

[University Name]
[Faculty / Department of Computer Science]

Schema-Guided Breadth-First Discovery of Hardware Resources in Heterogeneous Redfish Services

Master's Thesis

[Author Name]

Supervisor: [Supervisor Name]
Submitted: [Month Year]

Abstract

The DMTF Redfish standard replaced the Intelligent Platform Management Interface (IPMI) with a RESTful, hypermedia-driven interface to the Baseboard Management Controller (BMC) of a server. In principle a client needs to know only one URI, the service root `/redfish/v1`; every other resource is reached by following links. In practice, monitoring tools are written against a fixed set of URI paths, and those paths differ between vendors, firmware generations and schema releases: power supplies live under `Chassis/{id}/Power` in one server and under `Chassis/{id}/PowerSubsystem/PowerSupplies` in the next; drives can hang below `SimpleStorage`, `Storage` or directly below the chassis. Hard-coded navigation therefore silently loses components whenever the monitored fleet changes.

This thesis treats the Redfish service as what the standard says it is, a graph of typed resources, and turns the question “which URI do I call for the power supplies?” into a graph search. The resource graph is restricted to its containment edges, which yields an almost-tree rooted at the service root, and the monitored components are described declaratively by their schema types (e.g. `PowerSupply`) rather than by their location. A breadth-first search (BFS) over this tree finds every instance of the requested types regardless of where a vendor placed them. To make the search affordable on slow BMCs, the thesis derives a *type reachability graph* from the published DMTF JSON Schema bundle and uses it to prune, before any request is sent, every subtree that cannot contain a requested type. The resulting algorithm, schema-pruned BFS (SP-BFS), is proven sound for schema-conformant services, falls back to conservative expansion for vendor extensions, handles legacy embedded arrays, and exposes level-synchronous parallelism.

The approach is evaluated on 26 recorded Redfish services from six sources (DMTF reference mockups, Dell iDRAC, HPE iLO, HPE Cray EX, Gigabyte and Intel) spanning Redfish protocol versions 1.0.2 to 1.20.0. Hard-coded navigation written for one schema generation recovers only 49 % (modern paths) or 78 % (legacy paths) of the 2,248 monitored components; even a hand-maintained union of both misses storage devices on three HPE servers. SP-BFS recovers 99.7 % of the components, exactly as many as an exhaustive tree traversal, while issuing $3.9\times$ fewer HTTP requests in total ($5\times$ in geometric mean per server). Its request count is within a factor of 1.22 (geometric mean) of the hand-maintained

union, without any vendor-specific code. With 16 concurrent requests the discovery of all servers fits into 333 request rounds instead of 3,461. The thesis concludes that schema-guided graph search is a practical way to make Redfish monitoring robust against vendor and version heterogeneity.

Keywords: Redfish, out-of-band management, Baseboard Management Controller, hardware monitoring, REST, hypermedia, breadth-first search, schema-guided traversal, interoperability.

Contents

Abstract	i
List of Abbreviations	ix
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Research Questions	2
1.4 Contributions	3
1.5 Thesis Outline	3
2 Background	4
2.1 Out-of-Band Server Management	4
2.2 The Redfish Standard	5
2.2.1 Protocol	5
2.2.2 Resource model	6
2.2.3 Schemas and versioning	7
2.3 Breadth-First Search	7
3 Related Work	8
3.1 Redfish-Based Monitoring	8
3.2 Redfish Client Libraries and Tools	9
3.3 Hypermedia Clients and Web Crawling	10
3.4 Web API Evolution	10
3.5 Summary	11
4 Problem Analysis and Formal Model	12
4.1 The Resource Graph	12
4.2 The Containment Tree	13
4.3 Resource Queries	13
4.4 Heterogeneity of Redfish Services	15

5	Schema-Guided Breadth-First Discovery	17
5.1	Overview	17
5.2	The Type Reachability Graph	18
5.3	The SP-BFS Algorithm	20
5.4	Soundness and Necessity	20
5.5	Cost	22
5.6	Level-Synchronous Parallelism	22
5.7	Discovery within a Monitoring Loop	23
6	Implementation	25
6.1	Architecture	25
6.2	The rfbfs Package	25
6.3	Engineering Aspects	27
6.4	Evaluation Harness	28
7	Evaluation	29
7.1	Experimental Setup	29
7.1.1	Services	29
7.1.2	Ground truth	30
7.1.3	Strategies	30
7.1.4	Metrics	31
7.2	RQ1: Heterogeneity and Hard-Coded Navigation	31
7.3	RQ2: Completeness of Type-Based Tree Search	34
7.4	RQ3: Cost of Schema-Pruned Search	35
7.5	RQ4: Latency, Parallelism and Amortisation	37
7.6	Shape of Redfish Resource Trees	39
7.7	Discussion	40
7.7.1	URI patterns versus type-based search	40
7.7.2	Why not follow Links or use \$expand?	40
7.7.3	Threats to validity	40
8	Conclusion	42
8.1	Summary	42
8.2	Limitations	43
8.3	Future Work	43
A	Static Navigation Recipes and Queries	51

B	Reproducibility	53
B.1	Data sources	53
B.2	Commands	53

List of Figures

4.1	Placement heterogeneity in three recorded services	16
5.1	Schema pruning on the upper levels of a service	18
6.1	Architecture of the monitoring tool. Only the discovery component (blue) interprets the Redfish resource graph.	26
7.1	Placement templates per service	33
7.2	Micro-averaged recall per vendor family (n = number of services).	34
7.3	Micro-averaged recall per component category.	35
7.4	Requests per service	36
7.5	Cost trade-off (a) and level-synchronous parallelism (b), summed over all 26 services.	37
7.6	Requests until the first answer	38
7.7	Shape of the containment trees	39

List of Tables

4.1	Resource queries of the monitoring tool. φ is true unless stated.	14
7.1	Services, recall and request counts	32
7.2	Micro-averaged recall per category and overall.	33
7.3	Most frequently pruned (type, property) pairs, summed over all services. .	37
B.1	Data sources (GitHub repositories) and the commits used.	53

List of Algorithms

1	Construction of the type graph H from a JSON Schema bundle	19
2	Schema-pruned breadth-first discovery (SP-BFS)	20
3	Monitoring loop with event-driven rediscovery	24

List of Abbreviations

API	Application Programming Interface
BFS	Breadth-First Search
BMC	Baseboard Management Controller
CSDL	Common Schema Definition Language (OData)
DCIM	Data Center Infrastructure Management
DFS	Depth-First Search
DMTF	Distributed Management Task Force
DSP	DMTF Standard Publication
HATEOAS	Hypermedia as the Engine of Application State
HPC	High-Performance Computing
HTTP(S)	Hypertext Transfer Protocol (Secure)
iDRAC	integrated Dell Remote Access Controller
iLO	HPE Integrated Lights-Out
IPMI	Intelligent Platform Management Interface
JSON	JavaScript Object Notation
MIB	Management Information Base
OData	Open Data Protocol
OEM	Original Equipment Manufacturer
PSU	Power Supply Unit
REST	Representational State Transfer
SNMP	Simple Network Management Protocol
SP-BFS	Schema-Pruned Breadth-First Search (this thesis)
URI	Uniform Resource Identifier

Chapter 1

Introduction

1.1 Motivation

Every modern server contains a small, independent computer, the *Baseboard Management Controller* (BMC), that watches the health of the machine even while the host operating system is down. It reads temperature and voltage sensors, drives the fans, records hardware faults and lets an operator power-cycle the machine remotely. For two decades the dominant interface to the BMC was the Intelligent Platform Management Interface (IPMI) [36], a binary protocol with a flat sensor model and a well-documented list of security weaknesses. In 2015 the Distributed Management Task Force (DMTF) published *Redfish*, a successor built from web technology: HTTPS, JSON and a REST-style resource model described by machine-readable schemas [24, 25]. Redfish is now supported by every major server vendor and by the open-source OpenBMC firmware, and it is the basis of monitoring pipelines in data centres and high-performance computing (HPC) sites [2, 14, 38].

The appeal of Redfish for a monitoring tool is obvious: instead of vendor specific command-line utilities or SNMP MIBs, a single HTTP client can ask any server for its processors, memory modules, drives, network adapters, power supplies and fans. The author experienced the other side of this promise while building a server hardware monitoring tool on top of Redfish. The standard fixes the *types* of resources and their properties, but it leaves the *arrangement* of the resources largely to the implementation, and the schema itself has evolved substantially:

- A Dell PowerEdge server with an iDRAC running Redfish 1.0.2 reports its power supplies as an array inside `/redfish/v1/Chassis/System.Embedded.1/Power`; the DMTF reference servers of the 2026 schema bundle additionally expose `/redfish/v1/Chassis/1U/PowerSubsystem/PowerSupplies/{id}`, because the `Power` resource was deprecated in favour of `PowerSubsystem` in Redfish release 2020.4.

- Physical drives appear below `Systems/{id}/SimpleStorage` (as embedded Devices), below `Systems/{id}/Storage/{id}/Drives`, or directly below `Chassis/{id}/Drives`, depending on vendor and generation.
- Resource identifiers are chosen by the implementation (1, System.Embedded.1, 437XR1138R2, Self, Node0), so even the same path template cannot be written down as a literal URI.

A tool written against one server therefore returns incomplete data on the next one, and it usually does so *silently*: a missing link is indistinguishable from a server that simply has no such component. Deciding “which REST API has to be called for storage, network, power or processors” turned out to be the hardest part of building the tool, and every newly supported server model required new special cases.

The solution that emerged in the tool, and which this thesis develops, formalises and evaluates, is to stop encoding *where* a resource lives and to describe only *what* it is. The Redfish service is converted into a tree of resources rooted at `/redfish/v1`, and the resources of interest are found by a breadth-first search (BFS) that matches the `@odata.type` of every visited node. The standard explicitly supports this view: Redfish is a hypermedia API in the sense of Fielding’s REST constraints [31], in which clients are expected to discover resources by following links rather than by constructing URIs.

1.2 Problem Statement

A naive graph search, however, is expensive. BMCs are embedded processors with limited CPU and memory; a single Redfish GET often takes tens to hundreds of milliseconds, and some implementations throttle concurrent sessions. The services studied in this thesis contain between 68 and 1,811 resources, most of which (log entries, JSON schema files, message registries, account and session objects) are irrelevant to hardware monitoring. Crawling the whole tree to find two power supplies is wasteful and slow. The problem addressed by this thesis is therefore:

How can a monitoring client locate all instances of a given class of hardware resource in an arbitrary, possibly non-uniform Redfish service, without vendor-specific navigation code and with a number of requests close to that of a hand-written, service-specific client?

1.3 Research Questions

RQ1 How heterogeneous are real Redfish services with respect to the placement of monitoring-relevant resources, and how much do hard-coded navigation paths

miss?

- RQ2** Does a type-based breadth-first search over the containment tree find the monitored components completely, independent of vendor and schema version?
- RQ3** Can the published Redfish schema be used to prune the search so that its request cost approaches that of hand-written navigation, without losing completeness?
- RQ4** What are the latency characteristics of the search (time to first result, parallelism across tree levels), and how does its one-time cost amortise over a monitoring session?

1.4 Contributions

1. A **formal model** of a Redfish service as a typed, labelled resource graph whose containment edges form an almost-tree, together with declarative *resource queries* that name component classes by schema type, embedded-array location and content predicate (Chapter 4).
2. **Schema-pruned BFS** (SP-BFS), a discovery algorithm that combines breadth-first traversal with a type reachability graph extracted automatically from the DMTF JSON Schema bundle. We prove that it is sound, i.e. loses no target that an exhaustive tree traversal finds, for any schema-conformant service, and we analyse its cost and its level-synchronous parallelism (Chapter 5).
3. An **implementation** as the discovery layer of a monitoring tool, separating a one-time discovery phase from a cheap polling phase with event-driven rediscovery (Chapter 6).
4. An **empirical study** on 26 recorded Redfish services from six vendor families and seven protocol versions, which quantifies placement heterogeneity and compares SP-BFS with hard-coded navigation and with exhaustive BFS/DFS (Chapter 7). All code, data and figures are reproducible from the accompanying repository.

1.5 Thesis Outline

Chapter 2 introduces out-of-band management, the Redfish data model and graph search. Chapter 3 reviews prior work on Redfish-based monitoring, hypermedia clients, web crawling and API evolution. Chapter 4 formalises the discovery problem and characterises the heterogeneity of real services. Chapter 5 presents SP-BFS and its analysis, Chapter 6 the implementation, and Chapter 7 the evaluation. Chapter 8 concludes and outlines future work.

Chapter 2

Background

This chapter summarises the technical foundations the thesis builds on: out-of-band server management and its protocols (Section 2.1), the Redfish resource model and its versioning rules (Section 2.2), and breadth-first search (Section 2.3).

2.1 Out-of-Band Server Management

Out-of-band management refers to monitoring and controlling a computer through a path that does not depend on its host operating system. In servers this path is provided by the BMC, a service processor on the motherboard with its own network interface, firmware and power domain. The BMC is connected to the platform’s sensors (temperatures, voltages, fan tachometers), to the power sequencing logic and, increasingly, to management buses of add-in devices such as network adapters, storage controllers and GPUs. Vendor BMC products include Dell’s iDRAC, HPE’s iLO and Lenovo’s XClarity Controller; OpenBMC is a Linux Foundation project providing an open-source BMC firmware stack [46].

IPMI. The Intelligent Platform Management Interface, specified by Intel, HP, NEC and Dell in 1998 and last revised as version 2.0 in 2004 [36], defines a binary request/response protocol (with the RMCP+ transport over UDP) and a flat model of *sensor data records*. IPMI has three properties that motivated its replacement: its data model cannot describe modern component hierarchies (multi-node chassis, NVMe drives, GPUs); its commands are extended through vendor-specific OEM codes; and BMC firmware built around it has a history of serious security weaknesses that expose the whole server to attackers on the management network [6].

SNMP. Many data centres monitor hardware via the Simple Network Management Protocol [9, 33], where a vendor MIB exposes BMC or host-agent data as a tree of object identifiers. MIB trees are statically numbered and vendor specific; a monitoring tool needs

one MIB per vendor. Interestingly, the SNMP GetNext walk is itself a tree traversal, an idea we revisit for Redfish.

2.2 The Redfish Standard

Redfish is developed by the DMTF Redfish Forum (formerly the Scalable Platforms Management Forum). Version 1.0 of the protocol specification DSP0266 was released in 2015 [15]; the protocol and its schemas have since been extended several times a year [24, 25, 27]. The Storage Networking Industry Association (SNIA) extends Redfish for storage systems under the name Swordfish [50].

