ANAN — A DEBUGGER FOR COMPUTE CLUSTERS

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich 12 (Informatik und Mathematik)
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von
Alexander Adler
aus Frankfurt am Main

Frankfurt, 2021
(D 30)

Contents

Part I

Introduction

Chapter 1

Danksagung

Der erste Dank geht an meine Eltern, die mich in diese (für mich zumindest anfänglich) trockene und kalte Welt gebracht haben und sich noch eine ganze Weile weiter mit mir befassten.

Danke an meine Frau: Sie zeigt immer wieder, dass man auch einem Hasen die Ohren lang ziehen kann.

Danke an meine Tochter: Sie lehrt durch Unkenntnis des Worts „Laune", dass man keine schlechte Laune zu haben braucht — weiß es andererseits durch die Ausbildung von „Ansichten" zu kompensieren.

Danke an meine beiden Betreuer Profes. Dres. Udo Kebschull und Volker Lindenstruth, die immer schon wussten, was als Nächstes kommt, aber trotzdem mit bestem Rat unterstützten.

Danke an Dr. Walther Scholl, der eine Klage stets zu schätzen weiß und in früheren Fassungen dieser Ausarbeitung genug Anlass zur Klage gefunden hat — hoffentlich sind in der vorliegenden Fassung die Anlässe weggefallen. Natürlich sind die verbleibenden Fehler nur dem Autor anzulasten!

Danke an Lenny Sandberg, der vermutlich der ideale Gutachter wäre — zumindest brauchte er für die Lektüre nur ein paar Stunden. Aber auf das Gutachten wartet die Welt bis heute …

Chapter 2

Zusammenfassung

וַיהוָה הֹלֵךְ לִפְנֵיהֶם יוֹמָם בְּעַמּוּד עָנָן לַנְחֹתָם הַדָּרֶךְ
וְלַיְלָה בְּעַמּוּד אֵשׁ לְהָאִיר לָהֶם לָלֶכֶת יוֹמָם וָלָיְלָה׃

The LORD went before them in a pillar of cloud by day, to guide them along the way, and in a pillar of fire by night, to give them light, that they might travel day and night.

Exodus 13,21

Das Projekt anan ist ein Werkzeug zur Fehlersuche in verteilten Hochleistungsrechnern. Die Neuheit des Beitrags besteht darin, dass die bekannten Methoden, die bereits erfolgreich zum Debuggen von Soft- und Hardware eingesetzt werden, auf Hochleistungs-Rechnen übertragen worden sind. Im Rahmen der vorliegenden Arbeit wurde ein Werkzeug namens anan implementiert, das bei der Fehlersuche hilft. Außerdem kann es als dynamischeres Monitoring eingesetzt werden. Beide Einsatzzwecke sind getestet worden.

Übersicht über den Aufbau von anan. Das Werkzeug besteht notwendigerweise aus zwei Teilen:

1. aus einem Teil, der *interaktiv* vom Nutzer bedient wird — dieser Teil ist nur einmal vorhanden;
2. und aus einem Teil, der *automatisiert* die verlangten Messwerte erhebt und nötigenfalls Befehle ausführt — dieser Teil ist pro untersuchtem System je einmal vorhanden.

Der erste Teil wurde anan* genannt, der zweite anand. (Somit besteht

*Ursprünglich stand die Abkürzung für **an**alyse **a**nd **n**avigate. Diese doppelte Zielsetzung erwies sich als zu eng. Glücklicherweise stellte sich heraus, dass das im einleitende Zitat benutzte Wort für "cloud" bzw. Wolke im hebräischen Original *ʿanan* ausgesprochen wird, womit der Name beibehalten werden konnte.

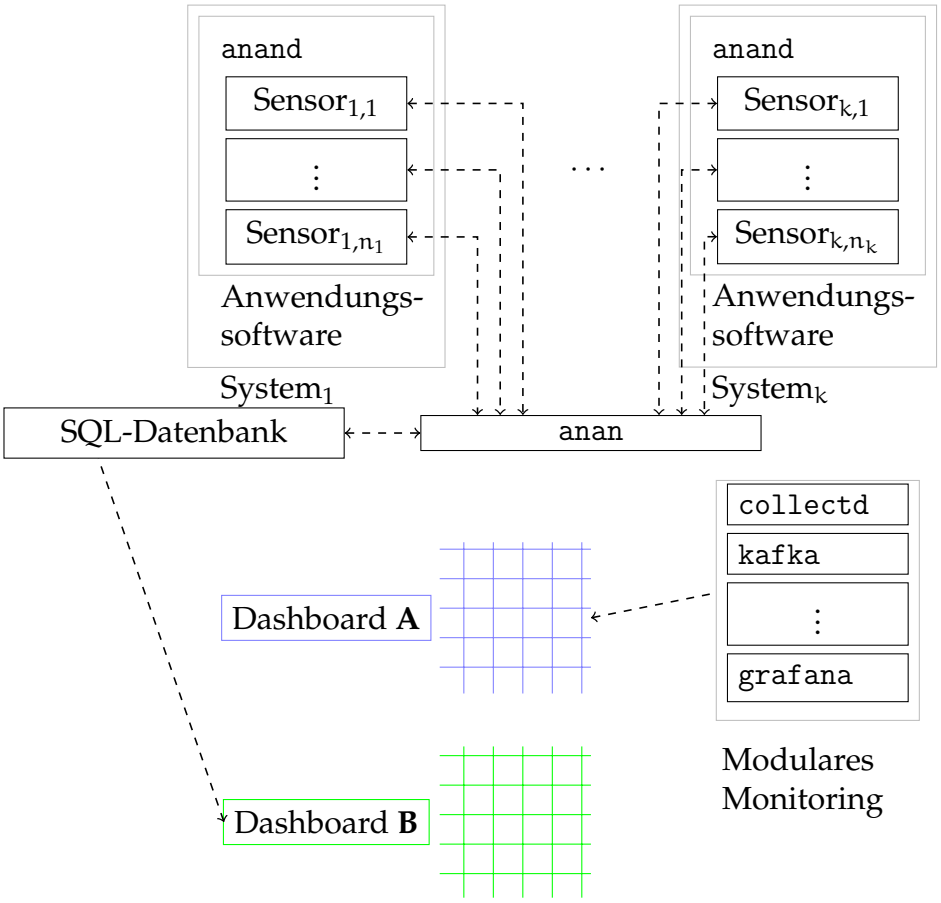

Abbildung 2.1: Übersicht von anan bei gleichzeitigem Einsatz für Monitoring und Debugging

das *Projekt* anan aus den beiden *Programmen* anan und anand.) Diese Zweiteilung ist notwendig: Zum einen müssen an jedem untersuchten System Messungen durchgeführt werden, die nur innerhalb des Systems möglich sind. Zum anderen müssen die Messungen zusammengeführt werden, um eine abstraktere Sicht als die der einzelnen Werte zu erlauben.

Weil anan ohnehin schon eine große Menge an Daten über den Zustand der Systeme erheben kann und für interaktiven Betrieb hinreichend schnell sein muss, liegt die Idee nah, anan für Monitoring einzusetzen. In Abbildung 2.1 ist der vollständige Aufbau bei einer Nutzung für gleichzeitiges Monitoring und Debuggen dargestellt. Dabei wird anand auf jedem zu untersuchenden System ausgeführt. (Ob dieses System ein Computer, eine virtuelle Maschine, ein Container oder etwas anderes ist, ist aus anans Sicht egal, sodass weiterhin unspezifisch von „Systemen" die Rede

sein kann. Natürlich unterscheiden sich die messbaren Variablen in den jeweiligen Fällen.) Jede Instanz von `anand` führt null oder mehr *Sensoren* aus, wobei nicht alle Systeme dieselbe Auswahl an Sensoren ausführen müssen. Ein Sensor ist ein in der Regel kurzer Algorithmus, der etwa einmal pro Sekunde ausgeführt wird, und ein Messergebnis erzeugt. Die Messergebnisse werden im Sekundentakt an `anan` übermittelt, wo sie für das Monitoring in eine SQL-Datenbank geschrieben werden.

Eine Schlüssel-Wert-Tabelle ohne weitere Struktur genügt dafür nicht, weil die übermittelten Daten zwar nur Zeitreihen aus sonst nicht weiter strukturierten Daten sind, aber sowohl für den Debugger als auch mitunter für das Monitoring korreliert oder in kompliziertere Beziehungen gebracht werden müssen. Das wäre mit einem einfacheren Datenmodell als dem von SQL vielleicht möglich, aber langsamer und mühseliger. Dadurch ginge der Hauptvorteil der einfacheren Datenstruktur, nämlich ihre Schnelligkeit, verloren.

Aus der Sicht von `anan` ist alle Software in den untersuchten Systemen — mit der Ausnahme des Betriebssystems und `anand` selbst — „Anwendungssoftware". Dazu gehört auch weitere Monitoring-Software, weil Monitoring beim Entwurf von `anan` höchstens ein nützlicher Nebengedanke ist. Es besteht daher nicht die Absicht, dass `anan` das alleinige Monitoring-Werkzeug sein soll. Daher wird dem Nutzer des Monitorings eine Reihe von *Dashboards* präsentiert, ohne dass er zu wissen braucht, ob sie (wie Dashboard A) ihre Daten aus einem modularen Monitoring-System oder (wie Dashboard B) aus einem monolithischen System wie `anan` gewinnen. Es ist auch erfolgreich versucht worden, einen Teil der Daten eines Dashboards aus gemischten Quellen zu erhalten.

Implementierung. Im endgültige Zustand ist `anan` eine äußerst schlichte Software; aus einer knappen Beschreibung des Verhaltens ließe sich innerhalb von ein bis zwei Monaten eine neue, vollständige Implementierung entwickeln. Dies geht auf eine Reihe von Vereinfachungen mit dem Zweck besserer Nutzbarkeit und höherer Effizienz zurück. Sensoren werden in einer schematischen Form ähnlich wie beim Werkzeug `awk` entwickelt, in mittels der Sprache Lua 5.3 Muster und Auswertungsalgorithmen eingetragen werden. Passt das Muster, wird der entsprechende Algorithmus ausgeführt. Das Verarbeiten von Informationen aus Dateien wird auf besonders kompakte Weise unterstützt, weil unter Linux (der relevantesten Plattform für `anan`) die meisten wichtigen Informationen in Dateien bzw. Pseudodateien zu finden sind. Allerdings ist der Zugriff auf alle Funktionen des Betriebssystem möglich, wenn auch deutlich weniger bequem. Es zeigt sich, dass sich so gut wie alle interessanten Variablen in dieser schematischen Form beschreiben lassen, viele davon sogar ziemlich

knapp — je knapper, desto einfacher die interaktive Nutzung.

Auch die benutzten Datenaustausch-Formate (das von Lua für Literale benutzte) und Netzwerkprotokolle (UDP mit `zlib`-Kompression und authentifizierter Verschlüsselung per `NaCl`) sind so einfach wie möglich gewählt — seltene Datenverluste sind in Kauf zu nehmen, wenn eine niedrigere Effizienz oder schlechtere Nutzbarkeit die Alternativen wären.

