \emptyset & \emptyset & \{A\} \\ \emptyset & \emptyset & \emptyset & \{A,C,D\} & \{A\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \{B\} & \emptyset & \emptyset & \{B\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{D\} \end{pmatrix} \tag{4.3}$$

The adjacency matrix is indexed in row-major order from top left, C-style. So if the element at $(3,\ 4)$ has members $\{A, C, D\}$ this means $A_{\ 3 \mapsto 4}$, $C_{\ 3 \mapsto 4}$ and $D_{\ 3 \mapsto 4}$, as can be seen Fig.4.3 (Recall from Ch.3 §3.1 that relations are left-to-right)

We could alternatively store graphs as *adjacency lists*. For a sparse graph, an adjacency list requires significantly less space than the equivalent fully-populated/allocated adjacency matrix would. However, with a list, there are a number of downsides. Firstly, getting the inbound edges to a vertex requires a search through *all* of the lists in the graph, which has time complexity $O(|E|)$. Finding an outbound edge to a particular vertex k is of order $O(|E_k|)$ (where E_k is the set of edges involving k). For large graphs of very high degree, this would be particularly slow. The adjacency list representation of Fig.4.3, equivalent to the adjacency matrix in Eq.4.3 is given Eq.4.4.

$$\begin{aligned} &1 \vdash (A \to 2, C \to 4) \\ &2 \vdash (A \to 6, C \to 1, C \to 1) \\ &3 \vdash (A \to 4, C \to 4, D \to 4) \\ &4 \vdash (\emptyset) \\ &5 \vdash (B \to 3, B \to 6) \\ &6 \vdash (D \to 6) \end{aligned} \tag{4.4}$$

However, we can avoid having to fully allocate an array of memory to store a large adjacency matrix. Given the sparsity of the matrix (most elements are likely to be $\emptyset$, the *empty set*) and considering the performance properties of the datastructure primitives Redis provides (§4.4), *pywikid* implements a multidigraph in the class *GraphRedis* using *Sets*, *HashMaps* and by using an implementation trick that exploits the fact that the Redis keyspace can itself be treated like a big hashmap, by using a key schema and encoding the adjacency matrix co-ordinates in the matrix element's key name.

With this datastructure layout, we have $O(1)$ access to vertices' metadata, we can get vertex degree and all, inbound or outbound edges in $O(1)$ as both the set membership operator and hashmap lookup are $O(1)$ in Redis.

As we have to store multiple graphs, prepending a graph's name

to every key in that graph prevents keyname collisions. A global set of graph names then needs to be maintained.

To STORE VERTICES, we maintain a set of vertex names within our graph namespace. In *pywikid* this set is called `<graph>:_vtx`. We also need to store metadata information about vertices, such as the last time a node was seen, or store references to keys in other databases, etc. This *key, value* data is stored in a *hashmap*.

A *set* holds the edge types between *source* and *dest*. We encode the source and destination in the edge key's name itself, as shown TABLE 4.2, `SADD` commands. It should be noted that Redis has no problems being asked for values in nonexistent key, and happily returns *None*.

Storing edges and vertices in as many separate keys as possible has a number of advantages – when using clusterised Redis, a good heuristic is to try and keep the number of keys high and the size of the values in those keys low, rather than having a few very large keys. This spreads the data evenly across the keyspace and therefore evenly across the cluster, and if we've got sets for each edge, we can then make use of Redis' native *set* operations such as `SINTER` or `SUNIONUPDATE` on the server-side, rather than doing a lot of data munging client-side.

As an example, the graph shown FIG.4.3 could then be stored in Redis with the commands shown TABLE 4.2. '`SADD` x y' is the add 'y' to set 'x' command, and '`HSET` x y z' sets the field 'y' in hashmap 'x' to the value 'z'. Arguments are delimited by spaces.

Redis Command	Description
`SADD graphs example`	Add *example* to the *graphs* set
`HSET example:_vtx:1 seen 793065600`	Set the key *seen* in the vertex hashmap to a value
...	We could add metadata to any more of the vertices
`SADD example:_adj:1:2 A`	Add $A_{1 \mapsto 2}$ to the adjacency set for vertices 1&2
`SADD example:_adj:1:4 C`	$C_{1 \mapsto 4}$
`SADD example:_adj:2:1 B C`	$B_{2 \mapsto 1}$, $C_{2 \mapsto 1}$
`SADD example:_adj:2:6 A`	$A_{3 \mapsto 4}$
`SADD example:_adj:3:4 A C D`	$A_{3 \mapsto 4}$, $C_{3 \mapsto 4}$, $D_{3 \mapsto 4}$
`SADD example:_adj:3:5 A`	$A_{3 \mapsto 5}$
`SADD example:_adj:5:3 B`	$B_{5 \mapsto 3}$
`SADD example:_adj:5:6 B`	$B_{5 \mapsto 6}$
`SADD example:_adj:6:6 D`	$D_{6 \mapsto 6}$

Table 4.2: Redis commands to build the graph in FIG.4.3

The implemented Graph Database API can be seen in the *pywikid* documentation, included in the APPENDIX.

4.6 Alternative Graph Databases

I elected to implement a Graph Database from scratch for a couple of reasons. Of course, there are existing databases for storing graphs that present powerful APIs for querying graph-based knowledge. One of the complications of choosing a Graph Database is that we need to be able to represent typed, directed multigraphs where the set of possible admissible types is infinite and not know to us ahead of time – basically, we need to have the ability to introduce new types into existing Graph stores.

Secondly, we need to store multiple graphs and then be able to compare them with one-another. And as previously laid out in our *Implementation Aims and System Architecture* (§4.1 & 4.2), we need a distributable Database that has clustering and redundancy features. As will be highlighted later, in CH.5, §5.2 there is a clear need for a high-performance database system.

Given the design requirements we can rule out all schema-based databases immediately. Specialised Graph Database systems such as *Titan* and *Neo4j* are potential candidates for use. Titan has support for multiple storage back-ends, including *Apache Cassandra*, which is very similar to Redis, and benchmarks comparably[11]. Neo4j is built atop a custom back-end. Thinking about the diff graphs (EQ.3.19) we need to build, they really treat the graphs more like sets than paths, and we're not interested in many graph operations, such as cycle detection, shortest-path planning etc.

[11] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5 (12):1724–1735, 2012

There's also an argument for maintaining simplicity – With a large amount of code also being written for the deployment and workings of the WIKID services, Redis is a simple single-binary solution to data storage. A lot of alternative Graph Databases either have complex installation procedures and/or many dependencies.

The *pywikid Graph Redis* implementation was one of the most complex parts of the project, and went through more than one major overhaul during development to change how graphs were represented in the database, as I experimented with the model and settled on it's semantics. Having the flexibility of self-implementing a Graph Database API helped make this experimentation process easier.

4.7 *Implemented Services*

The following sections detail implementation of the standard WIKID services in §4.8 as well as those specific to *The Tardis Project* in §4.9.

Service Name	Type	Graph	Integrates with
Example	Importer	example	MediaWiki

Each service section will be prefaced with a small table like the one above, giving the name of the service, it's type (see CH.3, §4.2 for types) and what external entity it integrates, if applicable.

