Introduction to Modern Observability

Iiro Kuusijärvi

Haaga-Helia University of Applied Sciences

Bachelor's Thesis

2024

Author(s)
Iiro Kuusijärvi

Degree
Bachelor of Business Administration

Report/Thesis Title
Introduction to Modern Observability

Number of pages and appendix pages
20

Abstract

When we are building software and services, we are building something for others. This necessitates creating and expressing (shared) understanding about what we are trying to. This includes anything and everything from the purpose of the service to our specific work aimed at expressing this understanding – which all work is. For example, when we write a simple Create-Read-Update-Delete request handler logic, this is part of expressing understanding about the service itself. Understanding is often expressed through many abstraction layers. It can be testing, documentation, and the code itself and so on. This paper describes a particular form of abstraction for expressing understanding where we observe the software and its users from the outside in – service acts as a black box with given user inputs and outputs generated by what is called “observability”.

The topic is motivated by the idea that most complete understanding of the software should be on the most curious person. Observability should allow for the most curious persons to dive deep into the service, asking arbitrary questions from it, questions it was not designed to answer explicitly. Software developers desire to understand in technical shapes and colors. For example, asking such questions from a given service and data like “how many items do the 99th percentile users usually have in their cart on average?”, or “how do cart sizes correlate with page load times?” to generate understanding about the most important customers for the service, and from this technical angle, observability is approached in this study.

Before we can start interrogating systems, we have to fit them for this new way of looking at them. This requires new approaches in the context of distributed systems - any software that cannot reason about asynchronicity and timing assumptions turns into a distributed system. These systems are usually modeled as event-based systems, where each interaction creates an event that is sent to downstream services for further processing.

This paper applies this event-based design to logging, and models logging as merely another event in this system - an expression of an atomic unit of work just like any other. We first build a context between monoliths and distributed systems, and then sketch a model of traditional and modern observability fundamentals. To highlight the differences, we proceed to build an example over this theory of a practical issue (logging) through a on-hands comparison of string and custom data structure based logging, in order to show the benefits of aligning the logging type with the event based model.

Key words
Observability, Distributed Systems, Software Engineering, Logging, Microservices

Table of contents

1 Introduction

This paper is about the observability of non-trivially concurrent systems. In the context of this paper such systems range from modern web applications to microservices. The particular technical motivation for the paper is the remote procedure call (RPC) programming model in the center of such services, *but anything* that performs a lot of tasks concurrently, out-of-order, asynchronously, non-deterministically can act as the abstract service at the background of this paper.

First issue with RPCs is that once they are introduced to a program, the application becomes non-trivially concurrent and its execution non-deterministic (Kleppman 2017, 275). This results from the fact that the underlying network in the general case gives no guarantees about giving a response to the RPC in return. Heterogeneous networks do not guarantee any timing assumptions about requests and responses, such as when they return or when it has failed to send them.

Second issue is that when RPCs are introduced to a program, the original program is integrated to the system that the RPC calls and our application turns into a system dependent of another system (which may be out of our control). As a result of this, we are now developing, testing and integrating systems interacting with other systems.

Third issue with RPCs is that they force us to make asynchronicity and timing assumptions (Kleppman 2017, 307) in the presence of heterogeneous network components and network faults, which means we have to handle distributed system fault scenarios – what if the RPC responds too fast, too slow, or never at all? From the client point of view, when a node is unresponsive they cannot tell what is the precise issue with the particular node – they cannot tell the difference between a crashed node and a delayed response. As such the client cannot determine what action they should take without further information. At this point the client is forced to make a timing assumptions by using an arbitrary timeout, adding retries, backoffs and so on to the execution logic. The ckuebt has to treat anything higher than this arbitrary time out as “failed”. (Kleppman 2017, 134-135) analyzes the issues stemming from RPCs and network reliability further. The application layer is generally blind to all of these issues, as it is isolated from the lower network layer and any possible orchestrator such as Kubernetes.

When these issues manifest, they introduce behavior to application that was not designed and not implemented by anyone, called *emergent behavior*. When a distributed system fault is triggered, the resulting behavior is best described as “random”, “bug” or “something doesn’t work” by the users. A trigger for a fault could be an API call never returning, hanging the program execution in place, visible to user as not being able to login. The trigger is often non-deterministic, which means

it does not consistently replicate in testing and debugging – but this also hints at our lack of understanding of the observed behavior (it has escaped our understanding, we cannot *observe* it).

Observability is one way to understand these behaviors, by acquiring visibility into the novel fault condition. This allows to store the event for later replay of the situation in testing environments, under the assumption that it has been setup correctly for predictable and unpredictable faults.