An das untersuchte System werden nur die geringsten Anforderungen gestellt. Zum einen verringert das die Wahrscheinlichkeit, mit dem eigentlich interessanten Phänomen zu interagieren und es möglicherweise zu verdecken. Zum anderen erleichtert es die Portierung auf andere Betriebssysteme oder Architekturen, obwohl gegenwärtig Linux/x86_64 das beherrschende System ist. Die Übersetzung für Android/aarch64, Mac OS X/x64, Linux/microblaze und weitere Systemen gelang meist völlig ohne oder mit geringen Anpassungen. Ob die übersetzten Programme korrekt arbeiten, ließ sich in Ermangelung passender Hardware nur in wenigen Fällen prüfen.

Ergebnisse. Weil `anan` der erste Vertreter einer neuen Klasse von Werkzeugen ist, sind zwei Arten von Ergebnissen nötig:

1. Ist `anan` performant genug, um praktisch eingesetzt werden? Wo liegen die Grenzen?

2. Falls die Antwort positiv ist: In welchen Situationen kann `anan` als Debugger erfolgreich eingesetzt werden?

Die Beantwortung der ersten Frage ist konzeptuell einfach. Dazu wurden Versuche angestellt, in denen `anan` die Daten von einer zunehmenden Anzahl von virtuellen Maschinen oder Container verarbeiten muss. Auch die Performanz bei großen bzw. vielen Sensoren ist gemessen worden. Es stellte sich heraus, dass eine einkernige Variante von `anan` ohne weiteres mit dreihundert bis fünfhundert untersuchten Systemen umgehen kann, nicht aber mit tausend. Eine Erweiterung auf mehrere Kerne ist (verhältnismäßig einfach) möglich, aber nicht beabsichtigt, weil gegenwärtig keine so großen Systeme untersucht werden.

Die Beantwortung der zweiten Frage ist komplizierter. Es ist nämlich möglich, dass ein Debugger auf der Ebene eines ganzen Rechenzentrums kein sinnvolles Konzept ist, diese Ideen aber zu besserem Monitoring führen; oder umgekehrt könnte dynamischeres Monitoring bereits wie ein Debugger nutzbar sein. Tatsächlich wurde festgestellt, dass `anan` in Verbindung mit der Datenbank eine mit sonstigen Monitoring-Werkzeugen vergleichbare, wenn auch etwas schlechtere Performanz erzielt. Die Latenz erhöht sich durch die zusätzlichen Verarbeitungsschritte um einige

Sekunden. Allerdings ist es möglich, mit akzeptabler Performanz viel komplexere Daten darzustellen, etwa die Korrelation zwischen offenen Sockets und Hauptspeicher. Diese Art Anfrage lässt bei üblichen Monitoring-Werkzeugen entweder gar nicht oder nicht effizient stellen, sofern man sie nicht durch Überlagerung zweier Diagramme in einem Dashboard umgeht.

Ohne die Möglichkeit, per SQL-Abfrage mit `JOIN` verschiedene Zeitreihen oder sonstige Datenquellen zu korrelieren, wäre sinnvolles Debuggen kaum möglich. Allerdings ließen sich die folgenden beiden (für Fehlersuche bedeutsamen) Konzepte auch ohne relationale Datenbank umsetzen:

1. Ein **Breakpoint** ist ein Sensor, der üblicherweise ein negatives Ergebnis gibt und beim Eintreten eines interessanten Ereignisses ein positives Ergebnis sendet.

2. Eine **Watchlist mit Ringpuffer** ist eine Liste von Sensoren, welche die meiste Zeit unauffällige Daten sammeln, aber um das Feuern eines Breakpoints herum möglicherweise Informationen über das Auftreten des interessanten Ereignisses enthalten.

Beide lassen sich per SQL-Abfrage implementieren und müssen daher nicht fest in `anan` einprogrammiert sein, obwohl es der Performanz vielleicht helfen könnte.

Auch die umgekehrte Frage lässt sich vorsichtig positiv beantworten: Es gibt gewisse Fragestellungen, bei denen eine Betrachtung jedes einzelnen Systems nicht genügend Informationen liefert, um ein Problem zu identifizieren, die Gesamtbetrachtung mit `anan` dies aber ermöglicht. Diese Fragestellungen sind allerdings selten. Ob das an der kurzen Einsatzdauer von `anan` liegt, ist unklar. Es bedürfte weiterer Arbeit, um zu ermitteln, ob `anans` insgesamt eher schwierige Nutzbarkeit ein Problem der Klasse *Cluster-Debugger* oder der konkreten Umsetzung `anan` ist. Im ersten Fall könnten immerhin die Erfahrungen mit dynamischerem Monitoring auf die üblichen Monitoring-Werkzeuge übertragen werden.

Chapter 3

General Introduction; Notation

Simple style. This thesis is intended to be easily readable. It applies the following techniques to achieve that:

- Short sentences;
- no "elegant" style;
- no academic understatement, especially complex nominal phrases ("reduction of production by introduction of a prediction") and passive voice ("this was done ... that was measured");
- strong division in sections and subsections.

Complex arguments or constructions are most often presented in a top-down style (part A consists of parts A_1, A_2, ...), although occasionally, the bottom-up style (part A_1, part A_2, ... yield part A) was given preference.

Paragraph leaders. Most paragraphs begin with a short sentence or leading caption to introduce, sometimes summarise, the paragraph's main statement. This feature should enable the reader to quickly skim well-known material and focus on new or interesting material. Often, the paragraph leader is written in a rather catchy and provocative way.

Lists instead of prose. When arguing for or against a decision or summarising results, many aspects may be relevant. It is most convenient to list them in a concise tabular style. The alternative would have been writing long paragraphs which contain all arguments without clear delineation. Of course, this uses more space on the page.

Mathematical notation. Most of this thesis is prose which discusses the design, execution and results of the software `anan`. Sometimes, mathematical notation is needed, as in $e^{i \cdot \pi} + 1 = 0$ or $(\mathtt{a}, \mathtt{b}) \longleftarrow (\mathtt{b}, \mathtt{a})$. The

typography is chosen in a way that formulae do not stand out but rather blend into their surrounding text.

Program code. All examples of program code or program input/output are typeset in `monospace`. Very short examples such as `int x = a+b;` are shown inline without highlighting, somewhat longer examples appear as paragraphs with line numbers and highlighting:

```
1  #define SWAP(a, b) {int tmp = a; a = b; b = tmp;}
2  void quicksort(int numbers[25], int first, int last){
3      int i, j, pivot, temp;
4      if (first<last){
5          pivot=first; i=first; j=last;
6          while (i<j){
7              while (number[i]<=number[pivot] && i<last) i++;
8              while (number[j]>number[pivot]) j--;
9              if (i<j) /* no semicolon after SWAP because {block} macro */
10                 SWAP(number[i], number[j])
11         }
12         SWAP(number[pivot], number[j])
13         quicksort(number, first, j-1); quicksort(number, j+1, last);
14     }
15 } /* adapted from: Akhil Bhadwal, https://hackr.io/blog/quick-sort-in-c */
```

Longer code and examples of interactions can be found in the appendix on p. 95. — Sometimes, names of tools are typeset in `monospace` in order to stress that the tool is a piece of code.

Chapter 4

Introduction to Debugging

Overview. Every computer programmer who has written more than a few lines of code knows the fearful realisation: "This program does not do what it is supposed to do." In short: The program is buggy, it has a bug.* The wish to remove bugs does not require any further explanation.

Levels of debugging. As will be shown in section 9, debugging *single-threaded isolated non-networked* programs is fairly well understood. Once one of these restrictions is lifted, almost arbitrary complexity can appear. Complexity is the natural enemy of debugging. Clearly, a single-threaded program without any loops is easier to understand and therefore to debug than a program with loops. In fact, a ladder of features can be found. Every feature adds capabilities to write more concise or efficient programs. At the same time, every step up the ladder makes debugging harder. The ladder could contain the following steps:

- branches and loops;
- global mutable state;
- multiple threads of execution;
- threads with shared memory;
- network communication over possibly faulty lines;

*Anecdotally, the term "bug" for a faulty behaviour in a program is said to stem from vacuum tube or relay based computers where a moth — a bug — hid in one of the warm parts and burned together with it. Once the bug was found, computation could resume. Others claim the term is used in analogy to bacteria and viruses which are also referred to as "bugs". The first explanation certainly has the advantage of being more charming.

- pieces of the program run on heterogeneous architectures (CPU, microcontroller, FPGA, GPU) and need to communicate.

This list is not exhaustive — there is no upper limit. Many in-between steps are missing. Also, steps may be skipped: Not every networked program has to contain global mutable state or multiple threads of execution.

Two ways to improve debugging. As in most branches of computing, there are two ways of improving:

1. better tools;
2. better methods.

The first option means better tools are used for the same problem, e. g. more powerful or more specialised debuggers. They typically have stronger demands on hardware. As can be found in section 9, most of the progress focuses on this area.

The second option means the same tools are used with better understanding of the underlying problems. Alternatively, programs can be designed in a certain (defensive) way to be easily debugged if necessary. Far fewer results can be reported in this area. Here, this thesis also finds little to add (but see page 79 for a few remarks). It mainly tries to present better tooling.

Inspection and interaction. A *symbolic debugger* allows the user to follow a program's execution. Depending on the type of debugger and program under consideration, the following things can be *inspected*:

- code to be executed;
- register values;
- contents of memory regions;
- variables;
- file system operation;
- data structures (as graphs or trees or similar visualisation).

The list is by far not exhaustive, although the most often used objects are listed. — Nearly every debugger has some means to *interact* with the program, e. g.

- return from a function;

- change a register value;
- change a variable.

The ability to inspect is more important than the ability to interact with and change the program's execution for the following reasons:

- It is pointless to interact blindly with a system that is in an unknown state.
- Interacting is a nuanced process. It cannot be put into a consistent scheme as easily as displaying control and data flow.
- Often, seeing the (faulty) behaviour of software suffices to understand and correct the bug.

Introspection on the source level. Obviously, a program without branches and loops does not require more tooling for introspection than pen and paper — such a "program" is hardly more than a formula into which values are inserted. Once loops and branches are added, i. e. the realm of Turing-complete algorithms is reached, a symbolic debugger becomes a more helpful tool. As long as the observed bug is to be found inside the program (as opposed to outside the program, e. g. in faulty hardware or file system corruption), the symbolic debugger is in principle the correct tool to find any bugs: Symbolic debuggers are currently among the best universal tools in cases where the behaviour can be solely explained based on the program's source.

Introspection on multiple levels. Typically, programming does not happen as isolated as imagined in the previous paragraph. Rather, disks run full, file systems get corrupted, network connections go down or drop packages or become slow. All of these events cannot be found directly by a symbolic debugger, although hints to them (e. g. error codes and exceptions) appear. Additionally, the symbolic debugger has to go to great lengths to follow the program's control and data flow without interfering too much — and the debugger might still interfere, leading to differences in behaviour ("Heisenbugs") [Weissenbacher, 2012].

Multiple levels might interact already once there is more than a single thread of execution and shared memory or message passing is used. The behaviour cannot be explained strictly following the source code. Now, the operating system's scheduling becomes relevant and especially the fact that it typically is non-deterministic. (Theoretically, including the operating system's scheduler into the code under observation could help explaining the behaviour. But this approach is impractical. It can be taken

meaningfully only in embedded systems where both the operating system and the application are small.) This effect only grows in scope when more steps on the ladder are taken.