2.2.1 Protocol

A Redfish service is an HTTPS server, usually running on the BMC. Its service root is always `/redfish/v1`; all other URIs are *opaque* to clients, which are expected to find them by following links. Resources are JSON documents [7] retrieved with GET; they are changed with PATCH, created in collections with POST, and *actions* (e.g. `ComputerSystem.Reset`) are invoked by POST on action targets. Authentication uses HTTP Basic authentication or session tokens (`X-Auth-Token`). Besides the network path, the Host Interface specification lets software on the host reach the same service in-band [21], and PLDM for Redfish Device Enablement lets add-in devices such as network and storage controllers contribute their own resources to the BMC's tree [22]; the latter is one reason why the shape of the tree depends on the installed hardware. The DMTF's Redfish User Guide describes typical client workflows on top of these mechanisms [23]. The protocol borrows its conventions from the OASIS Open Data Protocol (OData) version 4 [20, 48, 49]:

- `@odata.id` is the URI of a resource, and a JSON object of the form `{"@odata.id": "/redfish/v1/S..."} is a reference to another resource;`
- `@odata.type` is the fully qualified, versioned type of a resource, e.g. `#Processor.v1_7_0.Processor`;
- a *resource collection* has a `Members` array of references and a `Members@odata.count`; long collections are paged with `Members@odata.nextLink`;
- query parameters such as `$expand`, `$select` and `$filter` are optional features that a service may advertise in its `ProtocolFeaturesSupported` object.

Listing 2.1 shows a processor resource from a DMTF reference mockup. Its link `EnvironmentMetrics` leads to a subordinate resource, whereas `OriginOfCondition`, nested in the status conditions, merely points to the sensor that raised a warning and lives elsewhere in the hierarchy.

Listing 2.1: Abridged Processor resource (DMTF mockup public-rackmount1, schema bundle 2026.2).

```
{
  "@odata.id": "/redfish/v1/Systems/437XR1138R2/Processors/CPU1",
  "@odata.type": "#Processor.v1_24_0.Processor",
  "Id": "CPU1",
  "Socket": "CPU 1",
  "ProcessorType": "CPU",
  "TotalCores": 8,
  "Status": {
    "State": "Enabled", "Health": "OK", "HealthRollup": "Warning",
    "Conditions": [ {
      "MessageId": "Sensor.1.0.ReadingAboveUpperCautionThreshold",
      "OriginOfCondition": { "@odata.id": "/redfish/v1/Chassis/1U/Sensors/CPU1Temp" }
    } ]
  },
  "EnvironmentMetrics": {
    "@odata.id": "/redfish/v1/Systems/437XR1138R2/Processors/CPU1/EnvironmentMetrics"
  }
}
```

2.2.2 Resource model

The schema defines roughly three hundred resource types. The top of the hierarchy is fixed by the service root, which links to collections such as Systems (logical computer systems), Chassis (physical enclosures and boards), Managers (BMCs), UpdateService, SessionService, AccountService, EventService, Registries and JsonSchemas. Hardware components are reached from systems and chassis:

- ComputerSystem → Processors, Memory, Storage, SimpleStorage, EthernetInterfaces, NetworkInterfaces, ...
- Chassis → Power / PowerSubsystem, Thermal / ThermalSubsystem, Sensors, NetworkAdapters, Drives, PCIEDevices, ...

The specification distinguishes two kinds of links. Properties of a resource that reference *subordinate* resources (Processors, Power, Members) define a containment hierarchy. References inside the Links object express other relations and “shall contain links to resources that are related to but are not contained by, or subordinate to, this resource” [28]. Two further groups of references do not denote containment either: Actions contain POST targets, and the object Oem holds vendor-defined extensions, whose content is outside the standard.

2.2.3 Schemas and versioning

Every resource type is published by the DMTF as CSDL (OData XML), JSON Schema [52] and OpenAPI [44] documents in the schema bundle DSP8010. A JSON Schema file exists for every released version of every type (e.g. `Chassis.v1_9_0.json`), plus an unversioned file listing all versions. Redfish follows strict compatibility rules: minor versions only add properties, and properties are never removed but may be *deprecated*. The deprecation that affects monitoring most is the power and thermal model introduced in release 2020.4: `Chassis.Power` and `Chassis.Thermal`, whose power supplies, fans and temperatures are embedded arrays, were deprecated in `Chassis v1.15.0` in favour of `PowerSubsystem` and `ThermalSubsystem`, whose `PowerSupply` and `Fan` members are separate resources. Because deprecated properties remain valid, services in the field implement the old model, the new model or both.

Vendors extend the model in two ways: OEM properties inside `Oem` objects, with OEM schemas (e.g. `Hpeilo`), and *placement* choices that the standard leaves open, such as whether drives are linked from `Storage`, from `Chassis`, or both. The DMTF publishes *mockups*, static file-system snapshots of complete services, as reference material, together with tools to serve and record them [16, 17]. Interoperability profiles (DSP0272) allow a consumer to state which resources and properties it requires [26], and the Redfish Service Validator checks conformance of a live service against the schema [19].

2.3 Breadth-First Search

Given a directed graph $G = (V, E)$ and a source r , breadth-first search visits the vertices in non-decreasing order of their distance $d(v)$ from r . It keeps a first-in-first-out queue Q and a set of discovered vertices; each dequeued vertex u enqueues all undiscovered successors w and records u as their parent. The parent pointers form the *BFS tree*, a shortest-path tree of G from r . BFS runs in $O(|V| + |E|)$ time [13]. It was introduced by Moore for finding shortest paths through a maze [41] and is the standard traversal order of web crawlers, where it tends to discover important pages early [42].

Two properties of BFS are central to this thesis. First, the frontier at distance d , the set $L_d = \{v : d(v) = d\}$, is complete before any vertex of L_{d+1} is expanded. All requests of one level are therefore independent and can be issued concurrently (*level-synchronous* BFS). Second, BFS reaches each vertex along a shortest path, which for Redfish is the canonical, least deeply nested location of a resource. Depth-first search (DFS) uses a stack instead of a queue and has the same asymptotic cost, but it reaches deep vertices early and its frontier offers no natural unit of parallelism.

Chapter 3

Related Work

This thesis sits at the intersection of four lines of work: out-of-band monitoring systems built on Redfish (Section 3.1), client libraries and tools that talk to Redfish services (Section 3.2), hypermedia-driven clients and web crawling (Section 3.3), and the evolution of web APIs and its effect on clients (Section 3.4).

3.1 Redfish-Based Monitoring

Standards and overviews. The DMTF’s white paper introduced Redfish as a “RESTful interface” for “scalable platform management” that replaces IPMI [15]; the Redfish and OData white paper explains the annotation conventions (`@odata.id`, `@odata.type`) on which our graph model rests [20]. Gonçalves et al. survey the standard for the communications community [32]. The specification itself anticipates the problem studied here. It states that “different hardware or device types result in different resource trees, even for identical systems from the same manufacturer”, that “references between resources may result in a graph instead of a tree”, that clients traversing the tree “should provide logic to avoid infinite loops”, and that clients “should not assume that they can reuse hyperlinks across different Redfish service instances” (DSP0266 §5.3 [24]). It further requires that “clients shall not make assumptions about the URIs for the members of a resource collection” (§7.2.2) and describes discovery “through referenced URIs” from the single service root as the intended access method (§5.4.4). The specification thus mandates discovery, but it does not say how a client should discover *efficiently*. This thesis addresses that gap.

HPC monitoring. The group of Chen, Sill, Hass and colleagues at Texas Tech University has published most of the academic work on Redfish monitoring. Hojati et al. benchmarked Redfish against IPMI for automated hardware management [34] and built the Redfish Green500 Benchmark for automated power measurement [35]. MonSTer collects BMC and scheduler metrics for a 467-node cluster; it reads thermal data from the

Dell-specific URI `/redfish/v1/Chassis/System.Embedded.1/Thermal/`, reports an average of 4.29 s per Redfish request and about 55 s for an asynchronous sweep of all nodes, and notes that the iDRAC “has limited resources and cannot handle a large number of requests” [38, 43]. Redfish-Nagios integrates Redfish into the Nagios monitoring system using a fixed table of URIs under Systems, Chassis and Managers [2]; its extended version reports node round-trip times between 4.13 and 5.82 s [3]. Dakić et al. combine the Redfish API of HPE iLO servers with the vSphere API for policy-based monitoring, addressing resources such as `/redfish/v1/Systems/1` directly [14]. Outside Redfish, LDMS provides scalable in-band metric collection for HPC [1], and DiG adds a dedicated out-of-band board for high-resolution power monitoring, contrasting it with the roughly one-second resolution achievable over IPMI [40].

Two observations motivate this thesis. First, all of these systems address resources by *hard-coded URI templates* tuned to the vendor of their cluster. None of them evaluates what happens on a different vendor or firmware generation. Second, their latency measurements show that each request to a BMC is expensive, on the order of seconds under load. A discovery algorithm that crawls a service of 1,000 resources sequentially would take more than an hour at that rate, so request count is the right cost measure.

Security. BMCs are a sensitive attack surface. Bonkoski et al. analysed the IPMI implementation of a major BMC vendor and found vulnerabilities that expose the host [6]; PMFault shows that the BMC’s access to the power management bus can be used to fault or permanently damage server CPUs [10]; and Almeida et al. use Redfish as a case study for securing RESTful asset management APIs under IEC 62443-4-2 [4]. For our purposes these works imply that monitoring clients should run with read-only accounts and issue as few requests as possible, which again favours pruned discovery.

3.2 Redfish Client Libraries and Tools

The DMTF maintains a Python client library that handles sessions, paging and JSON parsing but leaves navigation to the caller [18]. Sushy, the Redfish library of OpenStack Ironic, wraps selected resources in Python classes with fixed attribute paths; its documentation warns that BMCs “are very flaky” and advises minimising requests [47]. The Prometheus `redfish_exporter` maps a fixed set of resources to metrics [37]. OpenBMC’s `bmcweb` is a server-side implementation [45, 46].

Closest to our approach are two DMTF tools that *traverse* a service. The Redfish Mockup Creator walks the complete resource graph and stores every resource as a file [16]; the resulting mockups can be replayed by the Mockup Server [17] and are the data source of our evaluation. The Redfish Service Validator crawls a service and checks every re-

source against its @odata.type schema, including JSON Pointer fragments of embedded objects [8, 19]. Both tools are exhaustive by design. This thesis instead targets *selective* traversal for a known set of component types, and it uses the schema not to validate what was fetched but to decide what need not be fetched at all.

The Redfish Interoperability Profiles (DSP0272) let a client publish which resources and properties it relies on and let a validator check a service against that profile [26]. Profiles detect *whether* a service provides what a client needs; they do not tell the client *where* to find it. Since protocol version 1.6.0 the schemas also carry *URI patterns*, and conforming services implement them (DSP0266 §5.4.5, §9.13.5 [24]). URI patterns narrow the search space but do not remove it. The 2026.2 bundle lists 21 permitted URI patterns for Processor, 10 for EthernetInterface, 8 for Drive, 7 for Memory and 6 for Sensor. Every pattern contains identifier variables that can only be bound by reading collections, and services older than 1.6.0, such as the Dell iDRAC mockups studied here, predate the patterns entirely. Section 7.7.1 discusses how URI patterns and SP-BFS complement each other.

3.3 Hypermedia Clients and Web Crawling

Fielding’s REST architectural style requires that clients drive application state through hypermedia links provided by the server (the HATEOAS constraint) [31]. URIs are to be discovered, not constructed; the generic URI syntax defines only how a path is written, not what its segments mean [5]. Most practical REST clients ignore this constraint and hard-code endpoint templates from documentation, which is exactly the failure mode we observe in Redfish tools. Redfish is one of the few widely deployed APIs whose specification insists on hypermedia discovery, which makes it a natural setting for a hypermedia-driven client.

Web crawlers solve a related problem at a larger scale. Najork and Wiener showed that breadth-first crawling discovers high-quality pages early [42]; Cho et al. studied how to order the crawl frontier so that important pages are fetched first [12], and how an incremental crawler should revisit pages as the web changes [11]. Our setting differs in three respects: the graph is small but each request is very expensive; the client knows exactly which node *types* it wants; and a machine-readable schema describes which types can occur below which. The schema allows *exact* pruning with a soundness guarantee rather than the heuristic frontier ordering of general crawlers. The discover-then-poll structure of our monitoring tool mirrors the incremental crawler’s separation of discovery and revisiting.

3.4 Web API Evolution

Empirical studies of web API evolution show that API changes regularly break clients. Li et al. classified the changes of five popular web APIs and their impact on client code [39];

Espinha et al. interviewed client developers and analysed client code, finding that even “loosely coupled” web APIs create strong coupling through hard-coded endpoints [29, 30]; Sohan et al. studied which changes providers make between API versions and how they document and communicate them [51]. Redfish is unusual in that its compatibility rules forbid removing properties, but deprecation plus optional features plus vendor freedom in placement still produce the silent breakage that these studies describe. The approach in this thesis can be read as a client-side strategy to decouple from such changes: the client depends only on type names, which are stable across versions, and on the published schema, which is versioned and machine readable.

3.5 Summary

Existing Redfish monitoring systems hard-code vendor-specific URIs and report per-request latencies of seconds. Existing Redfish tools either leave navigation to the user or traverse exhaustively. Web crawling offers traversal strategies but no schema-based pruning. To the best of our knowledge, no prior work (i) systematically measures placement heterogeneity across Redfish vendors and versions, or (ii) uses the Redfish schema to prune the discovery of monitored components with a completeness guarantee. These are the gaps the following chapters address.

Chapter 4

Problem Analysis and Formal Model

This chapter turns the informal question “which URI has to be called to get the power supplies?” into a precise search problem. Section 4.1 models a Redfish service as a typed graph, Section 4.2 shows why its containment edges form an almost-tree, Section 4.3 defines resource queries, and Section 4.4 classifies the kinds of heterogeneity a discovery algorithm has to tolerate.

4.1 The Resource Graph

Definition 4.1 (Resource graph). *A Redfish service is a labelled directed multigraph*

$$G = (V, E, r, \tau, \beta), \quad E \subseteq V \times \Pi \times \mathcal{P} \times V, \quad (4.1)$$

where V is the set of resource URIs that answer a GET, $r = /redfish/v1$ is the service root, $\tau : V \rightarrow \mathcal{T} \cup \{\perp\}$ maps a resource to the short name of its `@odata.type` (e.g. `PowerSupply`) or to $\perp$ if the type is missing, and $\beta(v)$ is the JSON body of v . An edge $(u, p, \pi, w) \in E$ exists if the body $\beta(u)$ contains, at JSON Pointer [8] π below the top-level property $p \in \Pi$, an object $\{"@odata.id" : w\}$ with $w \neq u$.

Every edge is classified by the property names on its pointer π . We partition

$$E = E_c \dot{\cup} E_x \dot{\cup} E_o \dot{\cup} E_a, \quad (4.2)$$

into *containment* edges E_c ; *cross references* E_x , where some key on π is `Links`, `RelatedItem`, `RedundancyGroup` or `OriginOfCondition`; *OEM* edges E_o , where some key is `Oem`; and *action/settings* edges E_a , where some key is `Actions` or `@Redfish.Settings`. Edges in E_a point to POST targets and pending-configuration copies and are never traversed.