In this distributed world, we are not developing and testing isolated monoliths or applications. Instead we have loosely, weirdly and unexpectedly coupled heterogeneous systems. The combination of states they can get themselves into is limitless, and impossible to preconfigure and expect in their totality – it is unreasonable to expect a metric or a custom log for every fault. Thus, we have to be able to handle novel, unexpected faults in imaginative ways. Handling distributed system faults at the application layer alone is not possible, as this layer is encapsulated from the other services it integrates with and from the layers below it, such as the network layer.

There are strategies that help mitigate this uncertainty, for example using fakes, mocks to drive application state during testing, using containers to mimic production integrations, but this does not bring back the previous guarantees. We have instead created a new environment with certain probabilistic fidelity towards the production environment with these measures. Probabilistic because we only have a certain probabilistic assurance that our service works as intended in production, due to our test and development environments *not fully replicating production context*. Each such environment becomes its own distributed system and should be understood as such.

Furthermore, the compiler and tests cant reason about the timing assumptions of RPCs. Neither can they, nor the application programmer, reason about the resulting asynchronicity models between the servers, services and different systems (these are *obscured* from the application layer and they are far too large to fit into the context window of a single programmer). ‘

In general, we do not have any guarantees about some network call returning within certain time, or if that request was even sent. As such, we are no longer guaranteed a working application after it has compiled and all the tests have passed; we only have a certain probabilistic model of it, giving us assurance about how it will behave in production context – observability into the production context becomes critical in order to understand our services.

1.1 Technical Context

Concurrency allows a *node* to execute many tasks at the same time. In a given time block, multiple tasks can be executed by a node. A node can be anything that can execute tasks, a server, a

thread, a service and so on. In a distributed systems context, concurrency is generally modeled as message-passing concurrency. In message-passing concurrency, different nodes of a system communicate by sending messages (requests and responses) using a *shared-nothing architecture* (Kleppman 2017, 278). Shared-nothing architecture refers to the fact that these nodes can only communicate through the messages sent in the network. Each completed task gradually builds up to some event that is sent forward to another node.

Non-trivial concurrency is a qualification to model the elastic, high availability cloud infrastructure the applications are deployed to. Unspecified amount of concurrent nodes executing at the same time makes the execution non-deterministic and impossible to debug by traditional means, such as using a debugger; using a debugger with complex microservices is not feasible due to the sheer amount of different software we would have to “step” through.

Distributed systems are composed of smaller services communicating and coordinating towards some common goal, this is where their other name, microservices, comes from. Depending of the abstraction level, a node in this distributed system could for example describe the whole `CheckoutService` (if we desire to understand things at a service level), or an individual server in the `CheckoutService` service (if we desire to understand things at a more granular level, depending of our needs – we might have to debug a server). Service-level abstraction is a natural abstraction level for the service operator because of the interface it provides for atomic, transactional units of work. We refer to these atomic units as *events.* A service has committed its part of the work when there is a call to another service and an event is fired off. This also enables the service to tie observability directly to service-level objectives (Majors et al., 2022, 138).

Past decade has been the era of cloud, where services can scale to multiple interdependent services with tens to thousands of nodes, and then in an instant scale down (this is the elastic property of cloud services). Practices from non-distributed services and monoliths do not transfer very well over to distributed communication (by teams and services) nor to the underlying cloud infrastructure. For example, during all of this the service as a whole, or one of its parts, may gain *emergent behavior* at a runtime – behavior that only manifests when the whole contraption is running in production context; behavior nobody designed, implemented, tested nor intended for the system to have; and most importantly, behavior nobody thought to create specific logs nor specific metrics for ahead of time, ahead of the fault – impossible by definition. Such behavior is mostly reported by end-users, if you are lucky your existing observability can indirectly indicate something is wrong, in the worst case they manifest as metastable failures which is a stable equilibrium where the system appears to be running as usual but is actually in some kind of a failure mode (Bronson et al, 2021).

1.2 Research Questions

The paper covers a software engineering and design problem, not a research or computer science problem. The main question is *how to model real distributed systems running in production given some observability tooling and instrumentation of application code*. By “modeling” we refer to *creating and improving the understanding of the system’s behavior.* For the purpose of this paper, a distributed system can be as small as one API call, or a more complex contraption in some cloud service going through 15 different services; it matters none since with only 1 network call to another system we can already explore the problem and solution space.

Observability is chosen as the tool to explore the questions, because it offers fair trade-offs compared to the two other alternatives, testing in production and verification. The main idea, which is not simplified, is that since modern distributed systems are event-based, so should their observability be.