Mocking. In the best case, all bugs leading to the relevant unwanted behaviour are to be found on the same level. If there is some way to shield every involved system from each other, then every system can be debugged on its own. For software, there is such a method: Every part of the system can be replaced by a "mock" replica [Karlesky et al., 2007] — a piece of software which behaves like the external system in every discernible way. Of course, such an approach is laborious and may introduce Heisenbugs itself. Nevertheless, it presents a systematic pathway to the bug.

Interacting bugs. The previous paragraph's situation is still idealised for the following reasons:

- There are no systems without any bugs (the exception proves the rule).
- Mocks cannot be built meaningfully (i. e. with the necessary faithfulness, but still cheap enough) for complex things such as file systems or network, let alone custom hardware.
- Since mocks are also software, they can introduce new bugs.
- The relevant part containing the bug's actual source may be overlooked. The user can easily assume it is not part of the system (e. g. the electrical supply may be at fault but the user does not consider checking it).
- Bugs can interact in complex ways.

In this case, no universal approach such as mocking can be given. Debugging in real life is a game of hide-and-seek. Therefore, it is important to have tools which support *introspection at multiple levels*. For symbolic debuggers, this is mostly out of scope, although relevant aspects are often supported, e. g. showing the values of named variables and registers. In the case of `anan`, it will be seen that such a simplistic scheme cannot be supported; more complex views have to be developed and implemented by the user.

Testing is not debugging. In fact, testing is the *dual* of debugging in the following sense: If a test fails, this means some code (either the code under test or the test itself) is buggy. Debugging helps to find the source

of the bug. After the bug is fixed, (automated) testing can make sure it does not return. In practice, tools for the preparation and automation of tests make good helpers for debugging because they provide debugging-related facilities such as mockers. Therefore, these two tasks are somewhat overlapping.

Chapter 5

Introduction to High Performance Computing

Overview. Moore's Law [Moore, 1965] used to describe accurately (up to about the year 2010) that the number of transistors in integrated circuits grows exponentially. The number of transistors is a proxy for all sorts of performance characteristics, e. g. clock rates or number of floating point operations per unit of time or power. It used to be possible to argue (albeit tongue-in-cheek) in the following way: "Let's buy a computer in three years and get results in a day — or buy a computer now and get results in a few years." Moore's law was understood to state the raw computational power of a single CPU doubles approximately every 18 months. If more performance than that is requested, multiple CPUs and finally multiple network-connected compute nodes are needed.

Many problems in engineering can be parallelised to a large extent. The complexity classes L and FL (deterministic calculation in logarithmic space [Cook and McKenzie, 1987]) are usually assumed to contain all problems which can be parallelised efficiently. (This does not mean that the actual parallel algorithm necessarily uses logarithmic space.) These classes contain solvers for ordinary differential equations; finite element models; k means clustering; training neural networks; approximate lattice reduction ... — and many other scientific and engineering tools. High performance computing (HPC) is the art of using slow CPUs, memory and networks to reach fast computation.

Levels of parallelism. A well-written parallel program is friendly to the following levels of parallelism and other peculiarities of HPC [Hager and Wellein, 2010, chapter 1], [Ruckert, 2015]:

- **Pipelines.** Today's CPUs split any instruction's execution ("retir-

ing") in tiny steps such as fetch instruction; decoding; fetch operands; execution; store result. After the first part of the pipeline is done, the next instruction can enter the pipeline. Thereby, theoretical speedups proportional to the pipeline's depth are possible (although never achieved in practice). Additionally, when instructions are scheduled, the CPU might notice that a different ordering or different naming of registers is advantageous ("clever scheduler"), although sometimes, this task is partially or fully relegated to the compiler's code generator ("clever compiler").

- **Branch prediction.** In order to keep pipelines filled, the CPU has to know many cycles in advance, which instruction is going to be executed. In the case of *conditional branches*, heuristics are used (although the code may state whether it believes this to be a likely branch or not) to *speculatively* follow the branch which the CPU predicts to be taken. In the case of a mispredicted branch, the pipeline needs to be flushed, incurring some cost.

- **Instruction level parallelism.** Certain operations such as integer multiplication/division and most floating point operations take more than one cycle. If there are multiple arithmetic units, many arithmetic operations can be executed simultaneously. This requires both "clever compilers" and "clever schedulers".

- **Single Instruction, Multiple Data (SIMD).** With SIMD, the programmer is able to explicitly request the instruction level parallelism he wants. Usually, special wide *vector registers* with a corresponding instruction set are used.

- **Caches.** Main memory is much slower than a CPU's registers. In order to bridge this gap, there are multiple levels of data and instruction caches: Memory which is a bit slower but also a bit larger than registers. Today's CPUs have two or three levels of caches which differ in their positioning between register and main memory.

  If a datum from main memory or a higher level cache is loaded into a register and stored in a lower level cache, the cache controller stores a whole *cache line*: a fixed-length contiguous area of memory. This behaviour enables the cache controller to prefetch areas of memory if the controller thinks they are likely to be needed in the near future. Some CPUs offer instructions to advise the memory controller on likely future fetches.

(The hierarchy of caches can be extended to hard disk drives, network attached storage and tape libraries which are again larger and slower/higher latency by orders of magnitude.)

When memory addresses whose content has been cached are written to, the cache controller has to take special precaution to make sure the program always sees the most current version. Famously, "There are only two hard things in Computer Science: cache invalidation and naming things" (Phil Karlton).

- **Multicore.** Computers with over a hundred cores are commercially available. Multicore systems add some more complications, e. g.:
  - In some cases, not all CPUs can run at the peak frequency (to avoid overheating).
  - Main memory is shared, some part of the cache may be shared.
  - In larger systems, CPUs may be located on different sockets, yielding non-uniform communication latencies and speeds.
  - Even on the same socket, latencies and speeds may be non-uniform.
- **Hyperthreading.** It has been observed that all the above-mentioned methods still leave the arithmetic and floating point units idle for too long. Therefore, another set of registers can be added to each CPU so that it can feed the instructions of two different threads into the pipeline. Although the single-threaded performance may be degraded (because of administrative costs), the theoretical multi-threaded performance doubles.
- **Extensions.** Special hardware can complete certain tasks much faster than general-purpose CPUs, e. g.
  - *Graphics Processing Units* for code which has little memory communication and few branches;
  - *Field Programmable Gate Arrays* for reconfigurable programming;
  - *Application-specific integrated circuits* for specialised tasks (there is dedicated hardware for deflate (de)compression [Summers and Engineer, 2008]; fast Fourier transform [Despain, 1979]; cryptography [Yuan et al., 2018]; etc.).

- **Multiple compute nodes connected via network.** Networking is much slower than any communication inside a single compute node. Also, typical implementations have non-deterministic latencies, occasionally leading to severely delayed or entirely dropped packets.

HPC is the art of rewriting straight-forward algorithms in clever, but complicated ways to maximise performance on all levels at once.

Tooling. Naturally, tools develop, and any specific tool mentioned now might be obsolete in a few years. But classes of tools and established members of these classes do not lose relevance quickly. Compilers tend to have facilities which report on various optimisations which were performed or could not be applied. Likewise, there are special memory and cache profilers that can help to increase the program's cache locality.

- **Libraries.** There is a set of libraries, based on the *basic linear algebra subprograms* BLAS [Duff et al., 2002], such as LAPACK [Anderson et al., 1990] for numerical linear algebra or FFTW [Frigo and Johnson, 1998] and many more. Using these libraries does not make a program automatically run optimally in a HPC setting — nevertheless, these implementations are likely to be faster than any homegrown code.

- **Open Multi-Processing (OpenMP).** A program written in Fortran or C/C++ can be annotated with *compiler directives* which specify how the program can be executed by parallel threads of execution. The compiler generates code which uses the operating system's corresponding threading facilities. If the operating system's threads were to be used directly, any possible gain in efficiency would be outweighed by a loss of clarity: OpenMP's notation is straight-forward as it was specifically developed for applications in HPC.

- **Open Message Passing Interface (OpenMPI).** After a program utilises a single node's CPU resources fully, the next logical step is using multiple nodes. In OpenMPI, the only way to exchange data between nodes is *message passing* (as opposed to automatically shared memory). OpenMPI could have been listed under the first point, "Libraries", because it is implemented as a library. — It presents function for unicast, multicast and broadcast communication between the nodes of a compute cluster; the typical numerical data types are supported. The library's design stresses raw performance above all.

HPC requires a good understanding of both the problem's and the solution's domain.

Chapter 6

The Contribution of anan

Introduction. The tool `anan` enables the user to measure anything interesting about a set of networked computers as easily as possible in order to allow him to inspect faulty or unexpected behaviour of a compute cluster. The remainder of this section tries to unpack and fill in the details of this task.

Analyse and navigate. Originally, `anan` was meant to be an acronym **an**alyse **a**nd **n**avigate. It was found that the required scope of `anan` is much richer than just navigating data structures. Instead, a *global view* of a cluster needs to be assumed. Additionally, the user should never be restricted in his choice of possible variables to be measured. The following workflow needs to be supported:

1. Find (symptoms of) faulty behaviour.
2. Develop a theory about possible reasons.
3. Based on the theory, predict the system's behaviour.
4. Perform measurements to falsify the theory. (In case of success, `goto` step #2.)
5. Accept the theory once the user is sufficiently convinced.
6. React based on the theory.

Step #1 is outside of `anan`'s scope — if a compute cluster behaves as it is supposed to, no debugging is necessary. Likewise, step #6 is mostly outside of `anan`'s scope, although some small interventions might be taken directly from `anan`. Steps #2, #3 and #5 are an application of the scientific method [Popper, 1989] to debugging compute clusters. (Of course, many

more approaches to science exist, each leading to a different method of debugging. Also, following the purist approach, theories can never be accepted, only rejected. This is clearly unhelpful in practice.) Only in step #4, software tooling is required. This step is anan's place in the workflow.

Scope of measurements with anan. Since anan is a software tool, anything it measures needs to reach it via software. This is mostly the operating system's task because anan operates in user space. If issues of hardware are to be considered, the operating system has to support this type of measurement. Even with anan, measuring a fan's speed remains impossible as long as the kernel has no access to the fan's speed. Likewise, if the kernel does not present correct information, anan cannot show meaningful data. The tool anan is not guaranteed to work in the presence of broken operating systems or low-level hardware failure. Other than these two cases, anan must support every possible data source, even if not out of the box.

Similarly, the collected raw data may need cleaning and additional analytical steps. Since there are many rich toolboxes such as Matlab [Schweizer, 2016], Mathematica [Wolfram, 1991] or even Microsoft Excel, anan offers only the most elementary functions for data analysis, relegating everything else to more specialised tools. The main feature enabling this is export into simple and standardised formats such as CSV [Shafranovich, 2005].

Finally, anan should never interfere with any part of the system under consideration and thereby interrupt normal (or even buggy) operation. This cannot be achieved to the fullest extent: A pure (side-effect free) program cannot do anything. Instead, the software parts of anan should minimise their use of resources. This directly contradicts the main goal of anan: the ability to swiftly measure anything the user requests. The chosen trade-off will be presented in the section on p. 61. Similarly, usability and diversity of feature are usually contradictions. Here, the line of trade-off is less clear.