Remark 4.1. URIs in Redfish are *opaque*: a client must not infer the type or the parent of a resource from its path. The model therefore uses URIs only as identifiers, and all structural information comes from the edges.

4.2 The Containment Tree

The Redfish specification describes resources as organised in a hierarchy below the service root, in which each subordinate resource is referenced by a navigation property of its parent. In graph terms, the subgraph $G_c = (V_c, E_c)$ of vertices reachable from r over containment edges is expected to be a rooted tree. Real services deviate from this ideal in three ways: the same drive is linked from both a Storage and a Chassis resource; a ResourceBlock of the composition service re-exposes processors and memory of a system; and a few vendor resources appear in more than one collection. We therefore call G_c an *almost-tree* and measure how far it is from a tree by the share of vertices with more than one containment parent,

$$\mu(G) = \frac{|\{w \in V_c : |\{u : (u, \cdot, \cdot, w) \in E_c\}| > 1\}|}{|V_c|}. \quad (4.3)$$

Across the 26 services of the evaluation, $\mu = 3.1\%$ overall and at most 12.1% for a single service (Chapter 7). A BFS over G_c turns the almost-tree into a proper tree T by keeping, for every vertex, the edge through which it was first discovered; by the BFS property this is a shortest path from r . The *depth* $d(v)$ of a vertex is its distance from r in G_c , and the *levels* of the tree are

$$L_k = \{v \in V_c : d(v) = k\}, \quad k = 0, \dots, D, \quad (4.4)$$

where $D = \max_v d(v)$ is the height of the tree. All services studied have $6 \leq D \leq 8$ (see Fig. 7.7).

4.3 Resource Queries

A monitoring tool is interested in a small number of hardware classes. Instead of a location, a class is described by *what* its instances are.

Definition 4.2 (Resource query). *A resource query is a triple $Q = (P, M, \varphi)$ where $P \subseteq \mathcal{T}$ is a set of resource types, $M \subseteq \mathcal{T} \times \Pi$ is a set of embedded locations (a parent type and one of its array properties), and $\varphi : \mathcal{T} \times \text{JSON} \rightarrow \{0, 1\}$ is a content predicate. The answer of Q on G is*

$$\begin{aligned} \mathcal{A}_Q(G) = & \{ v \in V_c : \tau(v) \in P \wedge \varphi(\tau(v), \beta(v)) \} \\ & \cup \{ \iota(v, m, i) : v \in V_c, (\tau(v), m) \in M, 0 \leq i < |\beta(v)[m]| \}, \end{aligned} \quad (4.5)$$

Table 4.1: Resource queries of the monitoring tool. φ is true unless stated.

Query	Types P	Embedded locations M	Predicate φ
Processor	Processor	–	ProcessorType is not Core or Thread
Memory	Memory	–	–
Storage	Drive	(SimpleStorage, Devices)	–
Network	EthernetInterface, NetworkAdapter	–	–
Power	PowerSupply	(Power, PowerSupplies)	–
Thermal	Fan, Sensor	(Thermal, Fans), (Thermal, Temperatures)	for Sensor: ReadingType is Temperature

where $\iota(v, m, i)$ is the identity of the i -th element of array m in $\beta(v)$: the element's own @odata.id if it has one without a fragment, and the JSON-Pointer URI $v\#/m/i$ otherwise.

The embedded part M is needed because older schemas describe components as array elements rather than resources: in Power v1 the power supplies are elements of PowerSupplies, in Thermal v1 the fans and temperature readings are elements of Fans and Temperatures, and in SimpleStorage the drives are elements of Devices. The identity function ι matters for services such as early iDRAC firmware, which give each embedded element a dereferenceable URI that is also reachable as a separate resource of type PowerSupply; without ι the same power supply would be counted twice.

Definition 4.3 (Anchor types). *The anchor types of Q are the types whose bodies must be fetched to evaluate Q :*

$$\text{Anc}(Q) = P \cup \{t : (t, m) \in M\}. \quad (4.6)$$

Table 4.1 lists the six queries used by the monitoring tool and in the evaluation. They are the only service-independent knowledge the tool requires.

Scope. The tool monitors the host, not the BMC itself. Resources below Managers (the BMC's own Ethernet interfaces, logs and accounts) are excluded by a set $X = \{\text{ManagerCollection}, \text{Manager}\}$ of *excluded types* whose subtrees are not entered.

Discovery problem. Given query Q and access to G only through GET requests starting at r , compute $\mathcal{A}_Q(G)$ while minimising the number of requests

$$C = |\{v \in V : v \text{ is fetched}\}|. \quad (4.7)$$

Because G is unknown in advance, every algorithm has to fetch at least the resources that contain the answer and one path of ancestors to each of them. An exhaustive traversal of G_c fetches all of V_c , which is complete but expensive. A hard-coded client fetches a fixed set of paths, which is cheap but complete only for services laid out as expected.

4.4 Heterogeneity of Redfish Services

The discovery problem is only interesting because services differ. From the schema history and the services in Chapter 7 we distinguish five kinds of heterogeneity. Figure 4.1 illustrates three of them with concrete paths from three recorded services.

H1 Identifier heterogeneity. Member identifiers are chosen freely (1, System.Embedded.1, 437XR1138R2, Self, Node0, QSBP74100021). Literal URIs never transfer between vendors.

H2 Model generation. The same component is described by different types in different schema generations, e.g. an embedded element of `Power.PowerSupplies` versus a `PowerSupply` resource below `PowerSubsystem`; a `SimpleStorage` device versus a `Drive`. Services may implement both generations at once.

H3 Placement. The standard permits several parents for the same type. Drives are linked from `Storage`, from `Chassis` or from a composition `ResourceBlock`; network adapters are found below `Chassis`, Ethernet interfaces below `ComputerSystem`.

H4 Optional features. `$expand`, `$select` and `$filter` are optional, so a client cannot rely on server-side query evaluation to find resources.

H5 Vendor extensions and defects. OEM subtrees, resources without `@odata.type`, and links placed outside the standard's navigation properties.

H1 and H4 exclude the naive solutions (literal URIs, server-side filters). H2 and H3 are what defeats hard-coded navigation, and they are the focus of this thesis. H5 is the reason why a discovery algorithm must degrade gracefully when the schema cannot tell it what a link means.

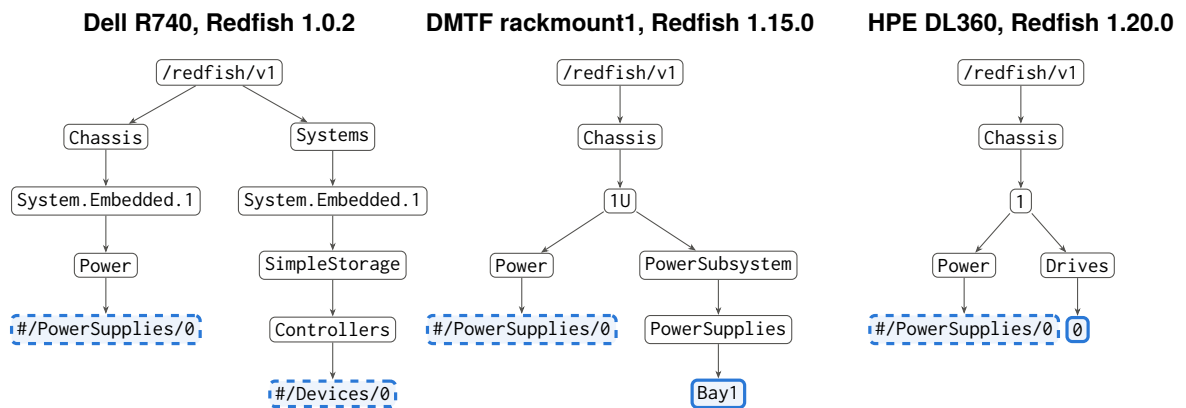


Figure 4.1: Placement of power supplies and drives in three recorded services (paths abridged; only relevant branches shown). Solid blue boxes are target resources, dashed boxes are embedded array elements. The same query must match all of them.

Chapter 5

Schema-Guided Breadth-First Discovery

This chapter presents the discovery method. Section 5.1 gives an overview, Section 5.2 constructs the type reachability graph from the Redfish schema, Section 5.3 states the schema-pruned BFS algorithm, and Sections 5.4 to 5.6 analyse its correctness, cost and parallelism. Section 5.7 embeds the algorithm into a monitoring loop.

5.1 Overview

The starting point is the plain idea behind the author’s monitoring tool: convert the service into a tree by following links from `/redfish/v1`, and find the requested resources by BFS, testing the `@odata.type` of every node. We call this *Tree-BFS*. It needs no vendor knowledge, and by construction it finds every target reachable over containment edges. Its weakness is cost: it fetches the entire containment tree, including thousands of log entries, schema files and registry entries that can never contain a power supply.

The key observation is that the client does not have to fetch a resource to know that nothing interesting can lie below it. The Redfish schema states, for every resource type and every navigation property, which types the property can point to. The client can therefore decide from the parent’s type and the property name alone whether a link can lead to a target. The resulting algorithm, *schema-pruned BFS* (SP-BFS), has two phases:

1. **Offline:** build a type graph H from the DMTF JSON Schema bundle and precompute reachability (Algorithm 1). This is done once per schema bundle and is independent of any service.
2. **Online:** run BFS from the service root, evaluating the queries on each fetched body and enqueueing only those links whose possible target types can reach an anchor type of some query (Algorithm 2).

Figure 5.1 illustrates the effect on the upper levels of a service: all service-level branches

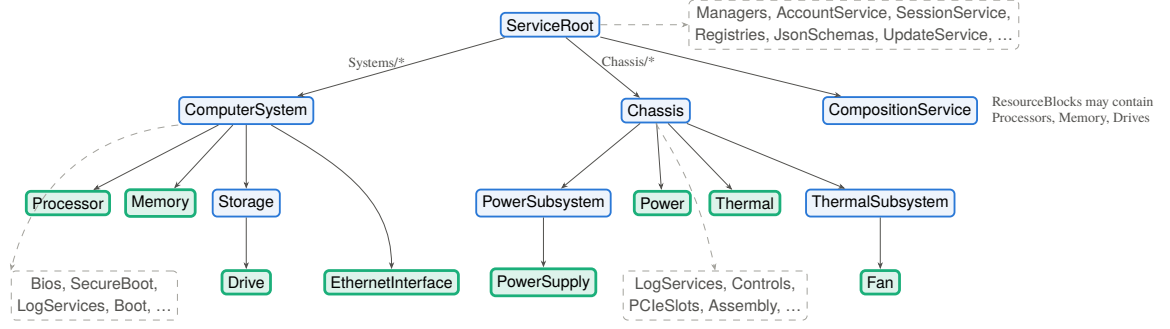


Figure 5.1: Schema pruning on the upper levels of a service for the six queries of Table 4.1 (intermediate collection resources omitted). Blue nodes are expanded, green nodes hold anchor types, and dashed grey branches are pruned before any request is sent because no anchor type is reachable from them in the type graph H .

except Systems, Chassis and CompositionService are pruned at the service root, and below a system or chassis only the branches that can lead to a monitored type survive.

5.2 The Type Reachability Graph

Definition 5.1 (Type graph). *Let $\mathcal{T}$ be the set of resource types defined in the schema bundle and Π the set of property names. The type graph is*

$$H = (\mathcal{T}, F), \quad F \subseteq \mathcal{T} \times \Pi \times \mathcal{T}, \quad (5.1)$$

with $(t, p, t') \in F$ if and only if some published version of type t declares a top-level property p , other than Links, Actions and Oem, whose JSON Schema subtree contains a reference \$ref to the definition of resource type t' . We write $H(t, p) = \{t' : (t, p, t') \in F\}$ for the possible targets of property p of type t , and $\text{Decl}(t)$ for the set of top-level properties declared by any version of t .

Taking the union over all versions makes H valid for every schema version a service may implement, from a Redfish 1.0 service that only knows Chassis.Power to a 2026 service that also knows Chassis.PowerSubsystem. Collections are ordinary types: the edge $(\text{PowerSupplyCollection}, \text{Members}, \text{PowerSupply}) \in F$ is obtained from the Members array of the collection schema.

For a set $X \subseteq \mathcal{T}$ of excluded types, the *reachability* of a type is the reflexive-transitive closure of H avoiding X :

$$\mathbf{R}_X(t) = \{t\} \cup \{t' \in \mathcal{T} \setminus X : \exists t = t_0, t_1, \dots, t_k = t' \text{ with } t_{i+1} \in H(t_i, p_i), t_1, \dots, t_k \notin X\}. \quad (5.2)$$

Applied to the DMTF schema bundle 2026.2 (6,826 versioned JSON Schema files), Algorithm 1 yields $|\mathcal{T}| = 278$ resource types and $|F| = 499$ labelled edges in about one

Algorithm 1 Construction of the type graph H from a JSON Schema bundle

Require: directory B of JSON Schema files $T.json$, $T.vX_Y.Z.json$

Ensure: edge map $H : \mathcal{T} \times \Pi \rightarrow 2^{\mathcal{T}}$, declared properties Decl

```
1:  $\mathcal{T} \leftarrow \{T : T.json \in B\} \setminus \{\text{Resource}, \text{odata}, \dots\}$ 
2: for all  $T \in \mathcal{T}$  do
3:   for all versioned or unversioned file  $f$  of  $T$  do
4:      $D \leftarrow$  definition of  $T$  in  $f$ 
5:     for all top-level property  $p$  of  $D$  with  $p \notin \{\text{Links}, \text{Actions}, \text{Oem}\}$  do
6:        $\text{Decl}(T) \leftarrow \text{Decl}(T) \cup \{p\}$ 
7:        $\text{WALK}(T, p, \text{schema of } p)$ 

8: procedure  $\text{WALK}(T, p, s)$   $\triangleright$  collect resource references below property  $p$ 
9:   if  $s$  has $ref to #/definitions/L (local, not yet seen) then
10:     $\text{WALK}(T, p, \text{definition } L)$ 
11:   else if  $s$  has $ref to U.json#/definitions/U with  $U \in \mathcal{T}$  then
12:     $H(T, p) \leftarrow H(T, p) \cup \{U\}$ 
13:   for all child schema  $s'$  of  $s$  in properties, items, anyOf, oneOf, allOf do
14:     $\text{WALK}(T, p, s')$ 
```

second. Reachability sets are computed lazily by BFS on H and memoised, so each $R_X(t)$ costs $O(|\mathcal{T}| + |F|)$ once.