The flow through the WIKID pipeline and *type* of each service is illustrated FIG.4.4

Figure 4.4: WIKID Service types for the services implemented, both as standard and for *The Tardis Project*, with reference to the architecture put forward by FIG.4.1

4.8 WIKID *Default Services*

These services are packaged as part of the *pywikid* module and provide the core WIKID functionality of consistency checking (*'Conch'* §4.8.1). Also provided are two WIKID housekeeping services, a stale data garbage collector *'Trawler'* (§4.8.2) and *'Watchdog'*, a self-monitoring cluster health checker (§4.8.3).

These services would be used in all WIKID deployments.

4.8.1 *Conch*

Service Name	Type	Graph	Integrates with
conch	Checker	*all graphs*	WIKID Graphs

Conch implements consistency checking as described in CH.3, §3.3 with effectively no modifications.

When Graphs are updated by *Importer* type services, they register a *'graphcheck'* job on the jobqueue. *Conch* service instances block on the queue waiting for a job to be produced by a *Importer*. This way we can have many Conch, or indeed many other types of *Checker* running checks in parallel.

Conch and most *Checker* type services will emit standard WIKID issue reports (§4.3.2) upon performing their check and finding an error. Reports are pushed to the reports queue, which is separate from the jobqueue.

From the *graph check* job, we are given a reference to the graph to be checked, which we shall call the *affected* graph. We then iterate over all the other graphs in the database, checking each in turn as the *prior* graph.

Redis has a *set scan* iterator operation that provides weaker guarantees about the members a scanning function will see as it iterates over a set, stating that:

> *A given element may be returned multiple times. Elements that were not constantly present in the set during a full iteration, may be returned or not: it is undefined.*

However, *scan's* big advantage is that it prevents *Conch* from needing to make its own full client-side copies of both *both* graphs it is checking, but as stated above, we may check the affected graph against the same prior graph.

4.8.2 Trawler

Service Name	**Type**	**Graph**	**Integrates with**
trawler	Checker	*all graphs*	WIKID Graphs

One of the problems that the *Graph Redis* implementation, and the mechanics of how we populate graphs by blindly adding without checking for pre-existence §4.5 presents is that services add to graphs without having prior knowledge of existing nodes, automatically overwriting where data exists, or adding if it doesn't. This means that old relations and nodes that are no longer discoverable because they've been removed from the input information set will still be in the graph, and so will continue to cause failed consistency checks.

Trawler addresses this issue, as a *Checker* type service that will iterate over all graphs registered within WIKID at regular intervals, and evict stale data. It does so in a *Least-Recently-Updated* manner. All vertices are tagged with the time they were seen by the *Importer* type services, so through iterating over a graph we can determine it's age, i.e. the seen time of it's newest vertex. *Trawler* is configured with a maximum-TTL for vertices in graphs, and will evict all vertices and edges from graphs where the vertex was last seen $age - TTL$ time ago.

Trawler logs the evictions it makes, and will generate a report at most once daily of all of these if it did actually evict anything from

the graph. This report of type *notice*, level *info* will then be picked up by the *Exporters*.

4.8.3 Watchdog

Service Name	Type	Graph	Integrates with
watchdog	*N/A*	*N/A*	Redis & WIKID clusters

Managing a *Redis Cluster* can be quite complicated, as the approach Redis takes is to minimise magic by giving administrators a lot of control over how the hashspace is sharded across nodes, and how nodes join the cluster. In a very large cluster, manually maintaining this would be tricky. Redis also produces a lot of information we can look at, so *Watchdog* exists to inspect the behaviour of all of the Redis database cluster nodes, as well as keep tabs on WIKID Services.

Watchdog's primary functions are checking that all Redis cluster nodes agree on the time, monitoring the PubSub traffic and that the hashspace is *well sharded*, meaning that the standard deviation of the database size of every node is low with reference to the mean database size. A poorly sharded database would incur performance penalties and reduce Redis' resilience to node crashes.

This service is *'Out of Pipeline'* in that it does not interact with the graph models. It is primarily used on the command line.

4.9 WIKID *Service Integrations for Tardis*

The following sections detail the WIKID microservices specifically implemented for importing information within *The Tardis Project's* administration environment.

4.9.1 DNS Discovery Service

Service Name	Type	Graph	Integrates with
dns	Importer	dns	DNS

Service Name	Type	Graph	Integrates with
dns	Importer	dns_reverse	DNS Reverse Zone

DNS, the *Domain Name System* is the hierarchical naming system for internetworked computers that translates *domain names* to *internet protocol addresses*. DNS answers queries about either a name, returning an IP address, called a *forward lookup*, and also by returning a

canonical name when asked about an IP address, called a *reverse lookup*.

DNS is configured via zonefiles that are held on a nameserver, generally one file per domain. These zone files hold DNS records within the domain, one per line in the format:

```
kal           IN A            193.62.81.1
davros        IN A            193.62.81.15
www           IN CNAME        davros
```

In the case of the `A` record, for *Address*, a name on the left is assigned to the IP address on the right. The `CNAME` is a *Canonical Name*, saying in this case that the name `www` is more canonically a reference to the name `davros`. These names are also relative, so the rest of the fully qualified domain name is implicit, in this case `tardis.ed.ac.uk`.

DNS Zone files also encode zone metadata - each has a serial number that *must* be incremented every time changes are made so that other DNS servers and cached lookups know to refresh. DNS also requires the `SOA` record type, where the zone file and nameserver give information about how long the zone should be cached for, contact information, who the other nameservers are higher up in the hierarchy [12] and other information, such as geographical location. *The Tardis Project's* `SOA` section looks like this:

[12]
Note
The DNS hierarchy has *Top Level Domains*, or TLDs. These are the 'endings', `.com` `.org`, `.net` and so forth. The nameservers for these are fixed and regulated by IANA and ICANN.
`tardis.ed.ac.uk` follows the hierarchy `ed.ac.uk` → `ac.uk` → `.uk`

```
$TTL 14400

; S.O.A.
@       IN SOA  rose.tardis.ed.ac.uk. support.tardis.ed.ac.uk. (
                        2017032300 ; Serial - date last updated
                        3600       ; Refresh (1 hour)
                        600        ; Retry (10 minutes)
                        2419200    ; Expire (4 weeks)
                        86400 )    ; Negative Cache TTL (1 day)
        NS      rose.tardis.ed.ac.uk.
        NS      dns0.inf.ed.ac.uk.
        NS      cancer.ucs.ed.ac.uk.
@       IN LOC  55 56 41.737 N 3 11 14.741 W 90m 10m 100m 10m
@       IN RP   support.tardis.ed.ac.uk. tardis.ed.ac.uk.
        TXT     "Contact: support@tardis.ed.ac.uk"
```

The '`@`' refers to *the current domain*, `tardis.ed.ac.uk`.

'DNSDisco' is a pywikid service integration that imports both forward and reverse DNS zones into *two* models – this service is unusual in that it maintains more than one graph model. The DNS protocol has a request called an *AXFR*, which asks the server to perform a full zone file transfer to us, the requester. This is the fastest

method of pulling a whole zone, and requires no prior knowledge of names within the zone. DNSDisco initially performs an AXFR request against the DNS server it is configured to import from, which results in a full set of records being returned.

To limit the rate at which DNSDisco makes AXFR requests, we store the TTL given in the `SOA` section as t, and re-fetch *just* the `SOA` at a rate of $\frac{t}{4}$. If the serial number in the `SOA` changes, or the TTL expires, we will then do another full AXFR.