The main research problem is *how to implement observability for unpredictable faults*. This question involves the constraints of:

1. No shipping of new code to fix an issue.
2. The team operating the service (a) may not access the faulty service it depends on (b) may not know which service is faulty, and (c) may not instrument the faulty service.

First constraint is a general constraint to make the problem interesting. It forces us to think about *how to create a system from which we can ask arbitrary questions,* rather than particular questions. Particular questions are established for (debugging) predictable faults, such as metrics for the average latency of the 99th percentile users – which already answers the question “*how to implement observability for predictable faults*”. It is also introduced as a constraint since if we knew what code to ship in order to fix the fault, we could have had monitoring set up for the fault ahead of time, and as such, these are not novel faults (Majors et al. 2022, 4). These faults are engineering oversights. Second set of restrictions model how most organizations have organized software development teams in a microservice setting, for example *most teams do not have arbitrary access to services they depend on,* so that is why we introduce the restriction *2a;* often such visibility is limited and intentionally restricted. This hints at an influential idea in system design, that software (organization) replicates the communication structure of the organization (software), known as the Conway’s law (Gilson, 2021).

Potential solution space with these constraints can be a mixture of modeling the system via formal verification, testing in production, and observability. Verification is hard to scale, but parts of certain

systems can be verified (Newcombe et al, 2015). Testing in production assumes an already existing sophisticated observability of services; and requires operational sophistication from everyone involved (Sridharan, 2022). Testing in production may not be feasible due to legacy systems or regulations limiting production access too much. It may prove to be too demanding to apply in practice. On the other hand, observability already in some form or another exists in most modern distributed systems, and improving on it can be gradual process, and there can be a different approaches such as network-level implementation depending on the requirements, leaving application-level teams working in peace. Having sophisticated observability also helps setting up testing in production.

1.3 Scope And Limitations

The scope and limitations of the research questions are defined by the context of the paper, which is provided to peers and under peer-review. First, the theoretical exposition is limited to basics of observability, and the empirical part is limited to something immediately practical, like a difference between logging styles where we compare a string-based, traditional logging, to a logging with the help of a data structure. Strings and custom data structures are both taught at introduction classes to programming. The constraints for the questions in section 1.2 model the real world distributed system organizations, services and development.

1.4 Research Objectives

A service operator wants to know when a unit of work behaves *kind of* differently from the median or average. This deviation can be marked by anything, it may execute slower than the rest, it could be associated with a fault, or it has failed and so on. Deviations are the most interesting things, and what we want to drill down to when we are being curious. Modeling logs as events helps to achieve this kind of observability, because they provide a machine readable data structure we can *query* out of the box. This helps creating better understanding about the deviations, and service itself. This paper argues for a shift from viewing observability as only the three pillars (logs, metrics and traces) to viewing observability as a way to model systems via events, where the three pillars are merely the way we build this understanding. Observability in a concurrent context is not a log line, or a metric, but a structured event.

The paper first goes over the theoretical foundations of observability, handling the common concepts from observability such as logs, traces and metrics, with some new ones like structured log

and contrasting them. This is then applied in the empirical section to show the practical difference between logging with strings versus logging with a data structure; the solution is simple and practical.

2 Building Blocks of Observability

Observability is classically understood as the “three pillars of observability” (Sridharan 2018). The three pillars are *logs, metrics and traces.* Combined, they are supposed to let us observe what the system is doing from the outside in. However, this is left wanting – just like we use functions in a programming language to express an idea, the three pillars are the initial building blocks for actually expressing understanding. That is, we have not created understanding nor observability if we simply log something, collect a metric there, and trace something arbitrary here. We will embrace a fuller definition of observability, as a “*measure of how well you can understand and explain any state your system can get into, no matter how novel*” (Majors et al., 2022).

In the context of this paper, we are interested in how to implement observability for our application and services – this a software engineering and service design problem. We are not interested in how to implement logging, metrics or tracing themselves, but rather how to use them to express understanding about what our service is doing, or not doing, from the outside in.

Even the simplest modern web application, with a frontend, a backend and a database layer, is non-trivial to test statically or dynamically. For example, in a three-tier architecture with a frontend, backend and a database layer, the technology stack is most likely to be *heterogeneous*, meaning that every part of the architecture could be written in using technologies with different assumptions, such as languages with different runtimes, semantics and process models which do not extend by any law of associativity to other layers or technologies of the architecture. There are few approaches to handle these issues, and observability is what this paper focuses on: we observe them, record them, study them.

For the purposes of this paper the relevant ties to (theoretical) computer science will be the understanding that we do not log in a hot loop, and that I/O operations are costly. As such, the theoretical context is built with caveats: our foundations are in the engineering and design practices of the past, inherited from monoliths and in the modern bleeding edge practices, discovered after moving to distributed systems. That is, the solutions engineers have found to observability and modeling issues with regards to distributed systems were not derived from first principles. They were engineered, often by people looking for any solution that’d be good enough.