Expansion rule. During discovery, a reference found in the body of a node u of type $t = \tau(u)$ under top-level property p is *expanded* (its target is enqueued for fetching) according to the following rule, where $A = \bigcup_j \text{Anc}(Q_j)$ is the union of the anchor types of all active queries:

$$\text{expand}(t, p) = \begin{cases} \text{true} & \text{if } t = \perp \text{ or } p \notin \text{Decl}(t) \text{ (unknown),} \\ \exists t' \in H(t, p) \setminus X : R_X(t') \cap A \neq \emptyset & \text{otherwise.} \end{cases} \quad (5.3)$$

Three consequences of Eq. (5.3) are worth noting. (i) If p is declared but is not a navigation property ($H(t, p) = \emptyset$), references inside it, such as `Status/Conditions/*` or `OriginOfCondition`, are cross references and are never followed. (ii) Vendor-specific properties that are not in the schema, and resources without `@odata.type`, are expanded, so the rule degrades to plain Tree-BFS rather than losing data. (iii) `Oem`, `Links` and `Actions` subtrees are skipped entirely, as in Tree-BFS.

Algorithm 2 Schema-pruned breadth-first discovery (SP-BFS)

Require: service root r , queries $Q_1, \dots, Q_m$, type graph H , excluded types X

Ensure: answer sets $\mathcal{R}_1, \dots, \mathcal{R}_m$

```
1:  $A \leftarrow \bigcup_j \text{Anc}(Q_j)$ ;  $\mathcal{R}_j \leftarrow \emptyset$  for all  $j$ 
2:  $\text{frontier} \leftarrow \langle r \rangle$ ;  $\text{seen} \leftarrow \{r\}$ 
3: while  $\text{frontier} \neq \langle \rangle$  do
4:    $u \leftarrow \text{POPFRONT}(\text{frontier})$ 
5:    $b \leftarrow \text{GET}(u)$ ; if  $b = \text{error}$  then continue
6:    $t \leftarrow \tau(b)$ ; if  $t \in X$  then continue
7:   for  $j = 1, \dots, m$  do ▷ evaluate queries, Eq. (4.5)
8:     if  $t \in P_j \wedge \varphi_j(t, b)$  then  $\mathcal{R}_j \leftarrow \mathcal{R}_j \cup \{u\}$ 
9:     for all  $(t, a) \in M_j$  and  $0 \leq i < |b[a]|$  do
10:       $\mathcal{R}_j \leftarrow \mathcal{R}_j \cup \{\iota(u, a, i)\}$ 
11:   for all reference  $(p, \pi, w)$  in  $b$  with  $\pi$  outside Links, Oem, Actions do
12:     if  $w \notin \text{seen} \wedge \text{expand}(t, p)$  then ▷ Eq. (5.3)
13:        $\text{seen} \leftarrow \text{seen} \cup \{w\}$ ;  $\text{PUSHBACK}(\text{frontier}, w)$ 
14: return  $\mathcal{R}_1, \dots, \mathcal{R}_m$ 
```

5.3 The SP-BFS Algorithm

Algorithm 2 gives the complete discovery procedure. It evaluates any number of queries in a single traversal, which is how the monitoring tool uses it; a single-query call is the special case $m = 1$.

Details. References are extracted by a recursive walk over the body that records, for each object with an @odata.id other than the resource's own, the top-level property and the JSON Pointer of the object. References whose target differs from u only by a fragment point into u itself and are handled by the embedded part of the query. Members@odata.nextLink pages are treated as members of the same collection. The *seen* set makes the algorithm terminate on graphs with cycles, as DSP0266 requires of traversing clients. A reference rejected by Eq. (5.3) is *not* added to *seen*, so the same resource can still be discovered through another, admissible parent.

5.4 Soundness and Necessity

Pruning is only acceptable if it never loses a target that the exhaustive traversal would find. We make precise which services this can be guaranteed for.

Definition 5.2 (Schema conformance). *A service G is schema-conformant with respect to H if every vertex $v \in V_c$ has a type $\tau(v) \in \mathcal{T}$, and every containment edge $(u, p, \pi, w) \in E_c$ satisfies $p \in \text{Decl}(\tau(u))$ and $\tau(w) \in H(\tau(u), p)$.*

Conformance is exactly what the Redfish Service Validator checks for each link: the target of a navigation property has the type declared by the schema.

Theorem 5.1 (Soundness). *Let G be schema-conformant and let V_c^X be the set of vertices reachable from r in G_c by paths whose inner vertices have types outside X . Then for every query Q_j , SP-BFS returns*

$$\mathcal{R}_j = \mathcal{A}_{Q_j}(G) \cap (V_c^X \cup \{\iota(v, a, i) : v \in V_c^X\}),$$

i.e. exactly the answers that Tree-BFS with the same exclusions returns.

Proof. ($\subseteq$) Every element added to $\mathcal{R}_j$ satisfies the membership condition of Eq. (4.5) for the body in which it was found, and every fetched vertex is reached over containment edges from r without expanding an excluded vertex.

($\supseteq$) Let $v \in V_c^X$ be a vertex whose body contributes an answer to Q_j , i.e. $\tau(v) \in \text{Anc}(Q_j) \subseteq A$. Choose a containment path $r = v_0 \xrightarrow{p_0} v_1 \xrightarrow{p_1} \dots \xrightarrow{p_{k-1}} v_k = v$ with $\tau(v_i) \notin X$ for $0 < i < k$. By conformance, $\tau(v_{i+1}) \in H(\tau(v_i), p_i)$ for every i , so the type sequence $\tau(v_{i+1}), \dots, \tau(v_k)$ is a path in H that avoids X , and therefore $\tau(v) \in R_X(\tau(v_{i+1}))$. Hence $R_X(\tau(v_{i+1})) \cap A \ni \tau(v)$ and $\text{expand}(\tau(v_i), p_i)$ is true for every edge of the path. Consequently every edge of the path lies in the subgraph $G' \subseteq G_c$ of edges that Eq. (5.3) admits. SP-BFS is a breadth-first search of G' (rejected edges do not mark their target as seen), so it fetches every vertex reachable from r in G' , in particular v , and evaluates Q_j on $\beta(v)$. $\square$

Corollary 5.2. *On schema-conformant services, SP-BFS and Tree-BFS have identical recall and precision for every query, and $C_{\text{SP-BFS}} \leq C_{\text{Tree-BFS}}$.*

For non-conformant services the expansion rule is conservative wherever the schema is silent (unknown types or properties). A target can then be lost only if a service links it through a *declared* property whose schema says it cannot lead there, i.e. through a genuine schema violation. Chapter 7 shows that this did not occur on any of the 26 services studied.

Soundness says that SP-BFS fetches no less than necessary. The converse question is whether it fetches more than necessary. Clearly it may: below a Processor it still visits SubProcessors, because in principle a processor can contain processors, even when the predicate excludes cores and threads. The following proposition shows that such requests cannot be avoided by any sound algorithm that has the same information.

Proposition 5.3 (Necessity). *Let $\mathcal{B}$ be any discovery algorithm that knows the schema H but, like SP-BFS, learns about the service only by fetching resources, and that is sound on all schema-conformant services. Suppose $\mathcal{B}$ has fetched u and $\beta(u)$ contains a containment reference (p, π, w) with $\text{expand}(\tau(u), p)$ true, and w is referenced by no other resource. Then $\mathcal{B}$ must fetch w .*

Proof sketch. Since $\text{expand}(\tau(u), p)$ holds, there is a type path $t' = t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_k \in A$ in H with $t_1 \in H(\tau(u), p)$. Consider two services: G , and G^* obtained from G by giving w the type t_1 and appending below w a fresh chain of resources of types $t_2, \dots, t_k$. Both services are schema-conformant, and they differ only at w and below it. If $\mathcal{B}$ does not fetch w , it observes identical responses on G and G^* and returns the same answer on both. But G^* contains an answer of type t_k that G does not contain, so $\mathcal{B}$ is unsound on one of them. $\square$

In this sense SP-BFS is *instance-optimal among schema-only algorithms*: every request it issues could have been the one leading to a target. The remaining gap to a hand-written client comes from knowledge that is not in the schema, e.g. that a vendor never puts drives under a composition resource block. A hand-written client exploits such knowledge, and it is also what breaks when that knowledge becomes wrong.

5.5 Cost

Let $V' \subseteq V_c$ be the vertices reachable from r in the admissible subgraph G' . The request cost of SP-BFS is exactly

$$C_{\text{SP-BFS}} = |V'| \leq |V_c| = C_{\text{Tree-BFS}}, \quad (5.4)$$

and its computation time is $O(|V'| + |E'|)$ for the traversal plus the one-off closure computations. With per-request latency ℓ and sequential requests, the discovery time is $C \cdot \ell$. We report the *pruning ratio*

$$\rho = \frac{C_{\text{SP-BFS}}}{C_{\text{Tree-BFS}}} \in (0, 1], \quad (5.5)$$

and aggregate it over services by the geometric mean, which is the appropriate mean for ratios.

A lower bound for *any* discovery algorithm is the size of the union of the shortest paths from r to the answer-bearing vertices, because each vertex on such a path must be fetched to learn the next link. For the query “all power supplies” on a modern service this is typically $\{r, \text{Chassis}, \text{Chassis}/1, \text{PowerSubsystem}, \text{PowerSupplies}\}$ plus one request per supply, i.e. $4 + n_{\text{PSU}}$ requests.

5.6 Level-Synchronous Parallelism

BFS processes the tree level by level. All vertices of level L_k are known before any vertex of L_{k+1} is fetched, so the requests of one level are independent and can be issued concurrently. Let $n_k = |L_k \cap V'|$ and let the client keep at most c requests in flight (BMCs typically limit

the number of sessions and degrade under load, so c is small). With the level barrier, the number of *request rounds* is

$$R(c) = \sum_{k=0}^D \left\lceil \frac{n_k}{c} \right\rceil, \quad \max\left(K, \frac{C}{c}\right) \leq R(c) \leq K + \frac{C}{c}, \quad (5.6)$$

where $C = \sum_k n_k$ and K is the number of non-empty levels. The expected wall-clock time is approximately $R(c) \bar{\ell}(c)$, where $\bar{\ell}(c)$ is the mean latency at concurrency c . For $c \rightarrow \infty$ the time is bounded below by the height of the tree, which for Redfish is small ($D \leq 8$ in all services studied). DFS has the same total cost but no level structure, so its natural parallelisation requires speculative expansion of several branches.

BFS has one latency drawback that the evaluation makes visible: plain Tree-BFS must sweep the broad upper levels (schemas, registries, logs) before it reaches depth 4–6 where hardware components live, so its *first* answer arrives late. Pruning removes exactly those broad, irrelevant levels.

5.7 Discovery within a Monitoring Loop

Hardware inventory changes rarely, while sensor readings change continuously. The monitoring tool therefore separates discovery from polling (Algorithm 3). Discovery produces a *poll set*: the distinct URIs of the answer-bearing resources (the resource itself for resource answers, the parent for embedded answers such as Power#/PowerSupplies/∅). Each polling cycle fetches only the poll set. Over N cycles the mean cost per cycle is

$$\bar{C}(N) = \frac{C_{\text{disc}}}{N} + C_{\text{poll}} \xrightarrow{N \rightarrow \infty} C_{\text{poll}}, \quad (5.7)$$

so any discovery algorithm is amortised after a few cycles. What still matters is (i) the *completeness* of the poll set, which is fixed at discovery time, and (ii) the cost of *rediscovery*, which is needed when the inventory changes.

The *signature* is a cheap fingerprint of the inventory: the `Members@odata.count` of the collections on the paths to the answers, which are among the resources already fetched and can be re-read periodically at a fraction of the discovery cost. Redfish event subscriptions (ResourceAdded, ResourceRemoved) can trigger rediscovery immediately on services that support them.

Algorithm 3 Monitoring loop with event-driven rediscovery

Require: service root r , queries $\mathcal{Q}$, type graph H , period Δ

```
1:  $\mathcal{R} \leftarrow \text{SP-BFS}(r, \mathcal{Q}, H)$ ;  $S \leftarrow \{\text{base}(x) : x \in \mathcal{R}\}$   $\triangleright$  poll set
2:  $\sigma \leftarrow \text{SIGNATURE}(r)$   $\triangleright$  sizes of collections on answer paths
3: loop
4:   for all  $s \in S$  in parallel (at most  $c$  ) do
5:      $b \leftarrow \text{GET}(s)$ 
6:     if  $b = 404$  or  $\tau(b)$  changed then
7:       mark rediscovery
8:      $\text{EMIT}(\text{readings and health of the answers contained in } b)$ 
9:     if rediscovery marked or  $\text{SIGNATURE}(r) \neq \sigma$  or an inventory event arrived then
10:       $\mathcal{R} \leftarrow \text{SP-BFS}(r, \mathcal{Q}, H)$ ; update  $S, \sigma$ 
11:    $\text{SLEEP}(\Delta)$ 
```

Chapter 6

Implementation

This chapter describes how SP-BFS is realised as the discovery layer of the Redfish hardware monitoring tool, and how the same code is exercised against recorded services for the evaluation.

6.1 Architecture

Figure 6.1 shows the components of the monitoring tool. The *collector* owns all communication with BMCs: it authenticates, limits the number of concurrent requests per BMC, applies timeouts and retries, and counts requests. The *discovery* component runs SP-BFS with the queries of Table 4.1 and the type graph built from the schema bundle; its only output is an *inventory*, i.e. the answer sets and the derived poll set, per server. The *poller* periodically fetches the poll set, normalises the readings (health state, temperature in °C, fan speed, power in W) into a vendor-neutral record keyed by the answer identity ι , and hands them to storage, dashboards and alerting. Nothing outside the discovery component knows a Redfish URI.