The plan for this thesis. The project anan is closely related to the following three fields:

- software debugging;
- monitoring;
- high-performance computing.

It is therefore necessary to give a brief overview of the *state of the art*. In order to avoid exceeding the scope, pointers to standard reference are

given, along with an excerpt of some ideas of specific importance for `anan`. Then, the tools are discussed and motivated which were used to develop `anan`, leading into an overview of the *implementation techniques*. A full reference can be found on p. 95. The results of using `anan` in different contexts can be grouped in two parts:

- *synthetic benchmarks*: How efficiently does the tool use certain resources?
- *practical benchmarks*: How good is the tool for a specific task?

Since `anan` is a new kind of tool, the results need careful interpretation in order to assess the merits of both the new class and this particular interpretation. Also, some other findings reached during the benchmarks are outlined in broad strokes.

Main results. It is hardly possible to give a convincing summary about something not yet presented. Nevertheless, the two main results can be hinted at:

1. On a technical level, `anan` fulfils the requirements for both a cluster monitoring and a debugging tool. Anything which the curious user might ask about a compute cluster can be measured or calculated by `anan`.
2. On a practical level, it remains unclear (although somewhat plausible) if the approach of cluster debugging and the implementation `anan` are sound approaches.

Part II

State of the Art

Introduction. This thesis occupies a spot at the intersection of cluster computing, (single-threaded) software debugging and monitoring or quality control. The result is a single-node software which can be used for debugging and related tasks such as monitoring.

First cluster debugger. To the author's best knowledge, `anan` is the first implementation of a cluster debugger. (He would very much appreciate to be informed of other implementations or other prior art.) Accordingly, the state of the art is `anan` itself. Instead of referring to other implementations of this concept, a short review shall be given of the main ideas of cluster computing; monitoring; debugging (in increasing degree of detail).

Chapter 7

Clusters

Introduction. The grandfather of cluster computing may well be the original Beowulf cluster [Becker et al., 1995]. The difference from previous attempts at large-scale computation is that commodity hardware is used:

- **Cheap FLOPS.** When the considered problem can be parallelised well, it is more economical and may be more efficient to buy hardware optimised for FLOPS per unit of money. Nevertheless, the system has to remain homogeneous.
- **Many CPUs.** That involves adding as many computational units (this today includes GPUs and possibly FPGA or ASIC cards) to each node. This helps reduce the price and keeps communication between nodes low.
- **TCP/IP over Ethernet.** Although the original Beowulf used UDP/IP over Ethernet, TCP/IP has become standard for many applications in high performance computing. The important point is not the decision for a specific networking stack, but rather for a standard stack. This may include InfiniBand today.
- **UNIX.** Today, UNIX survives mostly as Linux (together with several descendants of BSD which are not relevant in high performance computing). Again, the more economical (and arguably easier to use) option is chosen. A similar attitude is taken with regards to the user land and application software.

This paradigm is not the only one in use today, although it may be dominant. Therefore, `anan` can not assume it is running on a homogeneous system (see, e. g. some experiences with embedded systems on

p. 71). Similarly, the vast majority of high performance computation has a scientific background. Nevertheless, no application can be ruled out. (Neither assumption was found to influence `anan`'s design greatly.)

There is much more to be said about both Beowulf and other architectures and many more aspects of high performance computing [Hager and Wellein, 2010].

Chapter 8

Monitoring

Introduction. Since anan does not attempt to advance the state of the art with regards to monitoring, this section can be kept short. Most of its material is based on the surveys [Aceto et al., 2013, Tamburri et al., 2020, Fatema et al., 2014]. As will be discussed shortly, monitoring is a field that does not favour peer reviewed journal articles, let alone scholarly monographs. Therefore, the review is necessarily scarce and fragmented.

Scope of monitoring. There are different users of monitoring, amongst them

1. providers of cloud services;
2. operators of data centres ("collocation centres", "server hotels");
3. users of cloud services.

Accordingly, different properties are of different importance ([Aceto et al., 2013, §5]), e. g.

- **scalability**: how well does the monitoring system deal with too many sensor values;
- **elasticity**: how well does it deal with wildly fluctuating sensor values;
- **adaptability**: does the system interfere with normal operation, leading to "Heisenbugs" [Weissenbacher, 2012], [Agans, 2002, chapter 5];

(See [Aceto et al., 2013, §5] for the complete list.)

A trend towards higher dynamics, extensibility and richer visualisation can be found. Most likely, this is the result of two past developments:

- Increase in computational power available for monitoring;
- more flexible software platforms accessible (e. g. web browser based applications implemented in "HTML5").

Classification ("Taxonomy"). The Stanford Linear Accelerator Center has been maintaining the list of (network) monitoring tools at `https://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html` for over two decades. [Fatema et al., 2014] sees a trend from general-purpose to (specific) Cloud monitoring. This does not only mean that the tool measures fewer lower-level data points, it might also lead to application-specific data: A tool called "AzureWatch" (as of November 2021 available as "CloudMonix" at `https://cloudmonix.com/aw/`) can only be used sensibly with Microsoft Azure. This is by far not the only such example. Therefore, the first dimension for the classification is the type of cloud which can be monitored (this includes the distinction between general and single-purpose). This distinction is found to be correlated strongly with other variables:

- **General-purpose** cloud monitoring tools are less feature-full in general (they are less "adaptable", less efficient, less robust etc.) but more extensible, scalable, interoperable.
- **Proprietary** cloud monitoring tools are more feature-full in general but are less extensible and interoperable.

For obvious reasons, it cannot realistically be expected that a general-purpose tool is as efficient as a proprietary tool and vice versa (which is why [Fatema et al., 2014, §8.1] does not list this as future work).

Monitoring in the wild. [Tamburri et al., 2020] is a quantitative study of monitoring in the industry. The core question — split into eight smaller questions — is if monitoring is used adequately. The study found ("Findings 1–8"):

1. Monitoring is rarely seen as relevant.
2. It is performed manually or with ad-hoc tools.
3. Involving non-IT personnel in incident management is harmful.
4. Incidents are often handled manually.
5. Monitoring is not considered important by management; therefore, tools require long learning and are hard to use.

6. More complex cloud architectures seem to be somewhat more reliable. This finding is not entirely clear.

7. More complex cloud architecture may require (and does have) more and better monitoring.

In discussing the existing literature about monitoring, [Tamburri et al., 2020, §6.1] concludes:

> From our preliminary analysis of the literature, the existing solutions are fragmentary and the field is rather immature; the present literature is not organic and there is no apparent continuity between the gray and research literatures.

The term *grey (or gray) literature* refers to all informal (i. e. not only peer reviewed) literature and even oral statements. This source of knowledge has been overlooked traditionally, although it is useful in applied sciences. See, e. g. [Paez, 2017] for the importance of grey literature in medicine and [Mathews, 2004] in computer science. Obviously, only a minute part of work being done in the fields of monitoring and debugging is ever published, be it informal or in peer reviewed journals. A review of grey literature requires substantially more work because there is no formal way of gathering the relevant sources. With peer reviewed literature, most relevant material is gathered in a handful of journals. Grey literature might be scattered in blog posts, technical memoranda, lecture notes or conference proceedings.

The conclusion states:

> Our results offer a glimpse of the untapped potential behind using more structured approaches for cloud applications monitoring and continuous quality improvement.

This sounds like an allusion to `anan`'s goal.

Summary. Monitoring has to be everything to everyone, find issues before they happen and be so easy to use that it needs no user. Of course, the reality is the exact opposite: Monitoring is always specialised, it can only react to past events (which have to be specified precisely in advance) and requires a trained operator with a general understanding of all involved technologies and the system's architecture. A tool like `anan` cannot solve this disparity fully.

Chapter 9

Debugging

> By June 1949, people had begun to realize that it was not so easy to get a program right as had at one time appeared. It was on one of my journeys between the EDSAC room and the punching equipment that the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs. [Gillmor, 1987]

Introduction. The problem of grey literature holds in a similar way as in the case of monitoring. It may be even worse: As per the introductory quotation, debugging is as old as programming*. Since monitoring is used mostly in the context of large-scale computation, the scope of debugging is also larger than that of debugging. There is much "folklore" and tricks passed down from generation to generation. Nevertheless, some of it appears in the formal literature and will be summarised here.

Learn these n tricks. Although [Agans, 2002] does not state it as bluntly, debugging is really simple once some tricks are applied. [Agans, 2002] proposes the following "debugging rules" (which he calls "DEBUGGING RULES!"):

1. Understand the system.
2. Make it fail.

*Maurice Wilkes — from whose memoirs the quotation is taken — is considered to be one of the first programmers in history. Since this thesis does not focus on the history of computing, it should only be mentioned in passing that Ada Lovelace's — who is also sometimes mentioned in this capacity — programs were not executed on a stored-program computer and therefore should not constitute programming in the modern sense.

3. Quit thinking and look.
4. Divide and conquer.
5. Change one thing at a time.
6. Keep an audit trail.
7. Check the plug.
8. Get a fresh view.
9. If you didn't fix it, it ain't fixed.

These nine instructions are clearly helpful for finding bugs, removing them and ascertaining correct behaviour afterwards. The following criticism is partly due to the age of [Agans, 2002]:

- **Complete understanding is not possible anymore.** This text book was written at the heyday of the "dotcom bubble". Even very complex systems from the late 90's pale in comparison to today's commodity systems and certainly compute clusters with several layers of abstraction. Rule #1, "understand the system", requires reading the relevant manuals and data sheets "cover to cover". This is not practical anymore. Unless it is precisely known where an error is to be found, this hint is impossible to execute. If a WiFi modem is attached via USB to a laptop of a certain brand running a certain operating system and some error condition appears with regards to networking, any of these technologies (including the laptop's BIOS, CPU, PCI bus, memory and cache system, operating system, drivers, user land etc.) could be at fault. The relevant manuals fill tens of thousands of pages.
- **Bad or complex design.** Most of the examples include systems designed in an "electrical engineering style": Small, well understood boxes (such as transistors and resistors) are connected in standardised ways into bigger boxes (such as DC converters and op-amps) and finally the complete system. In the previous paragraph's example, a part of the complete system "broken network" is designed in this way, namely the hardware. This way of abstraction is not used for the software. Instead, the typical paradigm is layering abstractions: A web application runs in a web browser in a operating system's user land; the operating system was loaded by a boot loader

which was prepared and run by a BIOS. (But see [Herder et al., 2006] for a modular approach which apparently turned out to be a dead end.) Even the BIOS is not necessarily the first code run after a cold boot. These layers cannot be separated as cleanly as the electrical engineer's boxes. In fact, the layers are often not adhered to: For reasons of better performance, some hardware appliances use direct memory access. They bypass the "official" path via hardware interrupts and processing by the operating system. Rules #4 and #5 do not make sense in a complex setting where the system's parts are "complected" with each other instead of being easily separated.

- **No logs.** Many of the examples ("war stories") assume elaborate logging to be present. This cannot be taken for granted anymore. In fact, much software has only a very limited set of log messages which can be produced. Essential information cannot be extracted from the software without rebuilding it — which is impossible for many use cases. Of course, this does not apply to software which the user develops and debugs for himself.