2.1 Logging

Logs, traditionally understood, are lines in the application code that *log* something, such as the message “*Listening to port 8080*”, or what resource the user asked for, and so on. Logging allows

observability into the internals of the component and the logic that is being executed. Logs can't state what happened to the event across its whole lifecycle through the services and components. To be able *to narrate* the lifecycle of the request, we need to associate a unique ID for the event and some kind of tracing to tie the unique IDs and different service hops together, to represent the whole lifecycle of the event. From this, we can then *build a narrative* for a service operator about this particular event.

Logs have several costs associated with them. There is a performance cost because every log line implies an I/O operation. This could be synchronous IO (IO done immediately) or batched (something behind the scenes collects these events for writing them out). There is an economic cost of sending, storing and processing the logs. Together they generate a third, mental, cost for the programmer using the classical logging approach – he or she has to be aware that these operations have a non-trivial cost, and as such the programmer becomes discouraged from logging everything it took to execute some unit of work.

Logging has the nature of producing a lot of data, which with some refinements enables us to interrogate the event with arbitrary questions. This is a crucial fact that will enable us to build an observable system as defined earlier. However, traditional logging is neither structured nor data and this poses several issues. Traditional logs are *lines of strings* that some observability tooling needs to parse, or worse, the programmer manually using regular expressions. Furthermore, with traditional logging, the log lines are spread all over the component as a leftover tradition from the monolith "print debug" approach. As such, any information you might need is spread at some probability over any of the log lines you've written in some string form that needs to be parsed. This implicitly describes another problem of traditional logging in the distributed world: with traditional logging you kind of have an understanding that "*These 10 log lines spread all over this request handler speak about the same, single event, the incoming request*", but they do not form a single object describing this single event and what it took to execute it – they are only 10 different strings, printed to `stdout` as the handler logic is executed.

The issue, and the "observability smell", in distributed system context are these non-structured individual log lines, or, the traditional logging approach as a whole. We desire data and structure, a single structured log (consider it a class, object or type that we fill with arbitrary key-value fields) printed once to `stdout`, that describes a logical, atomic, transactional unit of work.

2.2 Canonical Log Line

An intermediate form towards structured logging was popularized by software developers at Stripe. The implementation is called *canonical log lines* (Brandur, 2019). It is a product of engineering

practices at Stripe. The name comes from the fact that it is a one log line, printed as the event was about to exit or error from the component, containing structured data in a key-value pairs. It is called canonical "*because it's the authoritative line for a particular request, in the same vein that the IETF's canonical link relation specifies an authoritative URL.*" (Brandur, 2019). This single log line holds everything which was required for the unit of work to execute. By everything, we literally do mean everything we can extract from the context, environment, metadata and the request at that point of time, at that stage of its lifecycle through the service. This can mean hundreds to thousands of key-value fields.

This approach provides a solution to two issues with traditional logging. First, it provides a service operator with structured logs and allows the leveraging of existing observability tooling that can immediately access the data structure's keys to get the values – we have changed parsing strings into indexing into structured data. Second, there is now a single canonical log line instead of many all over the request handler, hence there is no need to search for a particular log line anymore, not across the logs nor across the application code – it is contained in the canonical log line. A third benefit is that this also makes adding new data to the logs very cheap: writes stay at constant one because we can add more key-value pairs to the log structure. *This is cheap*. This approach of one line, per request, per service would become known as "observability event", the core of a modern distributed system logging approach. Implicitly at this stage, the canonical log line also represents the unit of work, and is for that reason called canonical, containing all the information about the unit of work.

2.3 Structured Logging

Structured logging, or an *arbitrarily wide structured event* (Majors, 2022), iterates from the approach of canonical log lines by formalizing the logging around *observability events.* (Majors, 2022) defines observability event as "*a hop in the lifecycle of an end-to-end request*". This means that observability events are defined as atomic units of work at the service level. Still, the observability event is "*one line per request per service"* (Brandur, 2019), we do not spray the code with logs or events. This unit of work is an event, but not all events represent an atomic, transactional unit of work, as they can just be parts of this unit of work being executed.

For example, as a service operator, when I host an API for clients I am interested in when, what, where and why their work behaved the way it did. In a web service we may have a request handler for a registration endpoint. The unit of work is "Registering A User", which may itself generate many events internally: "Validating User", "Sending Email", and so on, which compose this unit of work. As a service operator, I desire to know structurally that "Registering User" workflow for a

client failed at the “Validating User” stage, an event within the actual unit of work we’re interested in. We desire to *index* into this workflow, and structured data allows this.