We add the records that we can cross-check with other data imports into WIKID to main *dns* our model - these are the `A` records from before, the `AAAA` records which are basically the same as `A` but for IPv6, and the `CNAMES`. An `A` record like

```
davros        IN A           193.62.81.15
```

translates into a WIKID graph representation $A_{\ 193.62.81.15 \mapsto davros}$. Important to note is that we treat the IP address 193.62.81.15 as a vertex, and relate it to some nebulous name *davros*. As was mentioned in CH.3 §3.1, the abstracted graph's vertices are by no means always physical machines, but basically nouns. This would allow further relational properties to be assigned to the IP.

Similarly, the `CNAME` from above translates to $CNAME_{\ www \mapsto davros}$. Each of the vertices added to the graph in this manner are tagged with the time at which DNSDisco last saw their record in an AXFR.

AFTER RETRIEVING THE FORWARD ZONE, DNSDisco then iterates over all of the IP addresses it received and performs a reverse lookup on each one. While a reverse AXFR is technically allowed by the DNS spec, using lightly strange encoding, e.g. `61.82.193.in-addr.arpa` for all IPv4 addresses in the range `193.62.81.1/24`, we want to avoid this as it is not necessarily the case that the zone we are concerned with is the authority for these addresses, or they might be split into separate VLANs, or... the server might flat-out refuse a reverse AXFR. We should also consider the possibility that an IP may have been incorrectly configured into either zone, so being able to discover across different zones helps up spot misconfigurations.

A *reverse* DNS query receives a `PTR` as the result, which is very similar to but not really the same as a backwards `A` or `AAAA`. While there may exist many `A` records from different names to the same IP address, and *technically* you should be able to do the same[13] with `PTRs` from the same IP to many names, by convention and to avoid undefined behaviour in many DNS name resolvers the practice of having multiple `PTRs` for a name is strongly frowned upon. For us, this means that there isn't a bijection between `As` and `PTRs`, so we don't *expect* every edge seen in the forward DNS model to be present in the reverse DNS model, which will actually turn out be a set of dis-

[13] Paul V Mockapetris. Rfc 1035 domain names: Implementation specification, section 3.5 in-addr.arpa domain. Technical report, 1983

connected 2-vertex graphs, from IP addresses to names and nowhere else.

Nevertheless, the responses received from the reverse lookups are added to the reverse DNS model as `A` relationships, i.e. we assume for simplicity's sake that

$$A_{\ x \mapsto y} \Leftrightarrow PTR_{\ y \mapsto x} \tag{4.5}$$

This has a subtle effect on consistency checking that I'll touch on in the Evaluation, Ch.5, in that the mis-modelling of this relation causes WikiD to raise *partial inconsistency* issues as a result of the one-to-one-edness of `PTR`s but the perceived one-to-many-ness of the `A`s.

4.9.2 *Netbox*

Service Name	Type	Graph	Integrates with
netbox	Importer	netbox	Netbox DCIM & IPAM Database

The Tardis Project uses NetBox[14], a *DCIM & IPAM* tool that has recently been released by the cloud hosting provider *DigitalOcean*, who use it internally for system management and infrastructure orchestration. It is in effect a structured *wiki* with it's own data model specifically aimed at the *DCIM* and *IPAM* space. An example page from *NetBox* is shown Fig.4.5.

[14] DigitalOcean Inc. Netbox. URL `https://netbox.readthedocs.io/en/latest/`. Tool to help manage and document computer networks and infrastructure

NetBox presents a HTTP API for retrieving information, which is integrated into a WikiD service. NetBox is the most complete single source of information within Tardis, having the majority of our networking and hardware information. This WikiD service constructs a graph of the stored IP allocations, and network switch port connections.

As with *DNSDisco*, IP address allocations are added using a *A* or *AAAA* type relations:

$$A_{\ Address \mapsto Device} \tag{4.6}$$

The switch interface connections, which represent the physical network cabling between network interface ports in servers and switches are also added to the model, but because a physical cable is essentially directionless, we add two edges in both directions, so for example:

$$interface{:}etho_{\ server \mapsto switch}, \quad interface{:}P42_{\ switch \mapsto server} \tag{4.7}$$

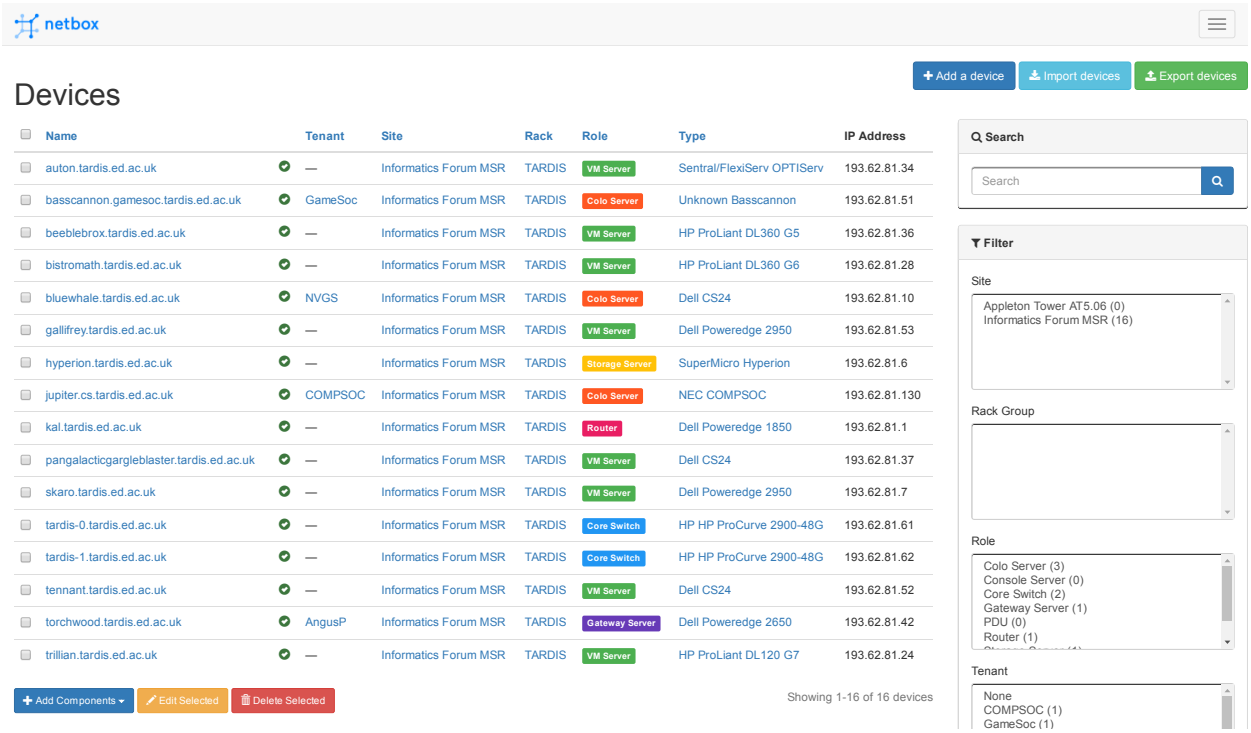

Figure 4.5: The *NetBox Devices Page*, showing physical machines and groupings within *The Tardis Project*.

means that the server's network interface, here called *etho* is connected to the switch's port *P42*. By convention, the relation is named for the interface it originates at.