Debugging based on the scientific method. The scientific method [Popper, 1989] is the basis of debugging. In fact, according to [Metzger, 2004], a single science or craft is insufficient. The following "ways" are required:

1. The way of the detective;
2. the way of the mathematician;
3. the way of the safety expert;
4. the way of the psychologist;
5. the way of the computer scientist.
6. the way of the engineer.

Amusingly, both [Agans, 2002] and [Metzger, 2004] are fond of citing Sherlock Holmes as a prime example of an effective approach to debugging.* — The six ways are more specific for software than the nine rules

*The author does not believe this to be accurate at all. Since detective fiction is not even tangentially relevant to this thesis, this argument is kept as short as possible: Detective fiction is written with the solution in mind. First, the crime is constructed. Then, a few hints are chosen. In order to weave the story's plot, this material is presented together with the detective's personality. With this trick, the author can present an almost omniscient personality who is more intelligent than the author and most readers [Stein, 1995].

in [Agans, 2002] and can be applied in cases where the nine rules cannot. Some of the criticism can be alleviated by this. Strictly following the "ways" is helpful only for single-level problems, e. g. a fault in a single program. If several levels interact and therefore, there is a large distance between symptoms and root cause, the approach is less applicable. Nevertheless, for the limited domain of debugging single-threaded programs, the approaches presented by [Agans, 2002] and [Metzger, 2004] are probably nearly optimal.

Debugging in the wild. The study [Perscheid et al., 2017] pursues two goals:

1. Review the recent literature about debugging;
2. study the practice of debugging in the industry with a mostly quantitative approach.

The study admits about the first goal:

> Henry Lieberman had to say that „Debugging is still, as it was 30 years ago, largely a matter of trial and error."

(The citation refers to [Lieberman, 1997].) Nevertheless, his literature review gives a great overview of the research landscape.

Regarding the second goal, it is almost impossible to summarise the findings without gross distortions. The following statements, therefore, are to be taken as sweeping generalisations from [Perscheid et al., 2017, §3.3].

1. Debugging is not taught anywhere. In the best case, it is learned "by osmosis" while observing more experienced coders at work. Possibly, in recent times, universities started mentioning debugging in their courses.
2. The general method is similar to the scientific method, based on theories which can be disproved ("the bug is not in a certain piece of code or a certain subsystem"). No "scientific" tools such as lab books are used, let alone digital tools such as wikis.
3. Hypotheses are formulated (and accordingly disproved) as early as possible.
4. Symbolic debuggers are used almost universally. The ancient technique of inserting `printf` statements is not preferred.

5. Almost no-one understands more than the most elementary features of debuggers. Especially *back-in-time* debuggers (see p. 36) and static analysis are unknown.

6. There is almost no correlation between programming language and type of debugging tool used.

7. Debugging takes a lot of time. Bugs are everywhere, although most of them are easily found.

8. Most people agree debugging did not become harder in the past ten years.

9. The hardest bugs could not have been found by (direct application of) the methods presented in [Agans, 2002] and [Metzger, 2004]. Special-purpose tools (e. g. memory debuggers such as `valgrind`) are very helpful, though.

10. Learning from past mistakes is helpful. People do not like doing that.

11. Test-driven development is far less helpful than expected.

An ad-hoc classification of debugging techniques. This is a list of debugging tools as encountered by the author. For every tool, its applicability to cluster debugging shall be assessed.

- **Tracing via (manually inserted)** `printf` **statements.** This technique stems from the time when `printf` literally meant printing on paper. It is applicable in every system with some means of outputting data to the user. The closest cognate in cluster debugging is probably log files.

- **Tracing via built-in execution tracer.** A tracer prints the line of code being executed. It may also interpolate the variables in that line. Since this creates immense amounts of output, this tool is rarely used in general-purpose programming. (Also, a tracer needs to be implemented from scratch for each programming language.) A typical niche for this technique is scripting languages with an unusually complicated syntax and semantics such as the UNIX shell. If a shell script has a relatively short suspicious part, using `set +x` and `set -x` to enable and disable tracing sometimes is helpful. The closest cognate in cluster debugging is selectively enabling or disabling verbose logging modes in specific tools. Detailed monitoring also qualifies.

- **Tracing via hardware *traps*.** Most modern computing hardware has built-in support for debugging. Special code can be executed triggered by events such as

  - register value changed;
  - RAM location read or written;
  - program counter reached a certain address;
  - illegal instruction or operation executed.

  Most symbolic debuggers use hardware traps. This requires very sophisticated code to support the corresponding hardware, operating system and all of its binary and linking formats. A similar tool does not exist for clusters. The most similar technique is implemented via *alerting* the user based on events occurring at single nodes. Another similarity can be found in special hardware enabling remote access to the production hardware such as IPMI [int, 2013].

- **Omniscient debugging.** A less blatant term is *back-in-time* debugging. This is usually implemented as an add-on to a symbolic debugger and somewhat similar in usage to reading an execution trace. While running the program, the debugger periodically stores a part of the program's state in memory. If the debugger is stopped at, e. g. the line `x = y + z;` and the user learns that `y` and `z` have unexpected values, he can ask the back-in-time debugger where these values have been set. (The same information could have been found by searching an execution trace which clearly is less convenient.)

  The notion of omniscient debugging does not really make sense in the context of debugging a cluster. The closest cognate is replaying logs and monitoring data. It is very apparent that there is a huge gap in usability between reading execution traces and using a back-in-time debugger. A similar gap exists with regards to cluster debugging.

Part III

anan: analyse and navigate

Introduction. This thesis is not an engineering report. Any technical issues are being discussed only when novel solutions are proposed or a non-obvious solution (out of several standard approaches) is chosen. Therefore, no complete presentation of `anan` is given.

Note: The *tool* `anan` consists of two *binaries*, called `anan` and `anand`. If there ever is a risk of conflating the tool and its binaries, the words "tool" or "binary" are given to disambiguate.

The first section (chapter 10) discusses the choice of technologies for most aspects of `anan`; the second section (chapter 11) discusses the actual implementation of `anan`. Naturally, the tooling around `anan` cannot be discussed independently. For this reason, these sections refer to each other recursively.

Chapter 10

Existing Technologies and Tools

Overview. The most important question is which programming platform (including the programming language) to chose. Based on this, it is usually easy to find surrounding helpers (such as libraries or build and documentation tools).

10.1 Programming Language Lua 5.3

Much of the material in this subsection is based on [de Figueiredo et al., 2017, Ierusalimschy, 2016].

Note: Versioning. The languages in the Lua family are closely related, but nevertheless incompatible programming languages. Once a project decides to choose Lua 5.1, an "update" to version 5.2 is not an update, but rather a rewrite with the advantage that much code can be reused. The term "version" does not apply to Lua versions in a similar way as it does to most other programming languages. Accordingly, code is usually written to run under exactly one version. Backwards or forwards compatibility is a feature which requires careful planning. Henceforth, the term Lua refers to Lua 5.3 except when explicitly noted otherwise.

New versions of Lua are released whenever the developers decide they want to do so; there is no fixed release schedule. When the first preparations for implementing `anan` started, Lua 5.3 was the most recent version published. Although there were already first alpha versions of the then-upcoming Lua 5.4, settling on a moving target was considered an unwise choice. Therefore, then's stable version 5.3 was chosen. Viewing the differences between these two versions, even today there is no good reason to switch.

Overview about Lua. Lua 5.3 is a lightweight imperative program-

ming language. The source distribution is a few hundred kilobytes in size (under 20 kLOC, including headers and comments) and implemented in the subset of C which also compiles as C++. For reasons of portability, by default, only C features available in the ANSI C standard are compiled into Lua, although on Linux and other POSIX compatible platforms, a few more features (such as the dynamic linker) are available. Lua strives for utmost minimalism: There is only one way to structure data (a mixture of array and hash table called "table") and only a few ways to organise control flow, although `goto` is available [Dijkstra, 1968].

Scripting. The reference manual [de Figueiredo et al., 2017, §1] calls Lua a scripting language. Commonly, the opposite of scripting is called "systems programming": Scripts are throw-away tools developed quickly and without too much regard to efficient resource usage, elegance or reusability. The only goal of "scripting" in the most narrow sense is to solve the exact problem on hand and forget about it. Lua allows this style of programming. In fact, this is the correct mode of operation for a debugger: Any command issued to a debugger is a minuscule throw-away program. The scripting nature of Lua shows in the following ways:

- **No compilation.** Lua is interpreted from source (after compilation to bytecode), the interpreter is available at runtime. Code can be changed and generated at runtime, allowing high flexibility. It will be seen later in chapter 11 how this property is enabling `anan`'s work flow.

- **Garbage collection.** Resources like memory, file descriptors or database handles are freed once they become inaccessible [de Figueiredo et al., 2017, §2.5]. This happens in a conservative manner: Only once the garbage collector can prove that the "mutator" (the program manipulating the resources subject to collection) can never access the resource, the resource is put on a list of objects to be collected. Eventually, all resources will be freed (this is guaranteed by Lua's garbage collector), unless the program crashes. But garbage collection is inherently *lazy*: A resource may be freed much later than possible. Therefore, a Lua programmer still needs to exert some care with regards to resource usage.

- **Duck typing.** "Duck typing was named after the 'duck test', by James Whitcomb Riley: 'When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck'" [Milojkovic et al., 2017]. In plain words: Types are not declared to

the compiler or interpreter. Instead, whenever an operation is attempted with a value, it is the programmer's task to make sure the value can perform that operation.

Lua does not allow to declare types for variables. Instead, *values* have types. This removes some of the bureaucracy needed in statically typed programming languages such as C. For an interactive program where the user should know the types of all data he is operating with, duck typing is an ideal fit.

- **Syntactic sugar.** Short or mnemonic or otherwise special notation for features with otherwise uniform syntax is sometimes called syntactic sugar. A famous example is the notation `i++` in C and related languages for the more verbose `i = i + 1`. This phenomenon is purely related to syntax; one cannot say that C is "syntactic sugar" for assembly language because the translation from C to assembly language is not trivial. — Lua offers special (short) syntax for literal values such as complex table structures [de Figueiredo et al., 2017, §3.4.9] and object oriented programming [Ierusalimschy, 2016, chapter 21]. Short notations for commonly used operations are convenient for an interactive program. In the best case, Lua itself can be used as the interactive language for `anan`. (In fact, there was never any need for another "interactive shell" language for `anan` because Lua was already deemed sufficiently concise.)

- **Dynamic and extensible.** Scripting languages tend to enable a higher degree of dynamics than compiled languages. The expression `x+y` in C has one of the following exactly two meanings:

  - integer addition
  - floating point addition.

  Moreover, the compiler is able to decide the meaning and generate corresponding code. Conversely, in C++, the operator + can be overloaded. Depending on the precise way of overloading the operator, the compiler might generate code which looks for the corresponding meaning only at runtime. Thus, not even the compiler knows what code `x+y` will run. C++ is more dynamic than C.