For a web service, the observability event is generally one log line per request per service hop as proposed in the canonical log lines approach. That is, if we have a request coming into our handler, we generate one observability event from this. Here, the observability event is “Registering User”. The structured log should contain everything that was required for this unit of work to execute, indiscriminately. This provides us an arbitrarily wide structured event (Majors et al, 52). Think of hundreds of key-value pairs that we can query, which we can feed to our systems for more insights. This logging approach is cheap with one write, adding a field may cost a few bits, but that is far overshadowed by the benefits. We should gather observability events from every network hop, process fork and similar change during the lifetime of the unit of work, when it is being executed. The single structured arbitrarily wide event should contain hundreds of fields in, containing everything it took to execute this particular unit of work while we keep writes at constant one - adding fields is essentially free in comparison to the IO operation.

A request that hits *your* frontend, *your* backend, and *your* database generates three observability events. A request that hits *your* frontend, *your* backend, and a *third party* service such as external API generates two observability events for us. This is because we cant by definition instrument an external service. The last observability event we can record is when the request is about exit or error from *our* backend. As such, these events come with the limitation that we can only collect them from services we can instrument as we need instrumentation at the log and trace level. As such, observability from, for example, vendored services (databases, other blackbox software) is hard to engineer.

The major difference to traditional logging is that structure, be it JSON, key-value pairs and so on, makes the log machine readable.

2.4 Metrics

Metrics are a high level abstraction of how a system is behaving within a certain time period, compressed into a number. That is, metrics gives binary statements, “good” or “bad”, ”yes” or ”no” about some metric over some time block - this is insufficient for drilling down, correlating complex states software can get into. Unlike logs, metrics do not provide observability into low level context (ie. service component internals). Instead, metrics are used for aggregate modeling of a system. For example, you might have a dashboard with several aggregate metrics, usually showing a trend over time. A single metric indicates *something* binary about *something* over *some* time period,

usually represented as a single number (such as average latency). A collection of such metrics shows a trend of system performance over time. The issue with metrics-based observability, or traditional monitoring of system conditions in general, is that the faulty conditions need to be defined ahead of time which is not reasonable expectation with modern distributed systems, with uncountable, arbitrary states they can get into. As such, by definition, monitoring only provides information about known faults, though the author admits that the dashboard operator might develop a sixth sense similar to "*if this metric fails that way, while that metric does not, it refers to this fault scenario X, which does not have its own metric*", but this is not engineering nor observability, rather, this is superstition used for debugging complex systems.

Another issue is the (already) prohibitive costs models for metrics, which only get worse for event based observability, for correlating and finding causal patterns in arbitrary dimensions across arbitrary cardinality, as we would have to collect metrics on every possible dimension – we would hit budget restrictions very fast.

2.5 Tracing

A trace, or a span, is a collection of "*causally related distributed events that encode the end-to-end request flow through a distributed system"* (Sridharan, 2018). Tracing enables this by using an unique identifier (ID) to associate a particular event with all the other events it triggers as it travels through the services, from the start of the request lifecycle to its end.

When a request is associated with such ID, we can associate the current context (any metadata about the execution environment) of each service with this particular ID as they executed their logic when the event travelled through them, and observe the event lifecycle and its effects on service, and services effects on the event itself. Then a tracing library and optional tooling combine these events associated with this unique ID to present what is called a trace, a span of the event. It contains all the observability events, forming a narrative for the service operator to read. An additional benefit of tracing is that it will uncover dependencies and correlations between your service components and your (distributed system) faults that you might not have known about now that you can correlate the requests with the unique ID.

Tracing is hard to implement afterwards, because you need to edit every component to propagate tracing information and to adjust for different technologies along the way - libraries, frameworks, applications all might differ, which means your implementation of tracing has to take the different technologies into account, as they all might propagate the request ID a little differently. Usually, as a hack around this prohitive engineering cost, microservices use a *side-car proxy,* something like

Envoy, to implement this at a network-level, taking substantially less time than application changes.

3 Implementing Logging

As a practical implementation of one observability related distributed programming pattern, we will inspect logging more closely by comparison of traditional, string-based logging and data structure based logging at the application level. We will highlight some of the problems with legacy logging practices, such as logging implicitly meaningful statements here and there and what this critique means in practice; having to combine them to represent a meaningful event *after the event has happened,* and other restrictions the approach brings about, hampering the ways developers can surface meaningful understanding from the service. These are compared against the benefits of structured logging: logging explicitly meaningful *events* as they occur, and logging them once.