It would be possible to import further information from NetBox, but as NetBox is a recent addition to *The Tardis Project*, we have yet to fully populate it. Once added to NetBox, adding the new information to the WikiD integration would be trivial as the NetBox API client code and WikiD import code is already complete.

Because NetBox is focused on physical infrastructure orchestration, it doesn't store other names that would be comparable to *DNS-Disco's* `CNAME` records.

Implementation of the NetBox service provided an unexpected challenge, as the relatively new HTTP API does not yet support fetching of the activity stream. This import into WikiD takes about a second at *Tardis'* scale of a single 42 unit rack of servers, so it seems that the naïve import method of rebuilding the entire graph at a set interval is infeasible.

To tackle this I ended up writing a web page scraper, complete with CSRF-protected login scripting to pull the activity feed off the NetBox homepage and use that to rate limit by deciding whether we should import changes again or not, based on if any administrators have altered the NetBox database since the last pull. The NetBox developers plan to add activity support in API v2.0.

4.9.3 *Switch*

Service Name	Type	Graph	Integrates with
switch	Importer	switch	Managed HP ProCurve Switches

The *Hewlett-Packard ProCurve™ 2900-48G* network switches within *The Tardis Project* have a rudimentary management interface accessible over SSH. While the switch does support SNMP, not all of the desired information (such as switch power supply and fan health) was available through that interface, and the SSH method was faster to develop.

The WikiD *Switch* service scrapes most of the status and configuration information from the switch, by sequentially executing shell commands with an *expect* script.

This service presented a particular challenge to implement, as the SSH interface has some abnormal behaviours. The Python code ends up calling out to an *expect (TCL)* script that imitates a user's keystrokes and input to the SSH console. Native Python methods of scripting the SSH interaction failed to work correctly, with neither *Fabric*, library for the use of SSH within Python, nor *pexpect*, a port of *expect* to Python worked.

Collection of a large amount of switch information was completed, but fetching a complete *ARP table* [15] proved very difficult. The ARP table is cached by the switch as traffic goes over it's interfaces, giving us a mapping from *MAC* addresses for particular machines to interface ports (From OSI Level 1 to Level 0). This means the full ARP table is not present on the switch most of the time.

To focus development efforts I stopped working on the Switch importer before it was able to populate a model, as *DNSDisco* and *NetBox* importers were higher priorities. As with the *NetBox* service in §4.9.2, these interface connections could be added to the graphical model as pairs directed edges between interfaces.

[15] *Note*
ARP: Address Resolution Protocol, in networking is the lower level routing protocol that maps IP addresses to MAC addresses, which are generally statically assigned to devices by manufacturers.

4.9.4 *Sentry*

Service Name	Type	Graph	Integrates with
sentry	Exporter	*N/A*	Sentry error tracking service

Sentry is a hosted real-time error tracking and notification system, aimed primarily at Web & Mobile Application developers. It provides a Web API for publishing structured error reports and logging information. Sentry will group issues by their fingerprints,[16] and show time-series plots of issue occurrences to help admins understand the frequency, scope, and impact of errors. Sentry will notify

[16] Sentry *rollup and aggregation* is quite complex and best explained by https://docs.sentry.io/learn/rollups/

Administrators when previously unseen issues appear. Part of the Sentry dashboard for WIKID can be seen FIG.4.6.

Note
Sentry has actually been integrated as an automagically loaded feature (if configured) for *all* WIKID services, as part of the microservice framework (§4.3), to capture any exceptions or crashes that occur in WIKID software itself, and dispatch them to the *Sentry.io* servers.

The *Sentry* service is the only implemented *Exporter* for *The Tardis Project* WIKID deployment, so serves as the primary interaction of whole WIKID system with administrators.

This service has access to all graphs and is allowed to construct it's own, it has no need to do either of these because it's an *Exporter* type at the end of the WIKID pipeline.

The service must first initialise a *raven* client (*raven* is the name of Sentry's endpoint client SDK), with a set of client API keys forming what Sentry call a *DSN* (Data Source Name). As an *Exporter* we wait on the *reports* queue for a *Checker* type service to find an issue and push it onto the queue – otherwise, we do nothing.

Once a *Report* has been received, we have to rearrange it from the WIKID report structure (§4.3.2) into a `'raven.events.Message'` object. This Raven message is tagged with the names of the two graphs in conflict, and a human-readable explanation of the issue is generated. A full Sentry report can be seen FIG.4.7, but the key parts of a *level:warning*, *issue:disagree* type report are the generated message and tableau:

`Conflict between dns_reverse and dns`

Source	Relation
`dns_reverse`	*A* 193.62.81.33 ↦ *chronovore.tardis.ed.ac.uk*
`dns`	*A* 193.62.81.33 ↦ *gradeatron.tardis.ed.ac.uk*

Table 4.3: Example tableau and message for a report from the *Sentry* Service.

```
'dns_reverse' was marked as the updated graph.
(There are no further common edges between these vertices)
```

This particular issue pertains to a *strong* inconsistency between `dns`, the forward DNS zone, and `dns_reverse`. This report would have been triggered by *Conch* checking a newly updated `dns_reverse` graph and finding it to be inconsistent with the existing `dns` graph. In this particular case, the Reverse DNS zone is in fact incorrectly configured.

This tableau and message aim to present the inconsistency in a clear and concise way while providing administrators actionable information about the cause of the finding.

The other types of *Report* are *missing* and *notice*. Missing Reports are used for the possible inconsistencies as discussed earlier, while *notice* reports are more generic (an example *Checker* that generates *notice* reports is *Trawler*, §4.8.2)

Figure 4.6: *Sentry.io Dashboard*, showing unresolved issues from WikiD.