6.2 The rfbfs Package

The discovery layer is implemented in Python 3 without third-party dependencies, in six modules totalling about 670 lines:

service.py

The transport interface: a single method `get(uri)` that returns the JSON body or `None` and counts requests. The evaluation uses `MockupService`, which serves a DMTF mockup directory.

https.py

`HttpsService`, the transport for live BMCs with the same interface: session login

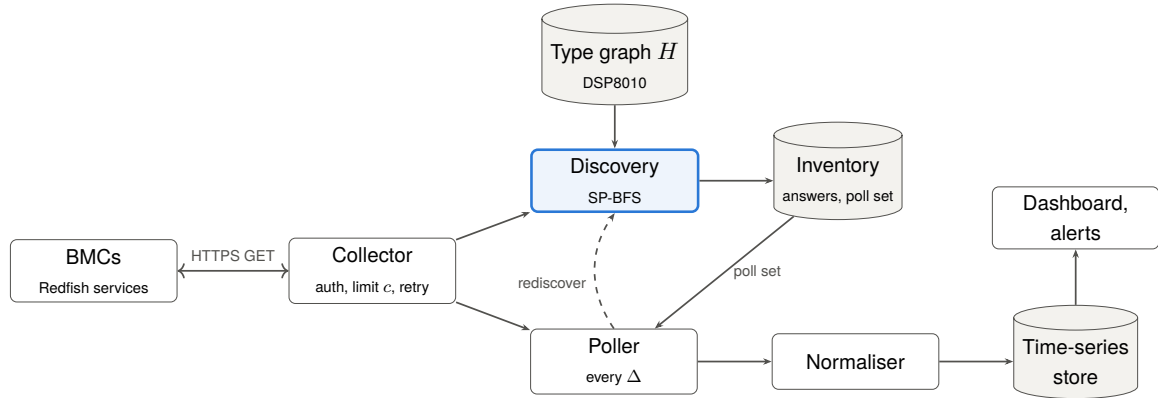


Figure 6.1: Architecture of the monitoring tool. Only the discovery component (blue) interprets the Redfish resource graph.

with fallback to HTTP Basic authentication, optional acceptance of self-signed certificates, timeouts, and retries with exponential back-off for 5xx and 429 responses.

model.py

Reference extraction (Definition 4.1): a recursive walk that yields every `@odata.id` reference with its top-level property, JSON Pointer and edge class (containment, cross reference, OEM, action/settings), and the short type name τ derived from `@odata.type`. The type name is the last dot-separated segment of the qualified name, which works for all historic spellings, e.g. `#ComputerSystem.v1_1_0.ComputerSystem` and `#ProcessorCollection.ProcessorCollection`.

schema.py

Algorithm 1 and the memoised reachability closure R_X (Eq. (5.2)).

queries.py

The declarative queries of Table 4.1 as immutable data: type sets, embedded locations and predicates.

discover.py

One traversal routine that implements, via parameters, the ground-truth crawl, Tree-BFS, Tree-DFS and SP-BFS (Algorithm 2); a level-synchronous variant `crawl_levels` that fetches each tree level with a bounded thread pool; and the static navigation baselines used in Chapter 7.

Listing 6.1 shows how the monitoring tool invokes the discovery. Adding a new component class (e.g. GPUs or PCIe devices) means adding one query; no navigation code has to be written.

Listing 6.1: Using the discovery layer.

```
from rfbfs.schema import TypeGraph
from rfbfs.queries import QUERIES
from rfbfs.discover import crawl_levels
from rfbfs.https import HttpsService

H = TypeGraph.from_schema_dir("DSP8010_2026.2/json-schema") # once per bundle
svc = HttpsService("https://bmc-17.example", "monitor", secret,
                  verify_tls=False) # same get() interface
inv = crawl_levels(svc, list(QUERIES.values()), schema=H, workers=4) # SP-BFS
poll_set = {x.split("#")[0] for ids in inv.targets.values() for x in ids}
```

6.3 Engineering Aspects

Sessions and authentication. The collector creates one Redfish session per BMC (POST to the Sessions collection of the SessionService) and sends the returned X-Auth-Token with every request, falling back to HTTP Basic authentication where sessions are not available. The monitoring account needs only the read-only ReadOnly role.

Concurrency. Level-synchronous BFS (Section 5.6) is implemented by fetching the current frontier with a bounded worker pool before building the next one. The fetched level is then processed in document order, so the result, including the request count, is identical to the sequential traversal; we verified this on all 26 services of the evaluation. The bound c is configurable per vendor because BMCs differ in how many parallel requests they tolerate.

Robustness. A failed request (timeout, 5xx, invalid JSON) is retried with exponential back-off and, if it still fails, recorded in the result's list of failed URIs, so that the inventory is marked as incomplete instead of silently missing a branch. The mockups contain such defects too, e.g. URIs whose percent-encoding differs from the stored path and directories whose letter case differs from the advertised URI; the mockup transport tolerates both, as a careful HTTP client would.

Identity and deduplication. Answers are keyed by the identity function ι of Definition 4.2, so the same power supply reported both as an embedded element and as a resource (early iDRAC firmware) is stored once. A remaining source of duplicates is services that implement both the legacy and the new power/thermal model. Their embedded and resource representations have different identities and are currently reported separately; merging them by their Location and serial-number properties is left as future work (Section 8.3).

Schema updates. The type graph is built from whatever schema bundle is configured. Because it is the union over all versions and schemas only ever add properties, a newer

bundle can only add edges, so upgrading the bundle never makes pruning less sound.

6.4 Evaluation Harness

The evaluation (Chapter 7) runs every strategy against every mockup through the same MockupService, so that request counts are directly comparable. The script `run_experiments.py` writes per-mockup, per-category, per-query and per-level results as CSV/JSON files, and `make_figures.py` computes all aggregate numbers and figures reported in this thesis from those files. Appendix B lists the exact data sources and commands.

Chapter 7

Evaluation

This chapter answers the research questions of Chapter 1 on 26 recorded Redfish services. Section 7.1 describes data, baselines and metrics; Sections 7.2 to 7.5 address RQ1–RQ4; Section 7.6 reports on the shape of the resource trees; and Section 7.7 discusses the results and threats to validity.

7.1 Experimental Setup

7.1.1 Services

Live access to many server models of different vendors and generations is rarely available to a single project. We therefore evaluate on *mockups*: complete recordings of Redfish services created with the DMTF Mockup Creator or equivalent tools, in which every resource is stored as the JSON document the BMC returned [16]. A mockup reproduces the resource graph exactly, so request counts and discovery results are identical to those against the live service. What a mockup cannot reproduce is latency, which we therefore model separately (Section 7.5).

We collected all public server mockups we could find, from four sources:

- the DMTF reference mockups of the Redfish-Publications repository at release 2026.2 (public-rackmount1, -tower, -bladed, -localstorage, -smartnic, -liquid-cooled-server);
- five Dell PowerEdge servers with iDRAC 7/8/9 firmware (R720, R740, T630, C6320, FC630) from Dell’s open-source Sailfish project;
- seven HPE ProLiant servers with iLO 5, iLO 6 and iLO 7 firmware from HPE’s iLO Redfish emulator;
- eight servers from the HPE Cray System Management Redfish emulator: three further

HPE ProLiant/Apollo servers, three HPE Cray EX liquid-cooled compute blades, and one Gigabyte and one Intel server.

Mockups that do not describe a server (PDUs, cooling units, switches, fabric managers) were not included. The mockup of a stock OpenBMC build was excluded because its system and chassis collections are empty. The resulting 26 services span Redfish protocol versions 1.0.2 to 1.20.0 and contain between 68 and 1,811 resources, 16,869 in total. Table 7.1 lists them together with the main results.

7.1.2 Ground truth

For each service, the ground truth is obtained by a full graph crawl that follows *all* references, including Links and Oem subtrees, and evaluates the six queries of Table 4.1 on every resource. Answers inside the BMC's own subtree (below a Manager) and pending-configuration copies referenced by @Redfish.Settings are removed, because they do not describe host hardware. The ground truth contains 2,248 monitored components: 82 processors, 456 memory modules, 108 drives, 134 network interfaces and adapters, 70 power supplies, and 1,398 fans and temperature sensors. Building it took 18,205 requests.

7.1.3 Strategies

Static-legacy

Hand-written navigation for the pre-2020 model: `Systems/*/Processors/*`, `Systems/*/Memory/*`, `Systems/*/SimpleStorage/*#Devices`, `Systems/*/EthernetInterfaces/*`, `Chassis/*/Power#PowerSupplies` and `Chassis/*/Thermal#Fans, Temperatures`. Collections are enumerated (*), so identifiers are not hard-coded. This is how the tools discussed in Section 3.1 work.

Static-modern

The same for the current model: `Storage/*/Drives`, `Chassis/*/NetworkAdapters/*`, `PowerSubsystem/PowerSupplies/*`, `ThermalSubsystem/Fans/*` and `Chassis/*/Sensors/*`.

Static-union

Both recipe sets, sharing a response cache, i.e. the best a maintainer who knows both schema generations can do with fixed paths. The recipes are listed in Appendix A.

Tree-DFS / Tree-BFS

Exhaustive traversal of the containment tree (excluding Links, Oem, Actions and the manager subtree) in depth-first or breadth-first order, evaluating the queries on every node. Tree-BFS is the original approach of the author's tool.

SP-BFS

Algorithm 2 with the type graph of schema bundle 2026.2.

7.1.4 Metrics

For service s and strategy S with answer set $\mathcal{R}_S(s)$ and ground truth $\mathcal{G}(s)$ (both unions over the six queries) we report

$$\text{recall}_S(s) = \frac{|\mathcal{R}_S(s) \cap \mathcal{G}(s)|}{|\mathcal{G}(s)|}, \quad \text{precision}_S(s) = \frac{|\mathcal{R}_S(s) \cap \mathcal{G}(s)|}{|\mathcal{R}_S(s)|}, \quad (7.1)$$

the micro-averaged recall over all services $\sum_s |\mathcal{R}_S(s) \cap \mathcal{G}(s)| / \sum_s |\mathcal{G}(s)|$, the number of HTTP GET requests $C_S(s)$, the pruning ratio ρ of Eq. (5.5), the request rounds $R(c)$ of Eq. (5.6), and the number of requests until the first answer of a single-category query. All strategies have precision 1 on all services, because every strategy evaluates the same query predicates on the bodies it fetches, so precision is not discussed further.

7.2 RQ1: Heterogeneity and Hard-Coded Navigation

Figure 7.1 shows, for every service, the property paths through which Tree-BFS first reached the storage, power, thermal and network components (collection members generalised to $\{\text{id}\}$, embedded array elements to $\#$). Processors and memory are found under two templates each, power under two, network under three, thermal under four and storage under five. Storage is the most diverse: drives are embedded Devices of SimpleStorage on all Dell and most DMTF services, members of Chassis/ $\{\text{id}\}$ /Drives on six of the seven HPE iLO services, members of Storage/ $\{\text{id}\}$ /Drives on the DMTF tower, the HPE DL380a Gen12 and the HPE XL675d A100, and first reached through CompositionService/ResourceBlocks on the Gigabyte server. The deprecation of Power and Thermal splits the services into groups: the Dell, Intel, Gigabyte and most HPE services use only the legacy embedded arrays for power supplies; the DMTF reference services, four HPE iLO 6/7 services and the Cray blades additionally or exclusively expose PowerSubsystem, ThermalSubsystem or Sensors.

The consequences for hard-coded navigation are shown in Figs. 7.2 and 7.3 and Table 7.2. Static-legacy reaches a micro-averaged recall of 0.785 and is complete on only 8 of 26 services, all of them Dell, DMTF-bladed, older HPE or Intel. Static-modern, which is what a developer would write from the current schema documentation, reaches only 0.490 and is complete on *none* of the services: its power recall is 0.23 and its thermal recall 0.26, because most services in the field still use the embedded legacy arrays. The union of both recipe sets reaches 0.990 and is complete on 22 services. It still misses 6 of 8 drives on the

Table 7.1: The 26 services with protocol version, number of resources $|V|$, tree height D and number of monitored components $|\mathcal{G}|$; recall of the static strategies and SP-BFS; and GET requests for a complete discovery. Tree-BFS has the same recall as SP-BFS on every service.

Service	Redfish	$ V $	D	$ \mathcal{G} $	Recall				GET requests		
					legacy	modern	union	SP-BFS	Tree-BFS	union	SP-BFS
DMTF rackmount1	1.15.0	255	7	28	0.75	0.64	1.00	1.00	229	75	91
DMTF tower	1.15.0	544	7	60	0.58	0.88	1.00	1.00	529	126	426
DMTF bladed	1.15.0	82	6	27	1.00	0.15	1.00	1.00	50	34	34
DMTF localstorage	1.15.0	68	6	18	0.78	0.67	1.00	1.00	55	24	25
DMTF smartnic	1.15.0	144	8	24	0.54	0.75	0.92	0.92	131	80	98
DMTF liquid-cooled	1.15.0	235	7	20	0.75	0.80	1.00	1.00	209	71	87
Dell R720	1.0.2	331	6	23	1.00	0.48	1.00	1.00	241	22	22
Dell R740	1.0.2	365	6	16	1.00	0.31	1.00	1.00	281	17	17
Dell T630	1.0.2	313	6	27	1.00	0.15	1.00	1.00	242	19	19
Dell C6320	1.0.2	251	6	12	1.00	0.33	1.00	1.00	216	15	15
Dell FC630	1.0.2	283	6	41	1.00	0.24	1.00	1.00	253	26	26
HPE DL325 G10+	1.6.0	877	6	70	0.94	0.36	1.00	1.00	647	39	60
HPE DL340e	1.20.0	1222	7	76	0.84	0.43	1.00	1.00	958	78	88
HPE DL360	1.20.0	1361	8	106	0.92	0.41	0.94	1.00	1076	57	78
HPE DL360 G12	1.20.0	942	8	91	0.88	0.52	1.00	1.00	700	99	128
HPE DL365 G10+	1.20.0	1811	8	100	0.90	0.40	0.94	1.00	1410	54	91
HPE DL380a	1.20.0	1487	8	112	0.83	0.44	1.00	1.00	1222	123	156
HPE DL380a G12	1.20.0	1714	7	181	0.83	0.37	0.96	0.98	1257	174	220
HPE DL325 (CSM)	1.6.0	852	6	66	1.00	0.32	1.00	1.00	618	33	35
HPE XL675d A100	1.6.0	454	8	128	0.98	0.33	1.00	1.00	428	56	63
HPE XL675d A40	1.6.0	895	6	147	0.99	0.27	1.00	1.00	677	52	91
Cray EX235a	1.2.0	389	6	116	0.52	0.67	1.00	1.00	354	225	239
Cray EX425	1.2.0	483	6	136	0.62	0.67	1.00	1.00	455	339	347
Cray EX4252	1.2.0	1182	6	534	0.58	0.64	1.00	1.00	1096	931	939
Gigabyte	1.7.0	231	6	45	0.98	0.49	1.00	1.00	182	40	56
Intel	1.1.0	98	6	44	1.00	0.18	1.00	1.00	78	27	27
Total / micro-average				2248	0.78	0.49	0.99	1.00	13594	2836	3478

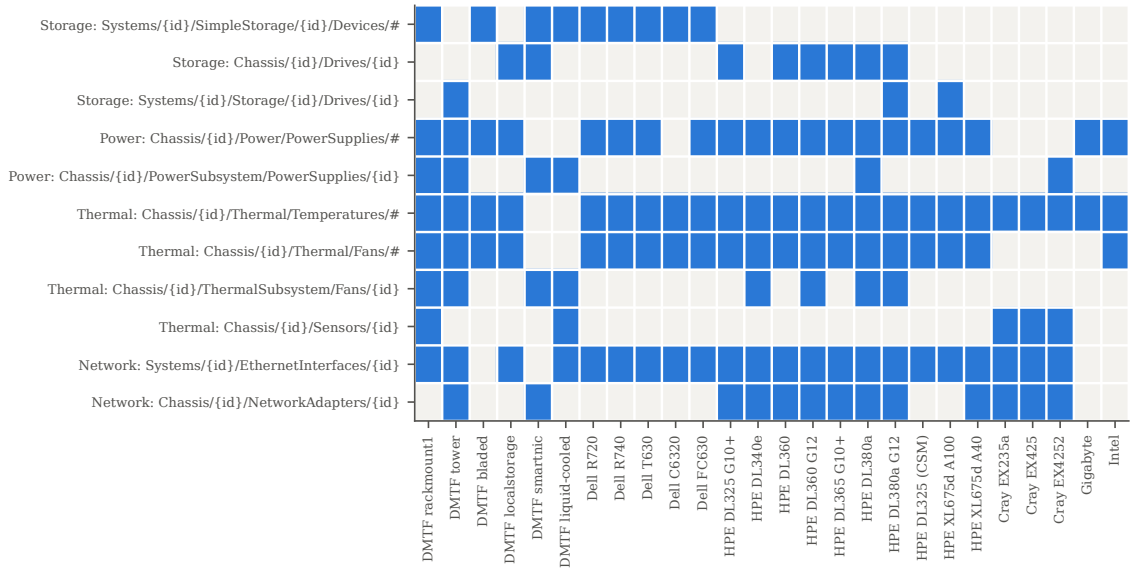


Figure 7.1: Where monitored components live. Rows are property-path templates (occurring in at least two services), columns are services; a filled cell means the service exposes components of that category under that template.