3.1 Traditional Logging

An exemplary of traditional logging, and its issues, is console logging in a web application. Most people have debugged their full stack applications by sprinkling console log statements around the execution path like magic fairy dust. This is a fairly tried and tested method of debugging web applications, and this signals how hard observability can be when this is the preferred method by most developers. An example of this approach could be the following info log statement usage in an application:

```
fn req_handler(req: Request) -> Response {
  // 1. Log the incoming request
  info("Received { req }")
  // 2. Process the request
  let data = process_request(req)
  info("Finished processing: { data }")
  // 3. Log the response we return:
  let res = Response { body: data }
  info("Sending out: { res }")
  return res
}
```

Fig 1. *Example of traditional logging approach.*

There are three info log statements, which imply three writes. Depending of the logging framework and programming language, second one results into something like *`2024-20-02 INFO ”Finished Processing: <some_data>”`*, which contains a timestamp, logging level and the actual message in it**.** It is then written into some output of choice when the program execution meets the log statements. If this was running in a browser, then they would appear in the console. It could also be stored, and batched so they’re written all at once (but still requiring a write for each statement).

The issue in the general case is that these log statements are not *structured* via data structure. There is only an implicit idea of structure in the way these statements are currently used to describe the request that came in and what happened to it. Similarly, there is only an implicit idea of meaning, which is lost in the execution context. That is, because these are *strings*, and not a single *data structure*, someone has to combine them from the handler and logs, manually, so that they can describe the request, and the event that happened. This also means that if we want to understand any single info statement in this handler, like ”*what finished even processing in the first place?”*, we have to read them all, *if* we can even find them all from the logs.

Because the statements are represented as strings, and not as structured data like a JSON object, for example ”{ processed: True }”, they are not machine readable. In order to start *querying* the info log statements from the logs, *to ask things about* them such as ”*what response did we send out on 2024-20-02?*”, something or someone has to parse a prefix of these statements first, acting as the key, so that we can then parse the value it maps to out of it. This means programmatically searching for mentions of “Finished processing” in the logs for example, wasting developer hours.

The difference to structured logging is that when using some kind of a structure, such as JSON, which is machine readable, we would access a field `data` to get whatever data we processed from it, which could be an AWS S3 bucket link, or just a boolean flag. Once this key value pair is gathered as part of some event like `UserRegistration` we can also encode the explicit meaning of the data as part of the structure of the log, so we do not need to scroll the logs to compose the meaning of the log line after it happened: we have the whole event collected in a single canonical, structured log, telling a simple narrative to us.

3.2 Structured Logging

A structured log is defined by its data structure. This data structure can be arbitrary, as long as it is machine readable list of key-value pairs. This is achieved by making a type, class or an object (depending of your language) to represent the structured log. We can call it the `Context` object:

```
pub struct Context {
  pub env: String,
  pub api_version: String,
  pub processed: bool,
  pub request: Request,
  pub response: Response
}
```

Fig 2. An example of a (simple) Context object, with five key-value pairs.

Fig 2. represents a simple Context object. The fields (env, api_version, processed, request, response) act as a key for their respective values which we can fill in the middlewares of the service or in the request handler. This already removes any reason to jump around the request handler, scanning for individual log statements since we can see them collected in this object, and immediately scan with eyes to whatever associated data we want to know.

```
fn req_handler(req: Request, ctx: Context) → Response {
    // 1. Add everything you can think of to Context object.
    ctx.env = telemetry::get_env()
    ctx.api_version = telemetry::get_api_version()
    ctx.request = req
    // 2. Process the request
    let data = process_request(req, ctx)
    ctx.processed = true
    // 3. Log, and return, the response
    let res = Response { body: data }
    ctx.response = res
    return res
}
```

Fig 3. Example of how to use Contex object in a request handler

Fig 3. shows how to use a Context object approach in practice: we fill this data structure with data we want to log as we execute the handler logic. The keys are assigned data, and unlike Fig 2 which contained three writes for three log statements, this handler logic contains zero writes. The middleware collects both the `Request` and `Context` objects, and decides what to do with them after the handler logic has executed. Usually the response is forwarded if there were no issues, and the context object is also forwarded, maybe to a queue, so the total combination will be constant two writes, compared to minimum of four in figure 2 – and writes in it will continue to grow when you want to add more log statements, writes in figure 3 will remain constant two, one for the log, one for the response.

Fig 3. also informs of something else related to logging practices: ”INFO”, ”WARN” and other logging levels appear to be logging practice smell from the monolith era, as they are absent now. They had to be carefully adjusted when writes and disk space was costly, networks were slow, you cannot have tens of debug statements slowing down programs in production, potentially causing concurrency issues with the writes, but modern distributed patterns have different assumptions and practices as we can see, and it feels natural.