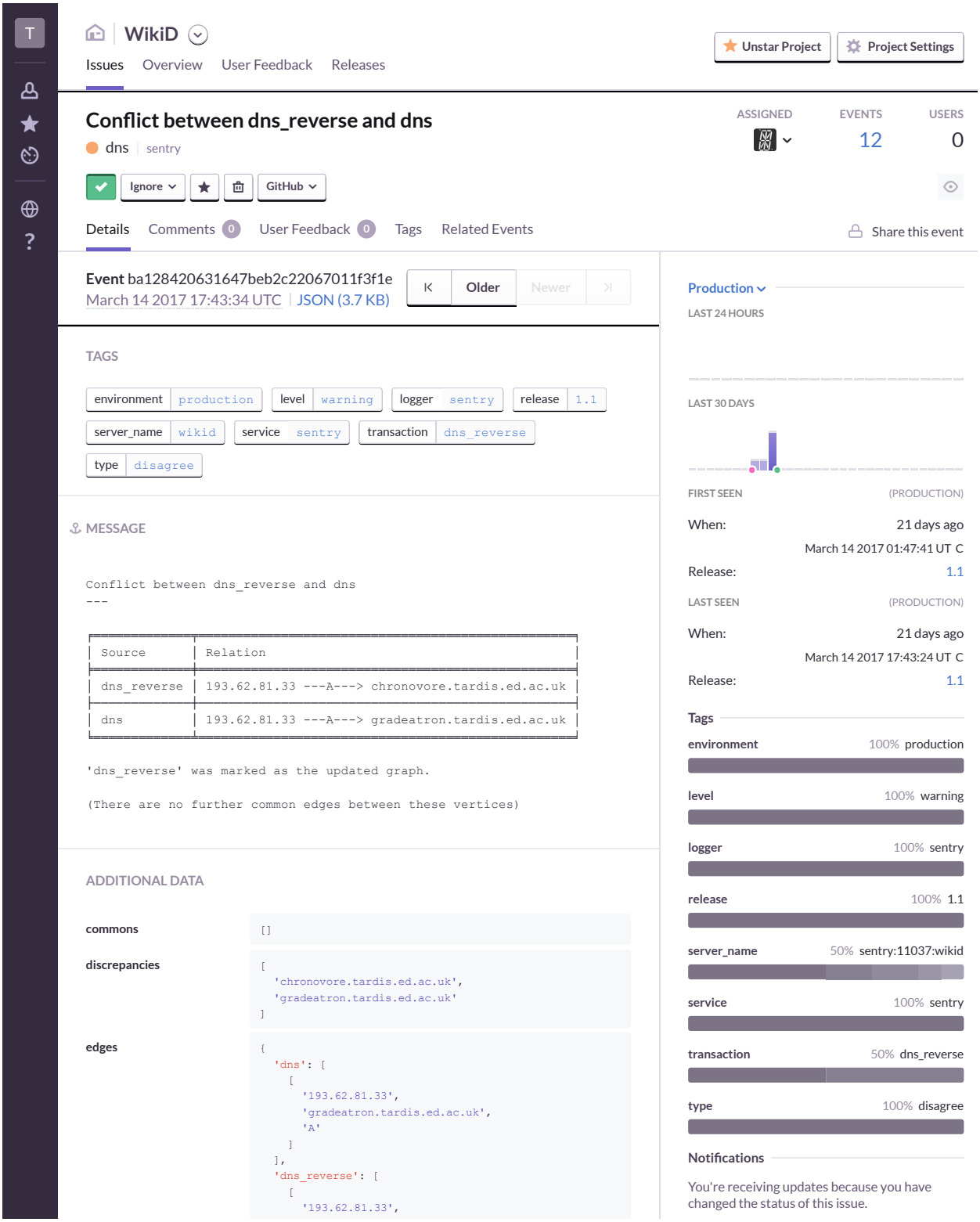

Figure 4.7: *Sentry.io Report*, showing a conflict between two graphs, *DNS* and *Reverse DNS* (Both from the *DNSDisco* service).

5

RESULTS & EVALUATION

5.1 Evaluation of Performance on Tardis

WIKID has been deployed to *The Tardis Project* for an evaluation period of 22 days, at the time of writing. As described in CH.4, the *Tardis* deployment has services *NetBox*, *DNS* and *Switch*, on-top of the WIKID default services.

WIKID immediately discovered 14 issues, 8 were strong inconsistencies and 6 were partial. The incidences of inconsistencies with respect to the services are shown TABLE 5.1.[1]

[1] *Note*
The switch Graph represents a small amount of knowledge, and in addition can only be checked against NetBox. The NetBox documentation of switch connections was actually added by reading these values off the switch's management interface, so I would not have expected any inconsistencies.

Strong Inconsistencies

	DNS	*DNS Reverse*	*NetBox*
DNS	-	6	1
DNS Reverse	6	-	1
NetBox	1	1	-

Partial Inconsistencies

	DNS	*DNS Reverse*	*NetBox*
DNS	-	4	1
DNS Reverse	4	-	1
NetBox	1	1	-

Table 5.1: Showing the tabulated coincidence matrices of inconsistencies between the deployed WIKID services on *The Tardis Project*

Of the total 8 strong inconsistencies WIKID found, all were true-positives. 6 have since been fixed based of the WIKID reporting. The remaining 2 issues actually pertain to misconfigurations outwith *The Tardis Project*' control, and as such can't easily be fixed by us.

Trawler ran once, after these issues were fixed and evicted the stale edges from the graphs.

As for the 6 partial inconsistencies, which give reports of potential missing information, as we would expect the true-positive rate of 100% for strong inconsistencies isn't mirrored for partials. None of the partial inconsistencies reported are factually incorrect, but most

have been dealt with by ignoring the report and taking no corrective actions. Most of them where missing information between *DNS* and *Reverse DNS*, which arise because of the `A` record `PTR` record in-equivalence covered CH.4, §4.9.1, EQ.4.5. The other is all the IP Addressees missing from *NetBox* when compared to *DNS*, which is a result of DNS having addresses for all *Tardis* machines including virtual ones, which aren't recorded in *NetBox*.

DUE TO TIME CONSTRAINTS, the WIKID deployment to *Tardis* only covered a portion of the potential data inputs that would have been appropriate. The virtual machine management system *Proxmox* could have been integrated to provide a fourth angle on network information, and scope outsize *IPAM* and physical connectivity wasn't explored, in part because *Tardis'* deployment of a proper configuration management system is still under-way and so that form of data was not available.

That being said, WIKID did discover a set of issues that were previously unknown to the administrators and provided actionable information that was quickly used to resolve the causal errors.

However, of the 6 *partial* inconsistencies, all bar one have been marked as resolved without administrators taking any action to address the cause of the issue. This suggests that the partial information reports are mostly non-issues in that they cause no harm or are relations that do not need to be in the model.

5.2 *Mock Consistency Check*

A deployed WIKID setup can only realistically discover existing problems within the information it is given, and it seems intentionally breaking systems and servers within the testing deployment on *The Tardis Project* would be an unpopular way of evaluating the coverage of WIKID's abilities.

Instead, a simulated pair of graphs, G and H are constructed by a testing *pywikid* service called '*Handle*', and fed into the database. Handle generates two pseudo-random graphs with 9000 vertices and $20,000$ edges of 100 different types, with the aim of making them deliberately inconsistent. We then verify that WIKID correctly identifies these inconsistencies.

The edges are generated randomly so it is likely that completely disconnected vertices exist in both graphs, and that there are some vertices of high degree (i.e. have many edges).

The second of the two graphs, H, is a random perturbation of G, where a 10% change of error has been introduced: With a 5% probability, the type of an edge was swapped to another randomly sampled from the edge types, and with a 2.5% probability in each

direction, the source or destination vertex of an edge was changed to a randomly re-sampled vertex.

Table 5.2: Parameters and performance of Mock Consistency Check evaluation

Parameter	Value	
Pseudo-random seed	`z_mEebrcXATwuSR`	
Number of edges, $\|E\|$	20,000	
Number of edge types, $\|T\|$	100	
Number of vertices, $\|V\|$	9000	
Time to generate graphs G & H	4.537	seconds
Time to build graph in database	33.724	seconds
Number of edges swapped	1031	(5.155%)
Number of vertices swapped	1000	(5.000%)

Conch, the WIKID consistency checker states that G and H now have 8879 mutual vertices, meaning that when performing swaps, 121 common vertices were lost.