  Lua is even more dynamic than C++. The following are some examples of Lua's dynamic nature:

  - **Metatables.** Every data type (except for `nil`) can have a *metatable* [de Figueiredo et al., 2017, §2.4]: a data structure which controls what happens in otherwise undefined situations. Metatables generalise operator overloading. More examples such as several implementations of object oriented programming (including multiple inheritance) or alternative module systems can be found in [Ierusalimschy, 2016, chapter 20].
  - **Environments.** Lua does not have global variables. Instead, in the default configuration, variables which are not defined to be `local` behave much like global variables in other lexically scoped programming languages. This default global *environment* can be replaced by another table. Using this facility, a block of code can be understood to run in a different context, e. g. a context in which undefined variables (normally `nil`-valued) yield error messages. Another example could be helper variables which automatically come into living based on other variables.
  - `eval`**.** Scripting languages often have a function `eval()` which calls the interpreter on an arbitrary string. A similar facility is available in Lua. This facility is perhaps the most dynamic feature conceivable in programming languages. It allows the implementation of tiny embedded languages. This feature is essential for an extensible tool like `anan`.

- **Weak performance.** Almost all scripting languages are slow. The main reasons are:

  - Interpreted, not compiled language;
  - high degree of introspection, dynamics, extensibility;
  - garbage collection.

  Lua has all of these features. Since they are highly important for `anan`, this means that `anan` can hardly compete with special-purpose debugging tools and should not try to do so. Nevertheless, there are two ways to speed up `anan` if this should prove necessary:

  1. **Native code.** Lua can be extended by native code implemented in C or C++ [de Figueiredo et al., 2017, §4], [Ierusalimschy, 2016, part IV]. The API is rich enough to allow even features such as threading to be added as a (small) loadable library [Fernández,

2016]. Additionally, it is a truism that slow code typically has a small number of hot-spots: tight loops which benefit most from translation to native code. If such hot-spots were to be found in `anan`, it is likely they could be translated to C quickly.

2. **Just-in-time compilation.** This is really the same idea as in (1) but the implementation is very different: The program is interpreted as usually, except that it is instrumented to find hot-spots automatically. Once a hot-spot is identified, it is compiled to native code. Since at runtime, there is more information available about the hot-spot than at compilation time, the just-in-time compiler can optimise the hot-spot better than the ahead-of-time compiler could ever do. There are a few reports about cases where just-in-time compiled code runs faster than compiled code.

Since `anan` does not need to perform huge computations, it is *a priori* unlikely that its performance would become critical. It is, however, expected that higher (single-core) performance means more systems can be debugged with more sensors at the same time.

If performance should ever become an issue, the Lua implementation `luajit` [Pall, 2008] can be used instead. No attempt has been made to find out if `luajit` does in fact run `anan` unchanged. Since `luajit`'s implementation is tightly coupled to the CPU and operating system it runs on, this would make `anan` vastly less portable.

10.2 Protocol Layer

Overview. The following functionalities are required for the communication between `anan` and `anand`:

- serialisation
- optionally: compression
- cryptography

Serialisation. There is a number of standard serialisation schemata (for most of which there is Lua tooling available):

- XML [Bray et al., 2006]

- JSON [Bray, 2017]
- ASN.1 [Barry, 1992, rec, 1988]

The first two are text based languages, the latter is a family of "encoding rules" of which most are binary.

Not ASN.1. Naturally, ASN.1 (especially when using "packed encoding rules") is much tighter than any text based encoding. On the other hand, ASN.1 requires each message to conform to a scheme. Without the scheme, a message cannot be parsed. Although this is also true for XML in a very limited sense (namely that entities cannot be resolved without the document type definition), real-world XML fragments and all JSON datasets can be parsed without reference to schemata. This rules out ASN.1: With ASN.1, all sensors have to either specify their schema (unnecessary bureaucracy) or conform to one of a limit set of schemata (too strong restriction of flexibility).

Not XML. The "X" in XML means extensible. In fact, XML has a stunning degree of extensibility: An entirely normal *modus operandi* consists of taking XHTML (the XML version of HTML), embedding scalable vector graphics [Ferraiolo et al., 2000] and XSLT templates [Clark et al., 1999] and processing them in some way. The processing application likely uses XML's elaborate namespace mechanisms to keep the distinct languages apart.

Additionally, XML offers redundant features such as attributes. The following fragments are equivalent ways of expressing the same data:

```
<group>
  <k1>v1</k1>
  <k2>v2</k2>
  <k3>v3</k3>
</group>
```

and

```
<group k1="v1" k2="v2" k3="v3"/>
```

and

```
<group>
  <entry k="k1" v="v1">
  <entry k="k2" v="v2">
  <entry k="k3" v="v3">
</group>
```

To XML experts, these three ways may mean slightly different things. If `anan` chose XML as serialisation format, it would need to distinguish between these three (and any other imaginable encoding) or support some sort of normalisation.

Not quite JSON. In JSON, there is only one (sensible) way of expressing the above-mentioned datum:

```
{
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
```

Lua has a natural way of deserialising data from its literal syntax [de Figueiredo et al., 2017, §3.1; §3.4.9]. There are libraries to serialise into this syntax. These observations (and some possible, though not likely complications with JSON described on p. 66) are the main reason why Lua's native syntax was chosen over JSON.

Compression. The most popular choice is `zlib` [Gailly and Adler, 2004]. It is neither the fastest nor the best compression nor a fair compromise between both (which is currently best achieved with [Collet and Kucherawy, 2018, Vladyka, 2018]). A quick measurement found that compression and decompression speeds range in the dozens of megabytes per second. This speed is certainly sufficient for `anan`. Considering that compression is only rarely required, the actual compression rates achieved by `zlib` are immaterial. (It is not entirely obvious if after compression, text-based notations like Lua's literal syntax are more or less space efficient than binary notations like ASN.1 with "packed encoding rules".)

Cryptography. The standard choice to ensure encryption, data integrity and authentication between server and client is TLS [Rescorla and Dierks, 2018]. There are probably dozens of high-quality implementations of TLS. Many of them are free and open software and easily accessible from Lua. Additionally, some network tools such as `tcpdump` understand TLS and can inspect the data stream when given the private keys. On the other hand, TLS is very complex and requires some non-trivial setup (key and certificate management). Therefore, the library NaCl (pronounced "salt") was chosen, specifically the implementation TweetNaCl [Bernstein et al., 2014]. This library provides encryption, integrity and authentication with very little bureaucracy (for the importance of this, see chapter 15) and is reasonably fast. A possible security hole is discussed on p. 54.

10.3 Deployment via GNU autotools

Introduction. Installing and configuring software is a tedious but essential step before using it. Different distributions of Linux have converged to different package formats and conventions. Other operating systems have their own facilities. Finally, there are some cross-platform packaging tools which purport to support "all" (common) operating systems. It is a herculean task to create packages and build recipes for all these tools. Instead, the focus was on creating a source distribution which a package maintainer can easily can turn into whatever format is preferred.

`configure` **scripts.** The standard procedure under UNIX and derived systems for building and installing software is the incantation

```
./configure ; make ; make install
```

The `configure` script tries to find out the relevant information in order to prepare a `Makefile` which specifies how to build and install the software. Traditionally, `configure` scripts have been written manually. Today, there is a group of tools collectively called "autotools" [Vaughan et al., 1999] which generate the `configure` script based on a short description of the project to be built. Almost all software package repositories know how to deal with "autotools" based packages.

Compiling Lua code. Lua is an interpreted language. It is advantageous to deploy `anan` and `anand` as statically linked executable objects for the following reasons:

- The tools should run on the target systems with minimal requirements.
- They should not interfere with any other installed software.
- No update of the target system should break `anan`.
- The binaries should not have any preference where should to be installed or executed.

In Lua's ecosystem, there is tooling which automatically does the following:

- Gather a program's sources;
- gather a program's binary objects (static libraries);
- find Lua's executable's binary object;

- create some boilerplate to glue all these together.

This process can be plugged into the `configure` script. Depending on the build environment, some or all facilities presented by POSIX can be compiled into `anan`. Since partial compliance to POSIX is rare — an environment is either (almost) fully compliant or not at all —, no effort has been made to automatically check for each feature and generate corresponding code.

Bring your own source. The (autotools-generated) source distribution of `anan` brings its own copy of all necessary libraries except for the `libc`. Although this may restrict the licensability of `anan` (some libraries are licensed under the GNU Public License which "infects" everything linked to it), this is not problematic as long as the original license is not removed. This also makes `anan` more self-sufficient and thereby more easily deployed.

Configuration ("run control file"). Every user of `anan` and every site using it are likely to want their specific configuration. This includes site-specific or project-specific queries and data-sources. To make this trivially easy, it was decided that both `anan` and `anand` expect their configuration to be the Lua script `./init.lua` relative to the current directory. Due to this choice, arbitrary configuration actions (including recursive inclusion of more configuration scripts) can be performed at startup. The existence of a `init.lua` is, however, not required.

10.4 SQLite

Introduction. Software debuggers do not persist all data by default. Typically, there are elaborate commands to dump memory areas and other structures in every relevant format. Storing this data to files comes as an after-thought. Loading it into the debugger (or other tools) for further scrutiny is a rare activity. Hardware debuggers tend to write their data into on-chip memory and consider this sufficiently persistent. Debuggers and databases do not typically pair. Once `anan` was used as a (makeshift) monitoring tool, it had to store its data in some persistent place. A database seemed like a good choice for that. After the database was set up, it became clear that a database is vastly more convenient to use than the facilities described in [Adler and Kebschull, 2020] (see p. 56 for a summary on the issues). — Most of the material in this subsection is based on SQLite's official documentation.

History of SQLite. SQLite was originally conceived as an extension for the "Tool Command Language" TCL. In the given use case, some tool was required which was in between a simple key-value store and a full-blown relational database management system (RDBM). On one hand, complex queries as with RDBMs were needed. On the other hand, the laborious setup as with then's (and similarly today's RDBMs) was prohibitive. Quickly, most features of SQL 92 [mel, 1992] were added, closely following the example of PostgreSQL [Stonebraker and Rowe, 1986] (from which many non-standard additions were learned). As a result, SQLite is certainly the most-often deployed database. According to `https://www.sqlite.org/mostdeployed.html`, accessed on October 27th 2021, "SQLite is the second mostly [sic] widely deployed software library, after libz".

Some relevant features. The following aspects of SQLite made it reasonable to add SQLite (in favour of other databases) to `anan`:

- **Single file.** SQLite stores the whole database as a single file on the file system (although an open database may need up to two additional files). This is the basis for most of the following points.
- **No client-server architecture.** Since every access to the database happens via the operating system's IO layer, there is no database server. Every "client" opens (and locks) the database file and reads or writes as it sees fit without requiring network or inter-process communication. The lack of a server greatly simplifies the setup of an SQLite database and basically removes the need to do any maintenance. This obviously is a requirement for easy adoption, the main feature in connection with `anan`.
- **Weak concurrency.** Unlimited concurrent reads are possible, but only one write at the same time. A server could schedule requests such that multiple concurrent reads and writes are possible. But since SQLite rejects the server-client architecture, only a well thought-out system of locking the database file can be used to ensure data validity under all circumstances [Härder and Reuter, 1994]. In `anan`'s case, some missing or erroneous entries do not hurt too much. Still, the database must never become corrupted.

  The introduction of a write-ahead journal has alleviated much of the trouble with concurrent writes. On commodity hardware, millions of inserts per second are not unheard of.
- **Language bindings.** There are several adapter libraries for Lua 5.3, although Lua's ecosystem is often weak in libraries. The chosen li-

brary provides a thin layer on top of the official API in C. By clever use of Lua's introspection abilities, most queries can be written in a natural style. An example:

```
db:query('SELECT name, age, temperature FROM thermal1
          WHERE name LIKE ? AND temperature BETWEEN ? AND ?',
                           name_pattern,            min,  max)
```

Of course, this coding style is not suitable for interactive use. — A more "introspective" style would have written `:name_pattern`, `:min`, `:max` instead of the question mark `?` and relied on variables of the respective names in the enclosing scope.