This `Context` object is serialized to some structured output data structure before sending it out to some sink (console, standard output, or some queue). JSON is a fairly common data structure to use for log data, and large ecosystem of observability tooling supports it out of the box.

Any structure in this context has three important properties for the purpose of bringing about an observable system: dimensionality, cardinality and the single write principle. *Dimensionality* of the data structure is defined by ”*the number of keys within that data*” (Majors et al., 14). The more fields it has, the higher its dimensionality. The goal is to have hundreds, if not thousands, of these fields – the goal is to be able to *correlate different objects*, such as Kubernetes clusters to Docker versions to AWS regions and so on. *Cardinality* of the data refers to ”*to the uniqueness of data values contained in a set*” (Majors et al., 13). As a general rule, observability will benefit from high cardinality data, providing higher resolution snapshot of the state of the system at a point in time. Since we are interested in *deviations* in the system, it is the high cardinality data, that is, data with a wide range of unique values, that allows us to isolate that particular behavior, request, user. Together high dimensionality of the structure and high cardinality of its data allow the correlation of any arbitrary data in order to investigate novel, unpredictable, unforeseen states our services have gotten into.

Third important property of the structured logging approach is the single write principle. This will make sure the performance wont suffer, excluding the cost of the single write. By using a data structure rather than traditional logging approach, we move from logging by line-by-line to logging by *appending a new field to the data structure* representing our log for the event. This enables logging to keep writes at constant one with the trivial cost of bit or two for the new field itself. This benefit is compounded by the fact that in a distributed system we do not want to execute more writes than necessary to other systems over the network, they are costly and prone to error. This also makes it cheap to increase the dimensionality and cardinality of data, as the cost of appending a new field to the structure is trivial - much less than the cost of a new write. This lets developers log everything that was required for the unit of work to execute in a given context; developers no longer have to think ”*should I log this?*” and thus pre-emptively restrict observability into program internals, instead they can now focus on expanding it: ”*what else should I log?*” and the answer is ”*everything*”.

A structured log is usually collected by some kind of a logging / tracing framework. There are vendor specific options (ie. Software Development Kit provided by the tooling vendor), or we can use vendor agnostic OpenTelemetry libraries, which ”consists of vendor-neutral open source tools, APIs, and SDKs” (Datadog, 2024). Most logging and tracing frameworks support exporting log data as JSON. The structure, be it JSON or something else, allows these logging frameworks to consume the data and, because the structure makes it already machine readable, to display us dashboards, tracing, visualizations out of the box by indexing into the keys of the data structure. There is no requirement to parse these by the service operator before getting visible statistics, dashboards and metrics, they can even provide us correlations!

4 Alternative Approach: Service Mesh Implementation

As mentioned in the tracing section, *refactoring observability into an existing system afterwards is a non-trivial operation*. Refactoring observability as suggested in this paper so far requires touching application code for instrumentation by the team, even if it is only the application code that team owns. But to apply this to a large company would imply that each team has to worry about implementing it, which is a scaling issue – it requires developer hours and would require each team with application code to do it. Alternative strategy, which can compliment or perhaps even replace the approach suggested in this paper is a network level implementation of observability. This service mesh approach is the standard today for microservice deployments, usually most Kubernetes deployments assume you are using it. This requires sophisticated understanding of infrastructure and operations. It may provide helpful refactoring of platform development and network engineering within the company.

As an industry case, Lyft implemented observability using a service mesh pattern without touching application code. Their journey started in 2018, when they ”completed decomposing [their] original PHP monolith into a collection of Python and Go microservices” (Grossman, 2022). The way this approach works is that in a service mesh, a *side-car proxy* sits in front of your application, and your application interacts with rest of the infrastructure, services and world through it via network requests; for example at Lyft ”*every instance of a service is deployed alongside a sidecar Envoy which acts as the sole ingress/egress for that service.”* (Grossman, 2022). The side-car proxy then intercepts the incoming / outgoing requests and responses, instruments them with logging, tracing, inspects them, pushes the structured logs to queue for later consumption by some observability tool.

With this more sophisticated approach, the infrastructure team implements the observability *much* faster at a scale by deploying it to the side-car proxies once the observability approach is designed and implemented; so rather than each application team spending time implementing and coordinating the approach, a dedicated network / platform engineering team spends their time on it, a one-write approach to implementing it at a scale. Caveats are that with anything involving Kubernetes, service mesh, there is an increase in complexity – this is a known success story for a reason.