The full consistency check of G against H took 5 hours and 10 minutes to complete with a single *Conch* instance. It published 1123 reports (§4.3.2) of inconsistencies, 939 *strong* (full disagreement) and 184 *partial* (potentially missing), representing a total realised error of 5.615%. This is actually what we should expect, if not a little higher – while we introduced a randomised 10% permutation to the H graph, we consistency checked from the *perspective* of the assumption that one of the graphs was a prior truth, the other the affected graph. This means the consistency check was not concerned with roughly half of the different errors because they were changed in a way that was *not* counterfactual to the prior. Performing a second consistency check setting the other graph as affected would turn up the remaining $\sim$ 5% error.

5.2.1 *Discussion*

The scale of this test served to stress the system. The magnitude of the graphs simulate a substantial datacentre, and the 10% rate of error is significantly larger than one would expect to see in a real production environment.

With these large models, the time complexity of the consistency check it the dominating performance bottleneck, rather than memory usage or model construction time. At 9000 vertices with 20,000 edges, the number of operations *Conch* must perform is huge, on the order of $O(|V_c|^2)$ in the worst case, which in this example would be $8879^2 = 78,836,641$ operations. This seems to be necessitated by the task of consistency checking, i.e. searching for a counterexample to the homomorphism hypothesis which requires iteration over the entire graph.

Clearly there is room for improvement. As it stands, each graph

is checked serially by a *Conch* instance, so while multiple instances can parallelise checking across multiple graphs, within a graph the check is performed serially. On a graph as large as the one in this evaluation, the time this takes is prohibitively long. It is possible to parallelise the work of checking within the *Conch* microservice itself, using multiprocessing features. This parallelism would increase the load on Redis but a sufficiently large cluster would have no issues handling it. With further work we could also parallelise consistency checking work within the same graph out to multiple *Conch* instances, though this would require significant rework.

5.3 *Remarks on Production Readiness*

Earlier in CH.4 it was outlined that an implementation and design objective was building a production ready tool. The deployment to *The Tardis Project* demonstrated an ability to assist in Administration maintenance and pro-actively fixed misconfigurations (assumedly) before they caused a real problem. To pro-actively identify misconfigurations was one of the aims of the project. However, testing on very large scale simulated systems in §5.2 indicate that optimisations need to be made for WIKID to be performant at large scales.

The WIKID setup on *Tardis* did demonstrate promising system stability – the Redis cluster remained stable without any master failovers or instance crashes, and all of the WIKID services ran continuously for *at least* 22 days. While this isn't a long enough period to make strong claims about service stability, it does suggest that the *pywikid* services aren't prone to crashing.

Recall that WIKID services are free to join and part from the cluster as they wish; In some environments where WIKID needs to be access controlled, this might be unacceptable. Implementation of an API key authentication system would solve this. *Redis Cluster* is easily secured by enabling password protection and disabling sensitive commands. It would still not be recommended to expose the Redis cluster to the internet given the traffic over both the client and cluster communication busses is unencrypted.

6

CONCLUSION

6.1 Introspection & Limitations

The *DNS & NetBox* services seem to have been a good pair to implement for *The Tardis Project*'s WikiD deployment. They served as a proof of concept, demonstrating the feasibility of a system like WikiD and it's usefulness for pre-emptively correcting errors in configurations across different sources. However, within the implementation I didn't delve into integrating configuration languages and applying WikiD to them. While we are in the process of deploying configuration management to *The Tardis Project*, it is not *currently* available as a source of information and not worth the time it would have taken to deploy just for use with WikiD.

The WikiD configuration model, presented §3.1 seems adept at modelling a fair amount of information, but could still be improved. We can't encode a relational calculus for the relations we store. For example, the equivalence $A_{\ x \mapsto y}, B_{\ y \mapsto z} \equiv C_{\ x \mapsto z}$ is useful information that would allow compactification of edges together and give us further ability to check consistency. Building this functionality in presents a difficult design problem, bearing in mind one of the aims of WikiD is to remain as generalised as possible; A solution would almost certainly have to implement an interface for adding rules into a centralised calculus store. This would probably be a complex feature, and could have a negative effect on performance.

We also have no concept of negative relations, though as touched on in §3.4, whether we'd actually need negation is unclear. At worst, we can assume it is safe to ignore or model around this need... Potentially, we could implement negation within a relational calculus as described above.

As noted in §5.1, the *Potential Missing Information* reports that are generated when a partial inconsistency is found were largely discarded. The rationale behind presenting them to administrators is that they *could* represent a problem, within the contexts of the data

source. Still, this high rate of reporting when no action need be taken prompts a rethink of how partial inconsistencies should be handled.

Sentry was a convenient service that I could integrate WIKID into, and avoid diverting efforts into building a Web UI. The WIKID use case is not *really* what Sentry was designed for, though having spend time used Sentry to digest the reports I'd surmise that it is sensible method of displaying the information exiting WIKID.

6.2 Future Work

I'd aim to implement, test and deploy further integrations on *The Tardis Project*, expanding into configuration languages, system setting and completing those that are already partially implemented. From the evaluations in this report, future work on improving consistency checking with edge compactification and parallelisation are clear progressions forward.

Further exploration of how we can present administrators with the information that leaves WIKID, beyond using *Sentry*, including a purpose-built interface.

6.3 Final Conclusions

SYSTEM SOFTWARE CONFIGURATION was one of the main motivations behind WIKID alongside infrastructure and networking orchestration. A lot of these motivations came from being an administrator on *The Tardis Project*.

The abstracted model of system states designed for WIKID, and the implementation of a consistency checker over these models has shown to be a viable tool, helping identify errors in the environments it has been tested in. This seems an interesting outcome given WIKID represents a new approach to System Administration and Infrastructure Management.

Furthermore, the tool can identify all errors that are presented it via the models it is fed. This addresses the void articulated earlier, between software configuration and documentation. It also is an administration tool that well complements, and even itself makes use of, traditional management tools in widespread usage. The system that has been built around WIKID has adhered to the implementation aims set out earlier, and is already a relatively mature core software package.

BIBLIOGRAPHY

Paul Anderson. Towards a high-level machine configuration system. In *LISA*, volume 94, pages 19–26, 1994.

Johan Andersson (Grokzen). Redis-py-cluster. URL `https://github.com/Grokzen/redis-py-cluster`. A Python Client for Redis Cluster.

Salvatore Sanfilippo et al. Redis cluster specification. *Redis Labs*, 2014. URL `https://redis.io/topics/cluster-spec`.

Vince Fuller, Tony Li, Jessica Yu, and Kannan Varadhan. Rfc-1519: Classless inter-domain routing (cidr): an address assignment and aggregation strategy. Technical report, 1993.

DigitalOcean Inc. Netbox. URL `https://netbox.readthedocs.io/en/latest/`. Tool to help manage and document computer networks and infrastructure.

Leena Joshi. Redis usage survey. *Redis Labs*, 2016. URL `https://redislabs.com/blog/the-results-are-in-redis-usage-survey-2016/`. Found that 67% of Redis users used it for data not stored in any other database.

Zeus Kerravala. As the value of enterprise networks escalates, so does the need for configuration management. *The Yankee Group*, 4, 2004.

Andy McCurdy. Redis-py. URL `https://github.com/andymccurdy/redis-py`. A Python Client for Redis.

Paul V Mockapetris. Rfc 1035 domain names: Implementation specification, section 3.5 in-addr.arpa domain. Technical report, 1983.