- **Sloppy typing.** Owing to the scripting nature of TCL, SQLite has a tendency to be more dynamic when it comes to typing of columns. SQL advises the database to give each column of each table a type. Once the column `AGE` is defined to be a non-negative integer, storing the string `"42"` in that column should yield some sort of error condition (and certainly the attempt with the string `"forty-two"`). By default, SQLite ignores this restriction: As in a dynamic programming language, not the variable (the column) has a type, but the value stored in the variable. Just like the variable `age` in a dynamic language can store any type of value at any given time, in SQLite, any column can have any value. This feature avoids type mismatches between `anan` and SQLite.

- **No access control.** Since SQLite should not require any setup, access control cannot be implemented on a database-level. Anyone who can open the database file can do anything he wants to it, be it via SQLite's API or raw file access. Therefore, access control can not be meaningfully implemented. This takes away some complexity with regards to setting up and moving databases.

- **JSON support.** If performance was of high important, then normalising the database to comply with at least the first normal form [Codd, 1983] would be required. But most of the time (and always in interactive settings), query performance is less important than expressive power. In the tiny case study on p. 68, it was convenient to encode some dataset as a JSON array of strings stored in a database cell (the normal, much faster solution would be one of several schemes involving an additional table with foreign keys —

depending on whether reads or writes are more frequent). Therefore, SQLite's built-in support for JSON (and especially the fact that JSON-based expressions can be used in *view*s) comes very handy. Any support for JSON from Lua's side is certainly helpful but does not aid in formulating SQL queries — for this, only built-in support is relevant.

- **Modern "standard" features.** Certain features are almost considered granted today, although they require much background work. Amongst them are:
  - Unicode support;
  - no artificial limits (file size, number of rows, columns etc.);
  - strings may contain arbitrary bytes (including null bytes) which may not conform to any encoding;
  - (well-documented) C-based API.

  Some of these are mere niceties for `anan`, some might prove essential later.

- **Grafana data source.** There is a plugin for Grafana which allows querying an SQLite database from Grafana. The results are being interpreted either as a single time series or as a set of time series. Although SQL is a less convenient language for formulating monitoring queries than specialised query languages, all relevant queries can be expressed. Additionally, no network round-trips are needed if the database resides on the same machine as Grafana.

10.5 Grafana

Introduction. A command line tool can present data only in limited ways. When `anan` was being designed, no provision was made to plot data from `anan` directly. Instead, data should be easy to export in various formats. External tools should ingest and display the data. — Since `anan` was given the additional task of monitoring, this was not sufficient. The choice of data to be shown had to be decided by the developer. The user was to be presented ready-made plots and should have little possibility to change things around.

Requirements.

- A *panel* is a time/value plot of a single or a few variables per system under consideration.
- A *dashboard* is a collection of about a dozen panels, typically centred around a topic. Some dashboards focus on an overview of a single machine.
- Single nodes or groups can be interactively removed or added to a dashboard via on-screen widgets.
- The range of time to be shown can be changed interactively.
- The dashboard is recalculated and plotted at regular intervals (which can be chosen by the user).
- Interesting pieces of plots can be centred around a topic. Some dashboards focus on an overview of a single machine.
- Single nodes or groups can be interactively removed or added to a dashboard via on-screen interactive elements. The groups can be defined by the application to be monitored.
- Data sources can send their data directly to Grafana or export it in a way which `anan` can ingest. Other models may be possible.
- Certain error conditions lead to alerts via email or other means (SMS; chat; ...) with detailed information attached. The flow of alerts can be regulated in order not to fill the recipients' inboxes unnecessarily or even cause a denial of service in the messaging system.

Most of these requirements can be directly fulfilled with the combination of Grafana and `anan`. Since the user interface of Grafana runs as a "HTML5" application in the user's browser, the demands on Grafana's server are modest.

Chapter 11

Concepts

Light weight demons. The client `anan` connects to many server instances of `anand` running on each system under consideration. To prevent bugs stemming from interference with the debugged system [Weissenbacher, 2012], the demon should do as little as possible. When no tasks are given, the demon should sit idle and wait for instructions. (As of now, `anand` does send almost empty keep-alive messages. This was introduced to keep traversed NATs open [Rosenberg et al., 2010], but was found to be unnecessary).

Demons are programmable. Obviously, any debugging tool must be able to inspect anything in its domain as fully as possible. Therefore, the demons have to be programmable. Each demon executes zero or more *sensors*. A sensor is a short snippet of code which measures or computes a quantity and sends it to `anan`.

`awk` **sensors.** Theoretically, sensors could be arbitrary code. It was, however, found that most sensors are built along the following scheme:

- setup code;
- open one or more files for reading;
  - split the file into (generalised) lines;
  - split the line into (generalised) fields (“records”);
  - execute some code if the lines’ records match some pattern or condition;
- summarise;
- tear-down code (send result).

The main reason for this is believed to be UNIX's idea of "everything is a file". In Linux, the main interface for curious user-space processes are the pseudo file-systems `proc(5)` and `sysfs(5)`.

The above-mentioned scheme is, incidentally, very similar to the one used by `awk` [Aho et al., 1979]. The question whether this incident is really pure luck is discussed briefly in chapter 15.

No `netlink`. Another interface is given by `netlink` sockets [Neira-Ayuso et al., 2010]. No effort has been made to also expose `netlink` sockets to `anan`'s sensors for the following reasons:

- The documentation for `netlink` is spread throughout the Linux kernel's source and a few manual pages which provide too sparse information.
- The communication via `netlink` uses a binary protocol. This protocol differs even for different endpoints of `netlink`.
- Although `netlink` enables access to an incredible wealth of information, there was never even the slightest need to access this information.

Note: There are more low-level kernel interfaces which can give detailed information about the kernel or user-space programs, e. g. `ioctl` and `fcntl` for files or `ptrace` for processes. Neither were considered relevant. Still, since `anand` has access to the full interface presented by POSIX (this includes POSIX' dynamic linker to load arbitrary libraries at runtime) and more, it has (cumbersome) access to these facilities.

No artificial intelligence. Artificial intelligence and machine learning [Bishop, 2006] constitute a broad class of methods which allow the programmer to specify only an algorithmic scheme; the details are filled in algorithmically based on "training data" — data for which the algorithm's correct behaviour is known. Although this algorithmic class is extremely rich, it is conceptually not the right tool for debugging. The methods in this class are non-interactive. In the realm of debugging and software testing, there the following non-interactive tools:

- **Fuzzing.** Random input is presented to the system under consideration [Sutton et al., 2007]. This class of methods is mostly interesting for bugs which are hard to find by an "honest" user but can be found by systematic search for untypical input. Nevertheless, this type of bug can show (in the worst case as a security vulnerability).

- **Symbolic execution.** The program is not executed with concrete input but with *symbolic variables* [Cadar et al., 2008, King, 1976]. At critical points, a theorem prover is called to either show that no illegal behaviour can occur or find input which leads to illegal behaviour. A hybrid with concrete values is possible [Sen, 2007].

The general approach taken by `anan` is to stay close to the analogy with software and to a lesser degree hardware debuggers. Fuzzing would translate to feeding random input into the cluster. Symbolic execution does not translate into anything meaningful; constructing a formal model of software execution is much easier than of a complete cluster including all relevant hardware. Both approaches do not fit well into `anan`. However, it is possible to use machine learning to detect anomalies [Omar et al., 2013] in monitoring. That approach has not been followed with `anan`.

Security. In order to access privileged kernel facilities, `anand` has to be run by root (or debug only user-space programs using user-space tools). Obviously, `anand` may accept instructions (code for sensors) only by an authorised sender. Therefore, public key cryptography [Bernstein et al., 2014] is used to *authenticate* all messages. Keys are stored alongside the binaries of `anan`. An attacker with access to the binaries should also have access to the keys and vice versa, therefore, this policy does not widen the attack surface. Additionally, all messages are *encrypted*, although this is far less pressing than authentication: It is unlikely that an eavesdropper (i. e. someone inside the network) can learn anything interesting from sensor results.

Denial of Service. An *active attacker* can resend previous messages to `anand`. Thereby, he can force any instance of `anand` into a state it has been in previously. In that case, `anan` will reject its messages, since they don't match the expected configuration. After a few seconds, `anan` will alert the user that the attacked instance of `anand` has not been sending data for a certain time. The active attacker can repeat this for each instance of `anand`. This leads to a denial of service. The attack can be mitigated by requiring increasing *sequence numbers* in every message to `anand`. This has, however, not been implemented because this attack is considered too far-fetched. An active attacker has probably more interesting targets than a debugger.

Unreliable transport. In almost all settings, the systems under consideration which run `anan` and `anand` are in the same network segment. Therefore, not much effort is needed to keep transport reliable. Single reports as in `https://www.openmymind.net/How-Unreliable-Is-UDP/` (retrieved October 2021) and similar anecdotal observations make UDP sockets [Postel et al., 1980] seem quite reliable even over several hops. Addi-

tionally, since no connection is being needed for each server, `anan` can easily handle hundreds of systems under consideration without running out of file handles or socket slots in the kernel's table. Finally, even a loss of a few packets should not make a large impact.

Compression. The choice of UDP as transport layer restricts the maximum size of a message to a theoretical $2^{16} - 1$ bytes. In practice, packets of this size are more likely to get lost due to fragmentation on the IP layer. Sometimes, demons send messages with a size fluctuating around the maximum size. In this case, messages larger than $2^{16} - 1$ bytes are lost because they cannot be decoded fully. — Sensor results are mostly textual or numeric data encoded as text and therefore easily compressed. The often-used `zlib` [Gailly and Adler, 2004] is used to compress sensor result prior to encryption/authentication. Very rarely, even after compression the maximum size is exceeded. In this case, old sensors which accumulated throughout the session can be dropped.

Interactive use. For interactive use, one measurement per second was deemed sufficient. This is far away from real time usage: Each round-trip costs at least a second (both `anan` and `anand` process data in blocks of about a second), adding at least two seconds between a configuration update and a corresponding sensor result. Of course, developing a sensor and inspecting its results requires usually more time. Therefore, a measurement per second is good enough for interactive debugging and certainly for monitoring, although not necessarily for real time usage.

Breakpoints; Variables; Watch-lists. In symbolic software debuggers, a typical pattern is the following:

- set a *breakpoint* at a place in close vicinity of the suspected bug;
- once the breakpoint fires, inspect the CPU's registers or the program's *variables*;
- record parts of the program's path through the program or values of the program's variables for later scrutiny.

No code. A debugger has to operate on some code, be it source or binary code. A cluster is running code on several levels, e. g.:

1. **application specific code**;
2. **deployment code**: ensure certain software is installed and configured in a standardised way;

3. **run control**: "init" scripts run at every start-up and shutdown to enable or disable certain services.

As there are already symbolic debuggers for most programming languages, `anan` is not needed for (1). In fact, the author inspected the first fifty entries on TIOBE's Index of the most popular programming languages (as published in January, 2021) and found that every language on the list has at last one symbolic debugger.