Table 7.2: Micro-averaged recall per category and overall.

Strategy	Processor	Memory	Storage	Network	Power	Thermal	All
Static-legacy	0.988	1.000	0.370	0.761	0.771	0.737	0.785
Static-modern	0.988	1.000	0.444	0.993	0.229	0.263	0.490
Static-union	0.988	1.000	0.815	0.993	1.000	1.000	0.990
Tree-BFS	0.988	1.000	0.963	0.993	1.000	1.000	0.997
SP-BFS	0.988	1.000	0.963	0.993	1.000	1.000	0.997

HPE DL360 and DL365 and 8 of 14 on the DL380a Gen12, because these drives are linked only from `Chassis/{id}/Drives`, a location that neither recipe set visits, so its storage recall is 0.815. These are exactly the failures that the author’s tool experienced in practice. Each of them was silent, and each would have required another vendor-specific path to fix.

Answer to RQ1. Real services differ substantially in where they place monitored components, mainly along the two axes H2 (schema generation) and H3 (placement). Hard-coded navigation for a single schema generation loses between 21 % and 51 % of the components; even a union of both generations loses components on three of the ten HPE ProLiant and Apollo servers.

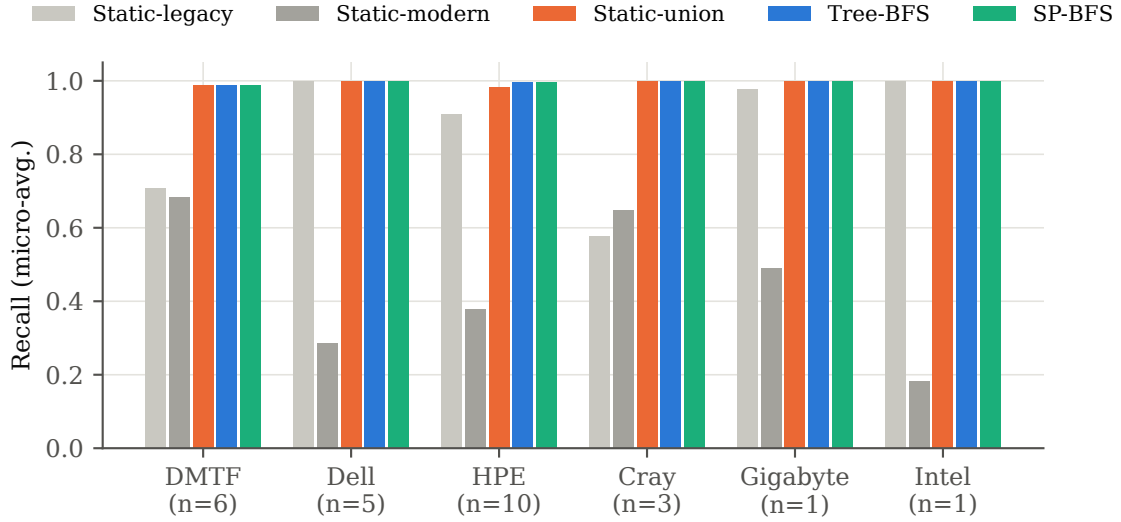


Figure 7.2: Micro-averaged recall per vendor family (n = number of services).

7.3 RQ2: Completeness of Type-Based Tree Search

Tree-BFS and SP-BFS find 2,242 of the 2,248 ground-truth components (recall 0.997) and are complete on 24 of 26 services, independent of vendor and protocol version (Table 7.1). The six components they miss are reachable *only* through non-containment references:

- On the DMTF smartnic mockup, a network adapter of a SmartNIC card chassis and its FPGA processor are linked only from the Links of a system’s network device function. The card’s chassis is not a member of the chassis collection.
- On the HPE DL380a Gen12, four drives are linked only from an HPE OEM resource directory (Oem/Hpe/Instances), not from any standard navigation property.

Both are arguably defects of the respective service, since DSP0266 expects every resource to be reachable through the resource tree. They illustrate the price of restricting the search to containment edges. Following Links as well would recover them, but it would turn the almost-tree into a dense graph (every processor links back to its chassis, every drive to its volumes and controllers) and multiply the request count.

Answer to RQ2. Yes. Matching `@odata.type` over the containment tree finds 99.7 % of the components on all vendors and versions without a single line of vendor-specific navigation. The remainder is attributable to resources that the services do not place in the containment tree.

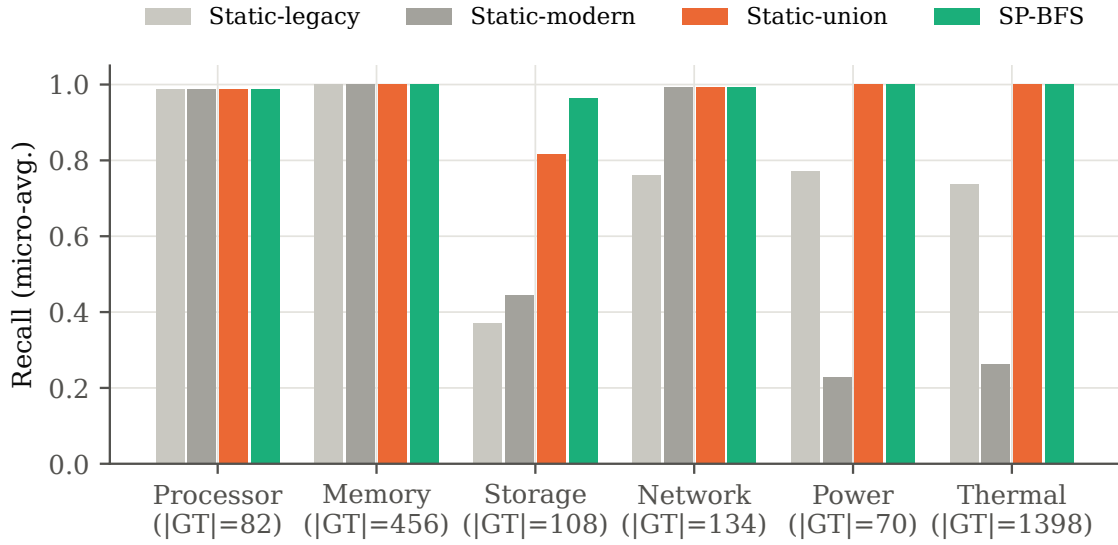


Figure 7.3: Micro-averaged recall per component category.

7.4 RQ3: Cost of Schema-Pruned Search

Figures 7.4 and 7.5a compare the request counts. Exhaustive Tree-BFS needs 13,594 requests for the 26 services (523 per service on average), because it fetches every log entry, schema file, registry and account. SP-BFS needs 3,478 (134 per service), a reduction by a factor of 3.9 overall. The per-service pruning ratio ρ has a geometric mean of 0.20, i.e. a five-fold reduction. It ranges from $\rho = 0.057$ (HPE DL325 CSM, 35 instead of 618 requests) and $\rho = 0.060$ (Dell R740, 17 instead of 281) to $\rho = 0.86$ on the Cray EX4252. The EX4252 is a blade enclosure whose resources are mostly the monitored temperature sensors themselves, so there is little irrelevant content to prune. Recall is unaffected on every service (Theorem 5.1): all 26 services behaved as schema-conformant in the sense that no target was reached only through a pruned edge.

Table 7.3 lists the properties at which pruning happened most often. At the service root, Managers, SessionService, AccountService, EventService, Tasks, Registries, UpdateService, JsonSchemas and CertificateService are cut on almost every service. Below systems, the BIOS, boot, secure-boot and log services are cut; below chassis, PCIe devices and slots, assemblies and controls. The most frequent cuts, however, are references out of the legacy embedded arrays (Power.Voltages, Thermal.Fans, Power.PowerSupplies). Several vendors attach dereferenceable sensor URIs to each embedded element; the schema declares these arrays as embedded objects, not navigation properties, so their contents are already known and the links need not be followed.

Compared with the hand-maintained Static-union, SP-BFS issues 23 % more requests in total (3,478 versus 2,836; geometric mean of the per-service ratio 1.22). On 7 services it issues exactly as many requests. The largest overheads are on the DMTF tower (3.4×),

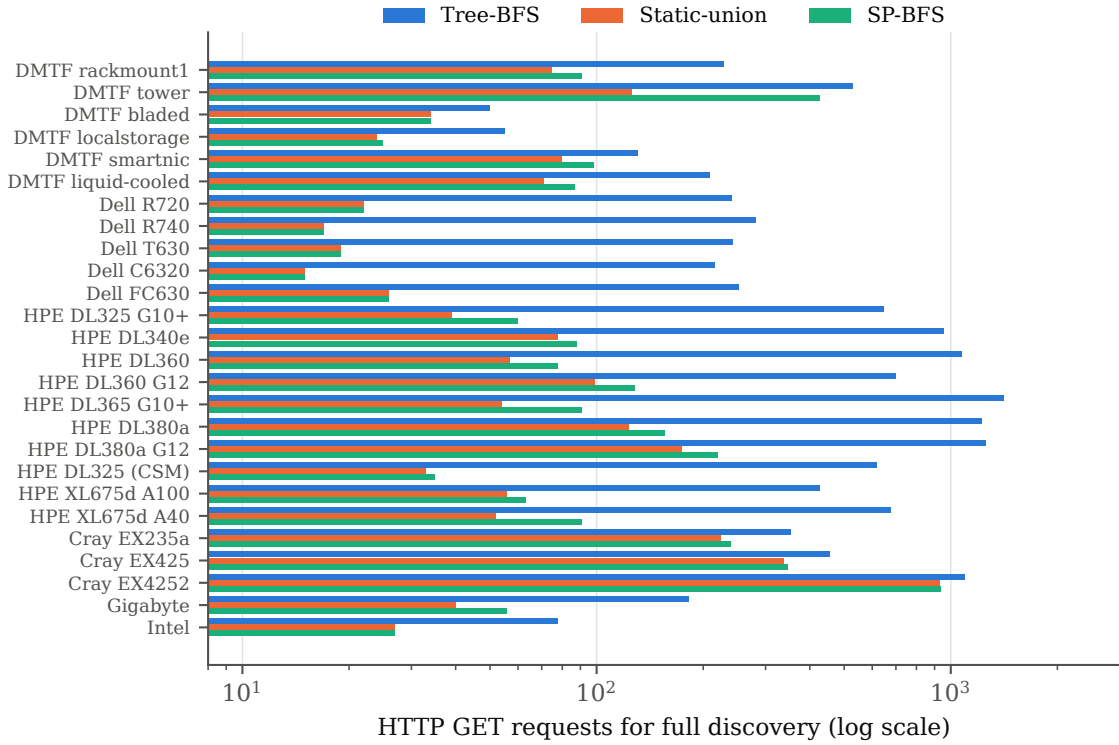


Figure 7.4: GET requests needed to discover all six categories on each service (log scale).

where SP-BFS descends into the 256 SubProcessors (cores and threads) of its eight CPUs because a processor *can* contain processors, and on three HPE servers (1.5–1.75 $\times$), where it descends into the network device functions and ports of the network adapters, because the schema allows EthernetInterface resources below a NetworkDeviceFunction. By Proposition 5.3, no algorithm that relies only on the schema can avoid these requests. Static-union avoids them only because its author knows, for this particular set of services, where drives do *not* occur. The same knowledge makes it miss the drives of the three HPE servers above.

Single-category queries. When only one category is requested, pruning is much stronger, because fewer anchor types keep fewer branches alive. Summed over all services that have the category, SP-BFS needs 162 requests to find all power supplies (7 per service, against 12,569 for Tree-BFS), 255 for all drives and 616 for all processors.

Computation. Building the type graph from the 2026.2 bundle takes about 1 s and is done once. The client-side processing of SP-BFS (reference extraction, query evaluation and pruning decisions, with the mockup file reads) takes 12 ms per service on average and at most 72 ms. This is negligible compared with a single BMC request.

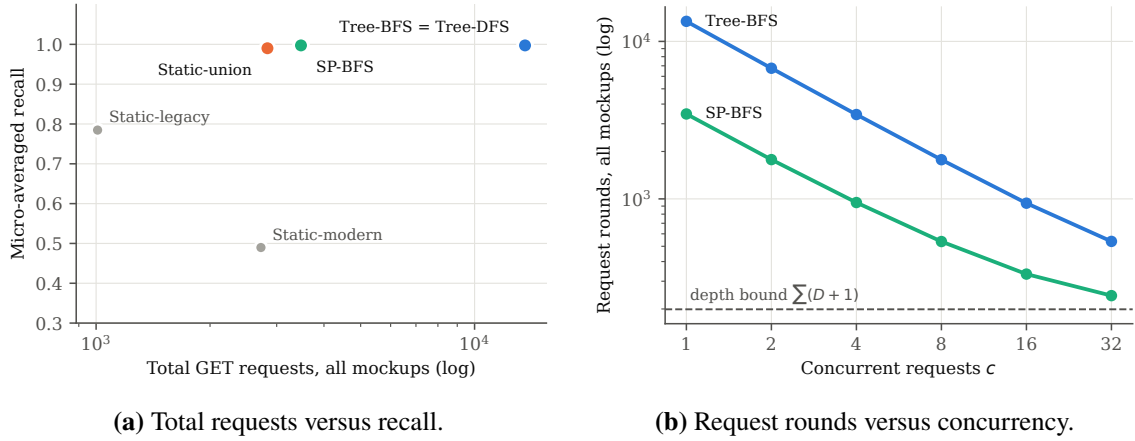


Figure 7.5: Cost trade-off (a) and level-synchronous parallelism (b), summed over all 26 services.

Table 7.3: Most frequently pruned (type, property) pairs, summed over all services.