Denis Ovsienko, Aaron Dummer, Alexey Andriyanov, and Arnaud Launay. Racktables. URL `http://racktables.org/`. A system for documenting hardware assets, network addresses, rackspace, network configuration and more.

Stephen Quinney. System configuration: An end to hacky scripts? *UKUUG*, 2008. URL `http://www.lcfg.org/doc/sysconfig_ukuug_2008.pdf`.

Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

Yakov Rekhter and Tony Li. Rfc-1518: An architecture for ip address allocation with cidr. Technical report, 1993.

Salvatore Sanfilippo (antirez). On redis, memcached, speed, benchmarks and the toilet. URL `http://oldblog.antirez.com/post/redis-memcached-benchmark.html`.

Salvatore Sanfilippo (antirez) and Redis Labs Inc. Redis. URL `http://redis.io`. Redis is an open source in-memory data structure store. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial hashes.

University of Edinburgh The Tardis Project. The tardis project. URL `https://wiki.tardis.ed.ac.uk`. The Tardis Project is a computing facility, run and maintained by students of The University of Edinburgh.

György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

Guido van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8–style guide for python code. *Available at ht tp://www. python. org/dev/peps/pep-0008*, 2001.

Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015.

LIST OF FIGURES

3.1 Graph Definition Semantics of the WIKID abstract model *(19)*

3.2 Illustrative example of a *Directed Multigraph* showing an arbitrary set of relational edges and vertices. *(19)*

3.3 Two graphs, G and H with 6 common vertices and some common edges. *(21)*

3.4 Two graphs, G and H showing only common edge. Vertex 4 is disconnected. *(22)*

3.5 Inconsistencies between G and H highlighted and the difference graphs $\hat{G}$ and $\hat{H}$. *(23)*

4.1 Service-level block diagram of the WIKID architecture *(26)*

4.2 A possible allocation of WIKID and *Redis* services to physical machines. *(27)*

4.3 An example multidigraph *(30)*

4.4 WIKID Service types for the services implemented, both as standard and for *The Tardis Project*. *(34)*

4.5 The *NetBox Devices Page*, showing physical machines and groupings within *The Tardis Project*. *(40)*

4.6 *Sentry.io Dashboard*, showing unresolved issues from WIKID. *(43)*

4.7 *Sentry.io Report*, showing a conflict between two graphs, *DNS* and *Reverse DNS* (Both from the *DNSDisco* service). *(44)*

LIST OF TABLES

4.1 WikiD Standard report structure, as generated by *Check* type services. *(28)*

4.2 Redis commands to build the graph in Fig.4.3 *(32)*

4.3 Example tableau and message for a report from the *Sentry* Service. *(42)*

5.1 Showing the tabulated co-incidence matrices of inconsistencies between the deployed WikiD services on *The Tardis Project* *(45)*

5.2 Parameters and performance of Mock Consistency Check evaluation *(47)*

APPENDIX

Welcome to WikiD's documentation!

Contents:

Indices and tables

- Index
- Module Index

class **pywikid.Service**(*conf={}*)

Generic Microservice, implements communication with Redis et al and helper functions that most services are likely to use.

As a convention, all methods are sunders or dunders, so extending classes won't accidentally override their functionality.

`__str__` and `__repr__` give unique identifier for this instance that should not collide locally or over the network.

Member r: Redis API instance

Member log: Logging class instance, additional setup may be needed

Member namespace:
Auto-generated string containing the class' Redis namespace

Configuration:

Two configuration files can be created, *config.yml* and *secrets.yml*. Config will be loaded and dictionary merged with the supplied *conf* argument, with the arguments taking precendence. The secrets file is seperate allowing it to be ignored by VC, and is loaded into a private dict.

Redis:

Using a redis cluster, which has some limitations over monolithic Redis, Including the exclusion of password auth.

The redis channel we announce to (using self.announce) is set with `conf['announce_channel']` which defaults to 'events'

Sentry:

If raven is installed, *enable_sentry* is *True* in config, *and* a Sentry DSN is in secrets, with keys *sentry_host*, *sentry_id*, *sentry_public_key* and *sentry_private_key* then Raven will be configured and throw any encountered exceptions at Sentry.

Usage:

Services should usually implement a *__call__* method that itself calls *daemon_loop*. Wrapping this in a *@daemonizer.run(...)* decorator from the module *deamons* is useful.

If a service wishes to populate a graph that should be checked against the others, it **must** register it with the *register* method.

Queues:

Discovery type services should enqueue a consistency check of their graph after making changes to it, using the `enqueue` method like so:

```
` s = Service() s.enqueue('graphcheck', {'affected': s.graph_namespace}) `
```

Similarly, 'Check' type services should report any issues they find using the *report* method, by passing it an instance of the `Report` class.

'Check' type services can get jobs to run using the *getjob* method, which blocks until a job is available. If the job is executable, it should run it then clear it from the processing queue using *jobdone*. If it is **not** executable, it should *rescind* the job and probably wait a bit befor looking for a job again. *Note* that this means only one worker for each job type will see the job, so if multiple checks must happen then multiple jobs of different job types should be enqueued.

Lastly, 'Export' type services should listen for reports using *get_report*, and do with the report as they please. At the moment only one exporter makes sense.

__call__()

The main operation of the service, expected to be implemented by a subclass. Callable, in almost all cases should be wrapped in `@daemonizer` or call something that is.

__eq__(*other*)

Equality check on two instances of the Service class, True if class names match.

Parameters: **other** (*pywikid.Service*) – Other (not self) object to compare equality with

Returns: True or False

__init__(*conf={}*)

Initialise Service Instance.

Arguments:

Parameters: **conf** (*dict*) – A dict containgin config variables, which will vary depending on the particular derivative of this generic class.

__repr__()

Provide qualifying information about who we are

Returns: string representation of self,with PID and hostname

__str__()

Provide qualifying information about who we are

Returns: string representation of self,with PID and hostname

__weakref__

list of weak references to the object (if defined)

_flush_reports()

Clear pending & completed reports

announce(*a*)

Publish standardised message to Redis with identifier, which is also stored in a sorted set of the same name

Parameters: **a** – Announce this string or formattable type

daemon_loop(*loop_f*)

Run a function in a loop, and redirect output descriptors to a file or to /dev/null if we can't open a logfile

encode_job(*name*, *data*)

Encode a job

enqueue(*job_name*, *job_data*)

Add a job to the jobqueue

genkey(*k*)

Utility method, takes a plain key and prepends the namespace, or a list of keys and generates the key heirarchy.

Parameters: **k** (*list(str) or str*) – Strings in list will be joined on thestandard key delimiter.

Returns: A string that represents the safe key adhering to WikiD's Redis key convention.

get_completed_reports(*length=-1*, *start=0*)

Return a list of processed reports len *length*

get_report()

Blocking report fetch from fresh queue Trims reports list to at most 200 elements

get_reports(*length=-1*, *start=0*)

Return a list of processed reports len *length*

getjob()

(Blocking) Take next job from job queue, add to processing queue

jobdone(*job_name*, *job_data*)

remove a job from the processing queue

parse_job(*name*)

Take a name from encoded jobqueue and return parts

raise_for_config(*e*)

Helper function, will add information to exception (usually KeyError or IndexError) for the context of it being config that should be in a config.yml

report(*rpt*)

Produce a persistent event object that will be used by export services

rescindjob(*job_name*, *job_data*)

Put job back on jobqueue.

setup(*conf*)

Optional hook into the end of `Service.__init__()` that an extending class may implement.