5 Discussion

The reliability of the research is questionable. Since the paper addresses a high-level software engineering and design problem, which is the *implementation of application and service level observability*, it involves a lot of opinions and qualifiers. This is partly justified on the grounds that the observer and the service operator is still a human-in-the-loop at some dashboard or another, to whom we have to impart meaning about our service. As such, matters of taste and subjective experience leak into the discussion – for example one argument for the structure-oriented approach was that the service-operator doesn't have to scroll, with their own eyes, the handler logic nor logs for individual log statements, which is a fallible operation, the operator could miss something; instead they have it all neatly wrapped up in a single object now (less fallible operation to check the keys of the data structure).

An implicit qualifier of the paper was the unstated assumption that we have some ambiguous microservice, service-oriented or event-driven architecture with old deprecated logging practices (which evolved in the context of monolith applications) which we are critiquing at the background of the text constantly. This assumption is partially justified on historical grounds - the move to cloud has not been an entirely greenfield operation where everything was made a new from ashes. This usually involves doing good enough work at packaging existing software into container(s) and shipping them. It is also justified on technical grounds - most services properly understood are distributed systems as they cannot reason about neither time nor asynchronicity, and they do not model their debug statements or logs as events from which they could benefit.

Second implicit qualifier was that monolith practices do not translate well to distributed systems and programming and that most observability practices out in the wild are still following practices inherited from monoliths, such as using strings, grepping, multiple writes, restricting logging due to costs (costs, which come from monolith practices). Most of this was left unexplored in paper, but there are many such monolith practices and assumptions that do not work in highly concurrent, networked context, such as modeling of the system: there was hope that a monolith context could still fit in the head of a programmer, or a single team, but there is no such hope in microservices context.

Further applied engineering research has been done on the the idea of observability events, for example Akita's network-level observability tooling. It aims to be drop-in solution for (some) observability needs, using advanced traffic analysis and eBPF (Akita Software, 2024). They were recently acquired by API tooling company Postman. Akita tracks your network (requests) and forms a model of it, APIs it has, based on events that occur in the network.

As far as the author is concerned, if the main problem of approaching things without a data structure became clear, and the benefits of data-structure oriented approach were also clear, the goal of the paper has been reached. As for personal learning process, it has helped me to understand some of the more subtle parts of the implementations; the approach proposed requires sophisticated error handling to make sure the `Context` objects are written out in any kind of fault scenario.

Bibliography

Sridharan, C. (2018) Distributed Systems Observability: A Guide to Building Robust Systems. O'Reilly Media. Available at: https://learning.oreilly.com/library/view/distributed-systems-observability/9781492033431/. Accessed: 11 February 2024.

Sridharan, C. (2022) 'Testing in Production, the safe way', Medium, 5 January. Available at: https://copyconstruct.medium.com/testing-in-production-the-safe-way-18ca102d0ef1 (Accessed: 17 February 2024).

Kleppmann, M. (2017) Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media.

Majors, C. et al (2022) Observability Engineering: Achieving Production Excellence. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly.

Majors, C. (2022) *Live Your Best Life With Structured Events*, *charity.wtf*. Available at: https://charity.wtf/2022/08/15/live-your-best-life-with-structured-events/ (Accessed: 19 May 2024).

Newcombe, C. et al. (2015) 'How Amazon Web Services Uses Formal Methods', Communications of the ACM, 58(4), pp. 66–73. Available at: https://doi.org/10.1145/2699417.

Bronson, N. et al. (2021) 'Metastable Failures in Distributed Systems', in Proceedings of the Workshop on Hot Topics in Operating Systems. HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor Michigan: ACM, pp. 221–227. Available at: https://doi.org/10.1145/3458336.3465286.

Leach, B. (2019) Fast and flexible observability with canonical log lines. Available at: https://stripe.com/blog/canonical-log-lines (Accessed: 3 March 2024).

Gilson, N. (2021) What is Conway's Law?, Work Life by Atlassian. Available at: https://www.atlassian.com/blog/teamwork/what-is-conways-law-acmi (Accessed: 10 March 2024).

Grossman, M. (2022) Scaling productivity on microservices at Lyft (Part 3), Medium. Available at: https://eng.lyft.com/scaling-productivity-on-microservices-at-lyft-part-3-extending-our-envoy-mesh-with-staging-fdaafafca82f (Accessed: 10 March 2024).

Akita Software. (2024) *How Akita Works, Akita Software*. Available at: https://docs.akita.software/docs/welcome-1 (Accessed: 10 March 2024).

Datadog. (2024) *What is OpenTelemetry? How it Works & Use Cases*, *Datadog*. Available at: https://www.datadoghq.com/knowledge-center/opentelemetry/ (Accessed: 10 March 2024).