Type.property	Cuts	Type.property	Cuts
Power.Voltages	123	ServiceRoot.Registries	23
Thermal.Fans	42	ServiceRoot.UpdateService	19
Fan.Assembly	36	ServiceRoot.JsonSchemas	19
ComputerSystem.Bios	27	ComputerSystem.LogServices	18
ServiceRoot.Managers	26	ComputerSystem.Boot	18
ServiceRoot.SessionService	26	ServiceRoot.CertificateService	16
ServiceRoot.AccountService	26	ComputerSystem.SecureBoot	16
ServiceRoot.EventService	25	Thermal.Temperatures	16
ServiceRoot.Tasks	24	Power.PowerSupplies	15

Answer to RQ3. Yes. Schema pruning removes on average 80 % of the requests of exhaustive tree search (per-service geometric mean) without losing any component, and brings the cost to within about a quarter of a hand-written client that knows both schema generations. Unlike that client, it needs no vendor-specific code and is more complete.

7.5 RQ4: Latency, Parallelism and Amortisation

Time to first answer. Figure 7.6 compares, for each of the 138 (service, category) pairs with a non-empty ground truth, the number of requests a single-category traversal needs before it finds its first answer. Contrary to the intuition that BFS finds things early, exhaustive Tree-BFS is *slower* than Tree-DFS in 78 % of the cases (median 234 versus 48 requests): the shallow levels of a Redfish tree are wide and filled with irrelevant resources, and BFS must sweep all of them before it reaches the components at depth 4–6. SP-BFS removes these levels and finds the first answer after a median of 9 requests (5 for power and thermal, 13 for memory and network).

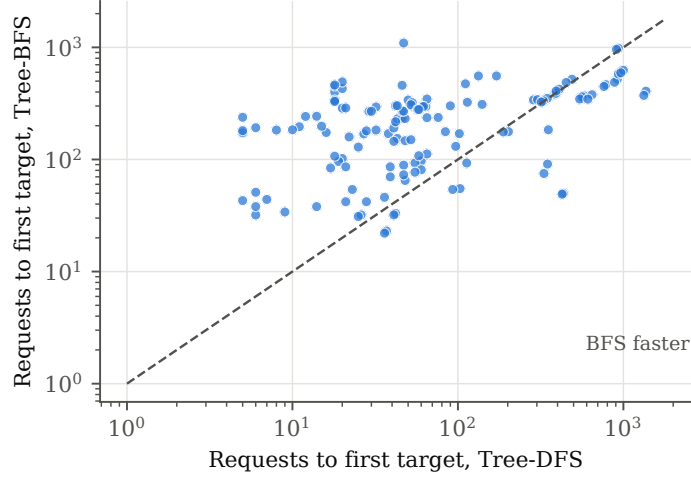


Figure 7.6: Requests until the first answer of a single-category query, Tree-BFS versus Tree-DFS (one point per service and category; points above the diagonal favour DFS).

Parallelism. Figure 7.5b shows the request rounds $R(c)$ of Eq. (5.6), summed over all services. Because the containment trees have only 6–8 levels, the level barrier costs little: SP-BFS needs 3,461 rounds at $c = 1$, 948 at $c = 4$, 536 at $c = 8$ and 333 at $c = 16$ (12.8 rounds per service on average), approaching the lower bound of 199 rounds given by the number of tree levels.

Latency model. Mockups carry no timing information, so we translate rounds into time with the per-request latency reported for iDRAC8 in production by MonSTer, $\bar{\ell} = 4.29$ s [38]. This is a pessimistic value measured under concurrent load. Under this model, an average service is discovered by Tree-BFS in 523×4.29 s ≈ 37 minutes sequentially, by SP-BFS in about 9.6 minutes, and by SP-BFS with $c = 8$ in 20.6×4.29 s ≈ 88 seconds, provided the BMC sustains eight concurrent requests without slowing down. Real BMCs slow down under concurrency, so the parallel figure is a lower bound, but the ordering of the strategies does not depend on $\bar{\ell}$.

Amortisation. After discovery, each polling cycle fetches the poll set: on average 47.8 resources per service for SP-BFS and Tree-BFS, and 47.2 for the less complete Static-union. By Eq. (5.7), the discovery cost of 134 requests is spread over the polling cycles: after ten cycles it adds only 13 requests per cycle, and with event-driven rediscovery (Algorithm 3) it recurs only when the inventory changes. Re-running an exhaustive Tree-BFS in every cycle, as a naive implementation of the original tool would, costs eleven times the polling set per cycle.

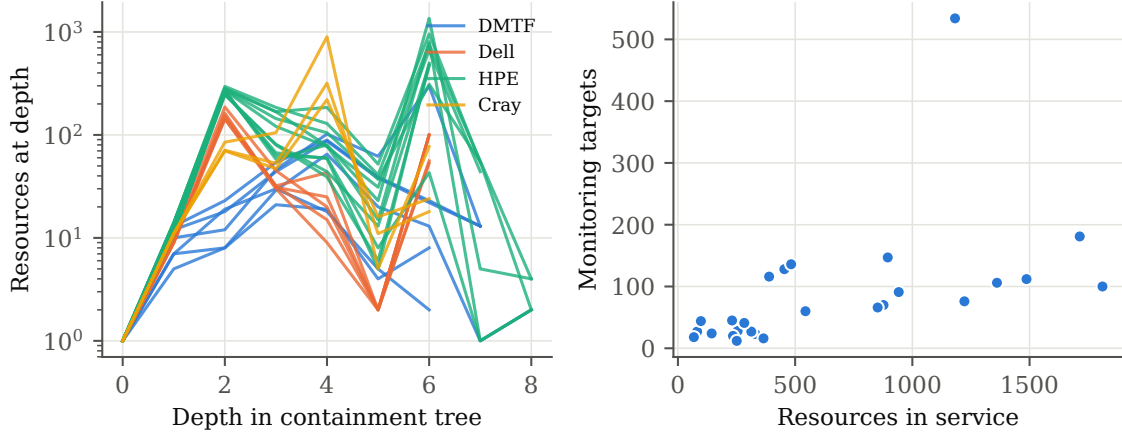


Figure 7.7: Shape of the containment trees: resources per depth for each service, coloured by vendor family (left), and monitored components versus service size (right).

Answer to RQ4. Unpruned BFS has poor time-to-first-answer on Redfish because the upper tree levels are wide and irrelevant; pruning fixes this (median 9 requests). The shallow trees make level-synchronous parallelism effective, and the discovery cost is amortised after a few polling cycles when discovery and polling are separated.

7.6 Shape of Redfish Resource Trees

The containment graphs are shallow and broad (Fig. 7.7): all 26 services have height $6 \leq D \leq 8$, and the number of resources per level peaks at depth 2–5. The share of vertices with more than one containment parent (Eq. (4.3)) is 3.1 % overall. It is highest on the HPE iLO 6/7 services (up to 9.3 %, e.g. drives linked from both chassis and storage) and on the Gigabyte server (12.1 %, resources exposed both under a system and a composition resource block). The almost-tree view is therefore justified, and the *seen* set of Algorithm 2 deduplicates the remaining multi-parent vertices. Services with more resources tend to have more monitored components, but the relation is loose (right panel), because the bulk of the resources is logs, registries and schemas. The number of monitored components alone therefore predicts discovery cost poorly, which is one more argument for pruning by type rather than by size. Finally, 120 resources on 13 services (mostly HPE and Cray) have no `@odata.type`; SP-BFS expands them conservatively.

7.7 Discussion

7.7.1 URI patterns versus type-based search

Since protocol version 1.6.0, conforming services must implement the URI patterns published in the schema (Section 3.2). One might therefore replace discovery by expanding the patterns. The patterns do not remove the search, however. Each pattern contains identifier variables that must be bound by reading collections, and the patterns of one type are alternatives (21 for Processor, 8 for Drive), so the client still has to probe which of them the service implements. This amounts to a tree search guided by string templates instead of types. The Dell services in our data, at protocol version 1.0.2, predate the patterns and use URIs such as `/redfish/v1/Chassis/System.Embedded.1/Sensors/Fans/0x17||Fan.Embedded.1` that match none of them. URI patterns can still help SP-BFS: where a service declares version $\geq 1.6.0$, a pattern proves that a type cannot occur below a certain path, which could tighten the type graph further.

7.7.2 Why not follow Links or use \$expand?

Following Links would recover the six components that no containment path reaches. But cross references connect almost every hardware resource to its chassis and system, so the traversal would visit most of the graph again (the ground-truth crawl needed 18,205 requests, 5.2 times as many as SP-BFS). A practical compromise is to follow only those cross references whose target type is an anchor type, which the schema also describes. \$expand would reduce the number of round trips further, but it is optional and inconsistently implemented, so it can only be an optimisation on top of the traversal, not a replacement.

7.7.3 Threats to validity

Internal validity. The ground truth is defined by the same query predicates as the strategies, so recall measures *reachability* of the components, not the correctness of the query definitions. We checked the query definitions manually on several services. The categories and the predicate that excludes cores and threads reflect the needs of the author’s monitoring tool; other choices would change the absolute numbers, not the comparison. The static baselines were written by us from the schema. A developer with different habits might write different paths, but Static-union already covers both schema generations and all paths the tools of Section 3.1 use.

External validity. The 26 services come from six vendor families but are dominated by HPE (13 services including Cray). Lenovo, Supermicro, Fujitsu, Inspur and current Dell

iDRAC 9 firmware are not represented because no public mockups were available; several of them are known to have their own placement conventions. Mockups are complete recordings, but some may have been edited by their publishers to remove serial numbers or to shorten logs, which affects Tree-BFS cost (and makes it an underestimate) but not the answer sets.

Construct validity. Request count is a proxy for time and BMC load. The latency model uses a single published average, which ignores the difference between cheap and expensive resources (collections with many members, sensor readings requiring bus transactions) and the degradation of BMCs under concurrency. Measurements on live hardware are needed to turn the request and round counts into reliable wall-clock numbers.

Chapter 8

Conclusion

8.1 Summary

This thesis started from a practical difficulty met while building a Redfish hardware monitoring tool: deciding which REST resource to call for storage, network, power, processors and the other components, when the answer depends on the vendor, the firmware generation and the schema version of each server. Instead of maintaining a growing table of vendor-specific paths, the tool converts the Redfish service into a tree and finds components by breadth-first search on their schema type. The thesis developed this idea into a method with guarantees and evaluated it on real service data.

- We modelled a Redfish service as a typed resource graph whose containment edges form an almost-tree (3.1 % multi-parent vertices, height 6–8), and monitored component classes as declarative queries over types, embedded locations and predicates.
- We showed on 26 services from six vendor families and protocol versions 1.0.2–1.20.0 that placement heterogeneity is real: hard-coded navigation for one schema generation finds only 49–78 % of the 2,248 components, and even a union of both generations misses drives on three HPE servers.
- Type-based tree search finds 99.7 % of the components on every vendor and version with no vendor-specific code; the rest is reachable only through Links or OEM extensions.
- Schema-pruned BFS (SP-BFS) uses a type reachability graph derived automatically from the DMTF schema bundle to skip every branch that cannot contain a requested type. It is provably sound on schema-conformant services and fetches only resources that any schema-only algorithm must fetch. It lost no component on any service, cut requests by a factor of five per service (geometric mean) relative to exhaustive BFS, and came within a factor of 1.22 of a hand-maintained static client.

- Unpruned BFS reaches its first answer later than DFS on Redfish, because the shallow levels are wide and irrelevant; pruning reduces the median time to first answer from 234 to 9 requests. The shallow trees make level-synchronous parallel fetching effective, and separating discovery from polling amortises the remaining cost.

The answer to the research problem is therefore positive: a monitoring client can locate all instances of a hardware class in an arbitrary Redfish service without vendor-specific navigation code, at a request cost close to that of a hand-written client, by letting the published schema guide a breadth-first search. For practitioners the recommendation is simple: describe *what* you monitor, not *where* it lives, and let the schema tell the client which links are worth following.

8.2 Limitations

The evaluation relies on recorded services, so wall-clock effects (latency, throttling, degradation under concurrency) are modelled rather than measured. The dataset is dominated by HPE and lacks Lenovo, Supermicro and current Dell firmware. SP-BFS is sound only relative to the containment tree; components that a service exposes solely via Links or OEM subtrees are not found. The type graph over-approximates containment because the JSON Schema does not distinguish contained resources from referenced ones outside Links, which costs requests (e.g. below network device functions) but not completeness.

8.3 Future Work

- **Live measurements.** Run SP-BFS and the baselines against a fleet of live BMCs to measure latency distributions, the effect of concurrency on BMC response times, and the resulting optimal c per vendor.
- **Tighter type graphs.** Use the CSDL containment annotation `ContainsTarget`, and the URI patterns of services at version $\geq 1.6.0$, to remove reference-only edges from H . In the 2026.2 bundle only 214 of 1,030 navigation-property declarations carry `ContainsTarget`, so this requires either schema improvements or a learned model of which edges are containment in practice.
- **Predicate-aware pruning.** Propagate content predicates such as “not a core or thread” into the pruning rule, e.g. by learning, per vendor, which subtrees never contain matching instances, with a periodic exhaustive audit to preserve completeness.
- **Selective cross references.** Follow a Links reference only if its declared target type

is an anchor type, to recover the components that are not in the containment tree at little extra cost.

- **Deduplication across model generations.** Merge the legacy embedded and the new resource representations of the same power supply or fan by Location, SerialNumber and MemberId.
- **Server-side assistance.** Combine SP-BFS with \$expand and \$select where a service supports them, and with Redfish event subscriptions and telemetry reports for rediscovery and polling.

Bibliography

- [1] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165, New Orleans, LA, USA, 2014. IEEE. doi: 10.1109/SC.2014.18.
- [2] Ghazanfar Ali, Jon Hass, Alan Sill, Elham Hojati, Tommy Dang, and Yong Chen. Redfish-Nagios: A scalable out-of-band data center monitoring framework based on Redfish telemetry model. In *Proceedings of the Fifth International Workshop on Systems and Network Telemetry and Analytics (SNTA '22)*, pages 3–11, Minneapolis, MN, USA, 2022. ACM. doi: 10.1145/3526064.3534108.
- [3] Ghazanfar Ali, Jon Hass, Alan Sill, Elham Hojati, Tommy Dang, and Yong Chen. A scalable, distributed monitoring framework for HPC clusters using Redfish-Nagios integration. Research Square preprint (CC BY 4.0), April 2023.
- [4] Douglas S. Almeida, Péricles Rezende Barros, George Acioli Júnior, and Thiago A. M. Euzébio. Implementation guide for secure RESTful asset management APIs under IEC 62443-4-2: A Redfish case study. *IEEE Access*, 14:42015–42038, 2026. doi: 10.1109/ACCESS.2026.3674966.
- [5] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internet Engineering Task Force, January 2005. URL <https://www.rfc-editor.org/rfc/rfc3986>. STD 66.
- [6] Anthony J. Bonkoski, Russ Bielawski, and J. Alex Halderman. Illuminating the security issues surrounding lights-out server management. In *7th USENIX Workshop on Offensive Technologies (WOOT '13)*, Washington, D.C., USA, 2013. USENIX Association. URL <https://www.usenix.org/conference/woot13/workshop-program/presentation/bonkoski>.