Parameters: **conf** (*dict*) – Same config dict as passed to __init__(), will automatically be passed.

sleep()

Sleep for pre-set interval

trygetjob()

Take next job from job queue, add to processing queue

class **pywikid.Report**(*issue*, *level*, *prior_graph=None*, *prior_edges=None*, *altered_graph=None*, *altered_edges=None*, *commonalities=None*, *vertices=None*, *message=None*)

Standardised object pushed out as a report from checking Serivces to Output Services.

__str__ dumps as a key-sorted JSON str

__init__(*issue*, *level*, *prior_graph=None*, *prior_edges=None*, *altered_graph=None*, *altered_edges=None*, *commonalities=None*, *vertices=None*, *message=None*)

__str__()

Serialize data

__weakref__

list of weak references to the object (if defined)

class **pywikid.Watchdog**(*conf={}*)

Watches over WikiD itself, monitoring services and databases (esp. Redis)

cluster_status()

Call and accregate all Status info for the cluster

dbsizes()

Calculates the average database size on each node, and the standard deviation which is indicative of shardedness

fold_dict(*d*)

Takes a dictionary with a set of sub-dicts, all having the same shape and folds into a dict of the same shape.

main()

Fetch cluster statistics

nodes_list_to_dict(*nodes*)

Take [dict, dict, dict] and produce a dict

timeskew(*times*)

Calculate mean cluster time, varience and per-node skew

Parameters: **times** (*dict*) – output in the style of :code `Redis.time()`

Returns: tuple(float, float, dict)

Raises: **ZeroDivisionError** – Could div by 0 if len(times) = 0

class **pywikid.Trawler**(*conf={}*)

Iterates graphs, destroying old stuff.

__call__()

Wait for a job indefinately

dangle()

Remove edges that got to non-veritces

evict()

Evict Stale Data from the graphs

Looks at the *seen* member in vertices, and the age of a given graph

log_eviction(*graph*, *vertex*, *graph_age*, *vertex_seen*)

Log an eviction

main()

Run through tasks

setup(*conf*)

Service.__init__ hook

trawl_summary()

Publish a report detailing actions taken if any

class **pywikid.Conch**(*conf={}*)

Consistency checking class and other WikiD job runner and cleanup stuff

__call__()

Wait for a job indefinately

gcheck(*data*)

Graph consistency checking implementation

main()

Fetch a job and call handler

setup(*conf*)

Service.__init__ hook

class **pywikid.GraphRedis**(*namespace='default'*, **args*, ***kwargs*)

Extend Redis (clustered) with support for directed multigraphs by adding functions over standard redis types.

As a convention all commands are prefixed with 'g', and follow the Redis RESTful way of behaving, hence no state infromation in this class.

Methods making use of Lua scripts will cache the script upon first use.

Methods are not necessarily atomic, so for any query that needs to be tranactional and have guaranteed atomicity, use a Pipeline.

The class also handles naming conventions for keys that are separate Redis entities but the same logical graph component.

Hash tags will be used if two keys need to be used together in a query (such as set union)

_UNSAFE_del_lock(*name*)

Will delete lock, freeing it if it was not released by a holder. Use with an excess of caution

_directed_gedge(*origin*, *dest*)

Return all edges from origin to dest

_directed_gedge_iterator(*origin*, *dest*)

SCAN iterator over edges

_get_edgekeys(*vertex=None*)

Generate a set of Redis keys corresponging to all edges involving vertex `vertex`

Uses *SCAN* so the guarantees about behaviour are weaker than perhaps expected. See Redis.io docs for more .. _SCAN: https://redis.io/commands/scan

Not really atomic, though each SCAN guarantees every key that matched as we began will be returned

_get_lock(*name*, *timeout=5*)

Generate a lock help serialise multi-key ops, given the cluster can't tolerate EVAL or MULTI with cross-shard key dependencies.

Blocking:

Will block until lock is acquired.

Atomic:

Locking class returned operaties atomically

Wrapping code that depends on a lock in a *try*, releasing in *finally* will prevent permanent lock holding.

_parse_edge_key(*edgek*)

Take a key of the form `:_adjacency:source:dest` and return *source* and *dest* as a tuple.

_parse_vertex_key(*vertk*)

DEPRECATED

Take a key of the form `:_vertices:name` and return *name*

flushgraph()

Delete this graph (be careful, ofc)

Un-registers immediately and then scans through namespace deleting keys

gadd(*vertices*, *edges*)

Joint method, add vertices and edges to a graph.

Parameters:
- **vertices** (*list((str, dict))*) – Vertices to add. First part of tuple is the name, second the data.
- **edges** (*list(tuple(str,str,str))*) – Edges to add. Tuple is (source vertex, dest vertex, edge label)

geadd(**edges*)

Add a set of edges. Edges are directed and stored in an adjacency martix, with edge labels being held in sets. Vertices are created if they don't exist.

Parameters: **edges** (*list(tuple(str,str,str))*) – edges involving this vertex, with labels

gedge(*source*, *dest*)

Return the edge types between source and dest

Parameters:
- **source** – Source Vertex
- **dest** – Destination Vertex

Returns: set of edge labels between source and dest, or empty set if none.

gedgerem(*source*, *dest*, *label*)

Remove edge from source to dest with label

gedges(*vertex=None*)

Return all edges involving a given vertex

Parameters: **vertex** (*str*) – Vertex to get edges of

get_age()

Return the age of a graph

ginedges(*vertex*)

Return all the edges *incoming* to a given vertex.

goutedges(*vertex*)

Return all the edges *outgoing* from a given vertex

graphs()

Return all graph names

graphs_iter()

Return iterator over graphs

gvadd(*vertex*, *data*)

Add a vertex to the graph, with data

Parameters:
- **vertex** (*str castable*) – Vertex to add to
- **data** (*dict*) – data to associate with vertex

gvcard(*vertex*)

Returns the cardinality of a vertex (number of edges)

gvertex(*vertex*)

Return the data at *vertex*

gvertices()

Return a set of all vertices

gvertices_iter()

Return an iterator to all vertices

gvrem(*vertex*, *justdata=False*)

Remove a vertex from the graph

If *justdata* is `True`, only the vertex's data will be removed and all edges to and from it will persist

Not the fastest operation in the universe... Also super not-atomic...

pipeline(*transaction=True*, *shard_hint=None*)

Return a new pipeline object that can queue multiple commands for later execution. Bing clustered Redis, this is more of a compatability method and a pipeline does *NOT* guarantee transactional atomicity.

register()

Register as a globally accessible graph. **MUST** be called if a service wants it's graph to interact with others.

switchspace(*newspace*)

Change a GraphRedis instance's namespace, by regenerating key prefixes

update_age(*age=None*)

Update the age of the graph after a recent fetch. If *age* is `None` the current time is used.

withdraw()

Opposite of register, removes from global graphs

class **pywikid.StrictGraphPipeline**(*connection_pool*, *result_callbacks=None*, *response_callbacks=None*, *startup_nodes=None*)

Pipeline for the GraphRedis as an extension of the StrictPipeline and StrictRedis classes