- [7] T. Bray. The JavaScript Object Notation (JSON) data interchange format. RFC 8259, Internet Engineering Task Force, December 2017. URL <https://www.rfc-editor.org/rfc/rfc8259>. STD 90.
- [8] P. Bryan, K. Zyp, and M. Nottingham. JavaScript Object Notation (JSON) pointer. RFC 6901, Internet Engineering Task Force, April 2013. URL <https://www.rfc-editor.org/rfc/rfc6901>.
- [9] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). RFC 1157, Internet Engineering Task Force, May 1990. URL <https://www.rfc-editor.org/rfc/rfc1157>.
- [10] Zitai Chen and David Oswald. PMFault: Faulting and bricking server CPUs through management interfaces. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):1–23, 2023. doi: 10.46586/tches.v2023.i2.1-23.
- [11] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 200–209, Cairo, Egypt, 2000. Morgan Kaufmann.
- [12] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1–7):161–172, 1998. doi: 10.1016/S0169-7552(98)00108-1. Proceedings of the Seventh International World Wide Web Conference (WWW7).
- [13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.
- [14] Vedran Dakić, Karlo Bertina, Jasmin Redžepagić, and Damir Regvart. The RedFish API and vSphere hypervisor API: A unified framework for policy-based server monitoring. *Electronics*, 13(23):4624, 2024. doi: 10.3390/electronics13234624.
- [15] DMTF. Redfish white paper. DMTF Informative Document DSP2044, Version 1.0.0, Distributed Management Task Force (DMTF), August 2015. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP2044_1.0.0.pdf. Published 2015-08-04.
- [16] DMTF. Redfish Mockup Creator. GitHub repository, 2016. URL <https://github.com/DMTF/Redfish-Mockup-Creator>. Walks a live Redfish service and records every resource as a mockup.

- [17] DMTF. Redfish Mockup Server. GitHub repository, 2016. URL <https://github.com/DMTF/Redfish-Mockup-Server>. Python tool serving a static Redfish mockup over HTTP(S).
- [18] DMTF. python-redfish-library. GitHub repository, 2016. URL <https://github.com/DMTF/python-redfish-library>. Python 3 client library for Redfish services.
- [19] DMTF. Redfish Service Validator. GitHub repository, 2016. URL <https://github.com/DMTF/Redfish-Service-Validator>. Checks conformance of a Redfish service against the Redfish CSDL schema.
- [20] DMTF. Redfish and OData white paper. DMTF Informative Document DSP2052, Version 1.0.0, Distributed Management Task Force (DMTF), April 2018. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP2052_1.0.0.pdf. Published 2018-04-05.
- [21] DMTF. Redfish host interface specification. DMTF Standard DSP0270, Version 1.3.1, Distributed Management Task Force (DMTF), August 2023. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP0270_1.3.1.pdf. Published 2023-08-03.
- [22] DMTF. Platform level data model (PLDM) for Redfish device enablement. DMTF Standard DSP0218, Version 1.2.0, Distributed Management Task Force (DMTF), June 2024. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP0218_1.2.0.pdf. Published 2024-06-21.
- [23] DMTF. Redfish user guide. DMTF Informational Document DSP2060, Version 1.1.0, Distributed Management Task Force (DMTF), August 2024. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP2060_1.1.0.pdf. Published 2024-08-01.
- [24] DMTF. Redfish specification. DMTF Standard DSP0266, Version 1.25.0, Distributed Management Task Force (DMTF), August 2026. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.25.0.pdf. Published 2026-08-07.
- [25] DMTF. Redfish data model specification. DMTF Standard DSP0268, Version 2026.2, Distributed Management Task Force (DMTF), August 2026. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP0268_2026.2.pdf. Published 2026-08-07.
- [26] DMTF. Redfish interoperability profiles. DMTF Standard DSP0272, Version 1.10.0, Distributed Management Task Force (DMTF), April 2026. URL <https://www.dmtf.org>.

org/sites/default/files/standards/documents/DSP0272_1.10.0.pdf. Published 2026-04-03.

- [27] DMTF. Redfish resource and schema guide. DMTF Informational Document DSP2046, Version 2026.2, Distributed Management Task Force (DMTF), August 2026. URL https://www.dmtf.org/sites/default/files/standards/documents/DSP2046_2026.2.pdf. Published 2026-08-07.
- [28] DMTF. Redfish schema bundle (DSP8010). Distributed Management Task Force (DMTF), CSDL, JSON Schema and OpenAPI files, 2026. URL <https://www.dmtf.org/standards/redfish>. Released together with DSP0268/DSP2046 version 2026.2; see <https://www.dmtf.org/standards/redfish>.
- [29] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 84–93. IEEE, 2014. doi: 10.1109/CSMR-WCRE.2014.6747228.
- [30] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*, 100:27–43, 2015. doi: 10.1016/j.jss.2014.10.014. Local copy is the TU Delft technical report TUD-SERG-2014-017 version.
- [31] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL <https://ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [32] Glauco Estácio Gonçalves, Daniel Rosendo, Leylane Ferreira, Guto Leoni Santos, Demis Gomes, André Moreira, Judith Kelner, Djamel Sadok, Mattias Wildeman, and Patrícia Takako Endo. A standard to rule them all: Redfish. *IEEE Communications Standards Magazine*, 3(2):36–43, 2019. doi: 10.1109/MCOMSTD.2019.1800045.
- [33] D. Harrington, R. Presuhn, and B. Wijnen. An architecture for describing simple network management protocol (SNMP) management frameworks. RFC 3411, Internet Engineering Task Force, December 2002. URL <https://www.rfc-editor.org/rfc/rfc3411>. STD 62.
- [34] Elham Hojati, Yong Chen, and Alan Sill. Benchmarking automated hardware management technologies for modern data centers and cloud environments. In *Proceedings of the 10th International Conference on Utility and Cloud Computing (UCC '17)*, pages 195–196, Austin, TX, USA, 2017. ACM. doi: 10.1145/3147213.3149212.

- [35] Elham Hojati, Jon Hass, Alan Sill, and Yong Chen. Redfish Green500 Benchmark (RGB): Towards automation of the Green500 process for data centers. In *2020 IEEE Green Technologies Conference (GreenTech)*, pages 47–52, Oklahoma City, OK, USA, 2020. IEEE. doi: 10.1109/GreenTech46478.2020.9289729.
- [36] Intel, Hewlett-Packard, NEC, and Dell. *IPMI – Intelligent Platform Management Interface Specification, Second Generation, v2.0*, document revision 1.1 edition, October 2013. URL <https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ipmi-second-gen-interface-spec-v2-rev1-1.pdf>. v2.0 first released 2004; revision 1.1 dated 1 October 2013.
- [37] jenningsloy318. redfish_exporter: A Prometheus exporter for Redfish-based hardware. GitHub repository, 2019. URL https://github.com/jenningsloy318/redfish_exporter.
- [38] Jie Li, Ghazanfar Ali, Ngan Nguyen, Jon Hass, Alan Sill, Tommy Dang, and Yong Chen. MonSTer: An out-of-the-box monitoring tool for high performance computing systems. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 119–129, Kobe, Japan, 2020. IEEE. doi: 10.1109/CLUSTER49012.2020.00022. Accepted manuscript via NSF PAR: <https://par.nsf.gov/servlets/purl/10214964>.
- [39] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service API evolution affect clients? In *2013 IEEE 20th International Conference on Web Services (ICWS)*, pages 300–307, Santa Clara, CA, USA, 2013. IEEE. doi: 10.1109/ICWS.2013.48.
- [40] Antonio Libri, Andrea Bartolini, and Luca Benini. DiG: Enabling out-of-band scalable high-resolution monitoring for data-center analytics, automation and control (extended), 2018. URL <https://arxiv.org/abs/1806.02698>.
- [41] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching, Part II*, volume 30 of *Annals of the Computation Laboratory of Harvard University*, pages 285–292, Cambridge, MA, 1959. Harvard University Press.
- [42] Marc Najork and Janet L. Wiener. Breadth-first crawling yields high-quality pages. In *Proceedings of the 10th International Conference on World Wide Web (WWW '01)*, pages 114–118, Hong Kong, 2001. ACM. doi: 10.1145/371920.371965.
- [43] NSF Cloud and Autonomic Computing Center, Texas Tech University. MonSTer:

- Monitoring tool for HPC metrics from batch scheduler and BMC resources. GitHub repository, 2020. URL <https://github.com/nsfcac/MonSter>.
- [44] OpenAPI Initiative. OpenAPI specification, version 3.1.1. Linux Foundation / OpenAPI Initiative, October 2024. URL <https://spec.openapis.org/oas/v3.1.1.html>. Released 2024-10-24. Redfish (DSP0266 §5.4.5) references OpenAPI v3.0.
 - [45] OpenBMC Project. bmcweb: OpenBMC webserver (Redfish, KVM, GUI and D-Bus). GitHub repository, 2018. URL <https://github.com/openbmc/bmcweb>.
 - [46] OpenBMC Project. OpenBMC: An open source firmware stack for baseboard management controllers. Linux Foundation project, GitHub organisation, 2026. URL <https://github.com/openbmc/openbmc>.
 - [47] OpenStack. Sushy: A Python library to communicate with Redfish based systems. OpenStack project, mirrored on GitHub, 2017. URL <https://opendev.org/openstack/sushy>.
 - [48] Michael Pizzo, Ralf Handl, and Mark Biamonte. OData JSON format version 4.01. Oasis standard, OASIS, May 2020. URL <https://docs.oasis-open.org/odata/odata-json-format/v4.01/os/odata-json-format-v4.01-os.html>. 11 May 2020.
 - [49] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. OData version 4.01. part 1: Protocol. Oasis standard, OASIS, April 2020. URL <https://docs.oasis-open.org/odata/odata/v4.01/os/part1-protocol/odata-v4.01-os-part1-protocol.html>. 23 April 2020.
 - [50] SNIA. Swordfish scalable storage management API specification. SNIA Technical Specification Version 1.2.8, Storage Networking Industry Association (SNIA), 2025. URL https://www.snia.org/sites/default/files/technical-work/swordfish/release/v1.2.8/pdf/Swordfish_v1.2.8_Specification.pdf.
 - [51] S. M. Sohan, Craig Anslow, and Frank Maurer. A case study of web API evolution. In *2015 IEEE World Congress on Services*, pages 245–252, New York, NY, USA, 2015. IEEE. doi: 10.1109/SERVICES.2015.43.
 - [52] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. JSON Schema: A media type for describing JSON documents. Internet-Draft draft-bhutton-json-schema-01, Internet Engineering Task Force, June 2022. URL <https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-01>. JSON Schema Draft 2020-12 (core); work in progress.

Appendix A

Static Navigation Recipes and Queries

The static baselines of Chapter 7 are defined as lists of steps. A property name follows that property (a single link or an array of links), * enumerates the Members of a collection, and #Prop enumerates the elements of an embedded array without a further request. Static-union is the concatenation of both lists per category, evaluated with a shared response cache.

Listing A.1: Static navigation recipes (rfbfs/discover.py).

```
STATIC_LEGACY = {
    "Processor": [["Systems", "*", "Processors", "*"]],
    "Memory":    [["Systems", "*", "Memory", "*"]],
    "Storage":   [["Systems", "*", "SimpleStorage", "*", "#Devices"]],
    "Network":   [["Systems", "*", "EthernetInterfaces", "*"]],
    "Power":     [["Chassis", "*", "Power", "#PowerSupplies"]],
    "Thermal":   [["Chassis", "*", "Thermal", "#Fans"],
                  ["Chassis", "*", "Thermal", "#Temperatures"]],
}

STATIC_MODERN = {
    "Processor": [["Systems", "*", "Processors", "*"]],
    "Memory":    [["Systems", "*", "Memory", "*"]],
    "Storage":   [["Systems", "*", "Storage", "*", "Drives"]],
    "Network":   [["Systems", "*", "EthernetInterfaces", "*"],
                  ["Chassis", "*", "NetworkAdapters", "*"]],
    "Power":     [["Chassis", "*", "PowerSubsystem", "PowerSupplies", "*"]],
    "Thermal":   [["Chassis", "*", "ThermalSubsystem", "Fans", "*"],
                  ["Chassis", "*", "Sensors", "*"]],
}
```

Listing A.2: Declarative resource queries (rfbfs/queries.py, abridged).

```
QUERIES = {
    "Processor": Query("Processor", {"Processor"}, predicate=_socket),
    "Memory":    Query("Memory", {"Memory"}),
    "Storage":   Query("Storage", {"Drive"}, {"("SimpleStorage", "Devices")}),
```

```
"Network": Query("Network", {"EthernetInterface", "NetworkAdapter"}),
"Power":    Query("Power", {"PowerSupply"}, {"Power", "PowerSupplies"}),
"Thermal":  Query("Thermal", {"Fan", "Sensor"},
                  {"Thermal", "Fans"}, {"Thermal", "Temperatures"}),
                  _temperature_sensor),
}
EXCLUDED_TYPES = {"ManagerCollection", "Manager"}
```

Appendix B

Reproducibility

B.1 Data sources

All services and the schema bundle were obtained from public repositories on 19 September 2026 (Table B.1). The mockup directories used are listed in `code/run_experiments.py`.

Table B.1: Data sources (GitHub repositories) and the commits used.

Content	Repository	Commit	Date
DMTF mockups, schema 2026.2	DMTF/Redfish-Publications	4f81814e39	2026-09-16
HPE iLO mockups	HewlettPackard/ilo-redfish-emulator	d668ab1580	2026-09-09
HPE CSM mockups	Cray-HPE/csm-redfish-interface-emulator	8d6a108bbd	2025-12-12
Dell mockups	sailfishdell/sailfish	3bbe0703c2	2019-11-22

B.2 Commands

```
# fetch data (shallow clones are sufficient)
git clone https://github.com/DMTF/Redfish-Publications.git data/pubs
git clone https://github.com/HewlettPackard/ilo-redfish-emulator.git data/ilo-redfish-
    emulator
git clone https://github.com/Cray-HPE/csm-redfish-interface-emulator.git \
    data/csm-redfish-interface-emulator
git clone https://github.com/sailfishdell/sailfish.git data/sailfish

cd code
python3 run_experiments.py ../data ../data/pubs/json-schema ../results
pip install matplotlib numpy
python3 make_figures.py ../results ../figures # also writes ../tables/*.tex
```

The experiment is deterministic. On a single core it runs in under one minute. The file `results/aggregate.json` contains every aggregate number quoted in Chapter 7.