A typical deployment tool such as `ansible` [Hochstein and Moser, 2017] already features a "stepper" — a program which runs the deployment's actions one by one and allows inspection of all relevant parameters. Since deployment is a relatively simple task in comparison to (1), such tooling is probably sufficient. In any case, `anan` is not needed for (2).

Similar considerations apply to init systems: Historical init systems such as SysVinit were shell scripts. If debugging was ever necessary, the normal tooling for shell scripts was available. Contemporary init systems such as `systemd` [Binnie, 2016] are vastly more complex and come with their own set of debuggers and profilers. It is also unclear, how to execute `anan` in the context of a booting system: It would certainly require at least some file systems to be mounted and networking to be running. Once this state is reached, most of the init system's task is done and not much is left to be debugged. Accordingly, `anan` is not needed for (3).

Analogy: Hardware debugger. The fallacy common to (1)–(3) is that `anan` should be mostly concerned with code running on one machine. But really, a higher level view is required. All machines contributing to the distributed systems need to be *viewed as a unit*, and this unit is the system to be debugged. This notion is similar to that of a hardware debugger. With an oscilloscope or a logic analyser, every single wire could be measured. Such an approach is inferior to a dedicated hardware debugger which is synthesised alongside the system to be debugged. It contains logic to measure relevant signals and store them in buffers based on (possibly external) triggers, communicate with control hardware and maybe "mock-up" other parts of hardware which are not delivered yet or expensive or dangerous to use. All of these features help to debug a system without running alongside its source. Such an *event-based* approach is the basis for `anan`.

A functional approach. The approach presented in [Adler and Kebschull, 2020] was found to be less helpful than expected. Since it was ultimately unsuccessful and is described in some detail there, only the failure will be described very briefly. — The functional notation supports the following workflow:

- collect one or more time series;
- optionally, map them along the axis of time or host name;
- if more than one, zip them together;
- optionally, perform another final calculation (using the other axis).

No other workflows are possible. Although pieces of code can be nicely named and composed, yielding readable and "self-documenting" code, many interesting queries cannot be expressed. A part can be shoehorned into this workflow, but the example in chapter 13.4 cannot. Another relatively common example is *filtering*: Not even difference sequences of variables can be expressed in the functional workflow. Of course, a special function could be added for applying (e. g. FIR) filters to the functional workflow. This, however, takes away its simplicity and ease of composition. That observation was the final blast to this workflow.

Vectorising to the rescue? In parallel to the functional notation, a short-hand to the above-mentioned workflow was developed, based on the vector formalisms in numerical programming environments such as Matlab [Angermann et al., 2020] and `numpy` [Oliphant, 2006]. The following differences were necessary:

- **No real vectorisation.** A term like `data.lambda "x/1000"` (which takes each element of the time series `data` and divides it by 1000) does not use a library like LAPACK [Anderson et al., 1990] to vectorise the computation (although it was considered to include LAPACK).
- **Lazy.** Instead, the term `data.lambda(...)` creates a value which (via Lua's metatables) behaves like an array. Whenever a value is requested, it is calculated on demand.
- **No cache.** For the following reasons, no caching is used:
  - Usually, the calculations performed via `lambda` and other notations (not introduced here) are small.
  - Cache coherence is a non-trivial problem.
  - The cache may use a lot of memory. This can be alleviated by a cache management/retirement strategy.

All of this complexity very likely does not buy better performance. Together with the functional notation, its vector short-hand was scrapped. The code was not removed from the code base, however, because it can still be reused for the relational formalism: If the result of an SQL query can be understood as a matrix which is indexed by host names and time, it can be endowed with the short-hand functional notation.

A relational approach. The approach outlined in this section and discussed further on p. 67 was not found by systematic inquiry, but it appeared as a makeshift solution that turned out better than expected. Instead of understanding sensor results as time series, they are viewed as tables of a relational database system. *SQLite* was chosen as database. The functional workflow translates into simple `SELECT` statements of the form

```
1  SELECT f1(f2(x, y, ... )) FROM ...
2  WHERE hostname IN (...) AND time BETWEEN ... AND ...
```

Variables, breakpoints, watch lists (again). In the relational approach, variables are represented as relations (data base tables). Breakpoints are queries which should normally return an empty set. Once the breakpoint query returns a non-empty set for a certain time range, the breakpoint is considered "firing". The watch list, then, is another query around the same time range. Both breakpoints and watch lists can be implemented as SQL "views" (names assigned to queries). The creation of views from breakpoint or watch list statements could be automated. Currently, this has not proven to be necessary.

Integration. The requirements of both `anan` and `anand` are modest. Integration `anand` into the init system `systemd` is trivial because it does not require any other specific services (other than networking and the file system) to exist. Likewise, since `anand` logs to `stdout`, any logging policy is easy to implement with existing tooling, be it discarding, storing or aggregating logs.

Part IV

Results, Summary, Future Work

Chapter 12

Overview

Scope. Since cluster debugging is a novel area, all comparison have to either compare `anan` with itself in different settings or compare `anan` with another specialised tool. In the latter case, the question is mainly "how much worse does the general-purpose tool perform than the specialised tool?"

Synthetic benchmarks. Before any meaningful comparison, it has to be made sure that `anan` can deal with a decent amount of data. This includes both a large number of systems under consideration and a (reasonably) large number of sensors. (The number of sensors is naturally limited by the maximum size of a UDP datagram.)

Practical benchmarks. Before practical benchmarks, `anan` is like an answer looking for the corresponding question. It is naïve to expect that problems will appear which `anan` is a perfect solution for. Rather, problems from practice were considered for which `anan` is a not necessarily optimal solution. This was expected to lead to applications in neighbouring fields, thereby finding the appropriate niche for `anan`.

Chapter 13

Benchmarks

Overview. Three different synthetic benchmarks have been performed. The third one led up to practical benchmarks and finally some practical application.

13.1 Synthetic Benchmark: 10 Containers on a Laptop

Goal. This benchmark should show that `anan` works with more than one instance. It was anticipated that once an easily automated test environment is available, development should be faster: Every change can be tested quickly against a standardised environment.

Methods. A Docker image was built, consisting of little more than the executable `anand` and the corresponding key and configuration files. Ten identical instances of this image were run. All of them were connected to the same network. The executable `anan` was run in a similar container. To rule out interaction with the environment outside the containers, all containers were run without access to the host network.

Results.

- The program starts and stops flawlessly.
- Both individual sensors and sensors sent to all instances are run.
- It can be ascertained that "one-off" sensors do, in fact, run only once. (This functionality was removed later on. Although it was never used, it became a burden to maintain.)
- Each individual configuration is executed.

- Very little resources (both memory and CPU) are utilised. This result is limited because Docker interferes with a clean measurement. Since further benchmarks were planned, no attempt has been made to measure more precisely.

Summary. The functionality of `anan` works as expected. No statement can be made about performance.

13.2 Synthetic Benchmark: In Ten Virtual Machines on a Blade Center

Goal. This benchmark was designed to show the limits of `anan`: How many systems under considerations could be debugged; is the bottle neck IO, CPU or RAM? Since it was decided that `anan` should not be used on data centres with more than about 1000 nodes, the anticipated results were to be interpreted the following way:

- If the performance would not allow 1000 nodes plus ample room to spare, then algorithms and data structures were to be improved, until this goal was reached.
- If the performance allowed far more than 1000 nodes, then additional features could be added that might weaken `anan`'s performance, but add capabilities.

At this point in time, `anan` implemented only the bare minimum: Sensors were hard-coded, the responses were parsed and only stored in memory. It was clear that some additional facilities are needed for any successful debugging session.

Methods. A Blade Center was used to set up 10 virtual machines. Each of the virtual machines had two CPUs and ample RAM for the experiment. The virtual machines ran Ubuntu 18.04, on top of that a then-current Docker. On each of the ten virtual machines, 100 instances of a similar Docker image as in the previous section were executed. Since they were all running inside a *Docker swarm* (a native clustering environment for Docker), each container was connected to each other through a central hub, yielding a star topology. A similar container was constructed for the executable of `anan`. For simplicity, that container was run on the swarm's *management node*.

Each container running `anand` was throttled to 0.01 CPUs. Additionally to the executable `anand`, it also ran a "payload" (a simple CPU-intensive

program) that could be "debugged" with `anan`. Theoretically, the 0.01 CPUs of the 100 containers should sum up to a single CPU being utilised, leaving the second CPU idle.

Results.

- `anan` could execute a single (hard-coded) sensor on about 1000 systems under consideration (some containers crashed and could not be restarted reliably).
- The CPU utilisation of the nine virtual machines on which `anan` did not run was about 1.5–2. Since that version of `anan` was a single-threaded program, `anan` used one CPU fully, leaving the other to the remaining 100 containers which were therefore CPU-starved.
- No precise measurement of lost sensor results was performed. Manual observations (by repeatedly counting the received messages per second) found that most messages were received and processed.
- The command line interface to `anan` remained usable, although sometimes input was slightly delayed.

Summary. The performance of `anan` was sufficiently good. Additional features may be added to reach a usable tool. It was unclear, how far `anan` was from reaching limits. The effects of shaky network connections were unknown. Only a handful of hard-coded sensors were tested, most of them of no relevance to actual debugging.

13.3 Synthetic Benchmark: In 140 Virtual Machines via OpenStack

Goal. Since in the previous benchmark, the limits were not clearly reached, it was decided to run a final synthetic benchmark: This benchmark should be failed by `anan`. Meanwhile, `awk` sensors have been added to `anan`. It was assumed that the more generic sensors might weaken `anan`'s performance.

Methods. Most of this section is based on joint work with Niclas Günther [Günther, 2020]. — An OpenStack installation (dubbed "Goethe-Wissenschafts-Cloud", or GWC) was used to create 141 virtual machines with a similar virtual hardware to the experiment on the Blade Center. This OpenStack installation used SuperMicro machines with two AMD Opteron™ processors 6172 (that is, $2 \cdot 12$ CPUs) and 64G of RAM as compute

nodes. An analogous Docker and swarm setup was used to connect 140 out of 141 nodes. The swarm manager was node 141. This node also ran `anan`.

Results.

- Although $100 \cdot 140 = 14\,000$ instances of `anand` were expected to run, only about 10 000 were ever active at the same time. The ultimate reason for this failure was not found. It was ascertained that this is not due to a lack of RAM, CPU, IO bandwidth or disk space.
- The low reliability of the GWC made the following results slightly suspicious. Nevertheless, inexplicable failures most likely have only a negative influence on performance (although the opposite is not impossible).
- About 90% of the sensor messages were not processed. This did not follow a clear pattern: Sometimes, the same node's messages were processed every second for a short period of time; sometimes, a node seemingly went silent for a few minutes. Sometimes, all nodes dropped about 90% of their messages equally (yielding an event rate of about 0.1 per second).
- This result was understood that the critical size for a single-threaded `anan` is about 1000 nodes.

Summary. It was found that `anan` reached maturity for practical applications: The performance of a version with a mostly complete feature set is good enough for about 1000 systems under consideration. It was assumed this gave enough leeway for unexpected changes because the EPN farm (in which `anan` was finally meant to be applied) only ever had about 250 nodes.

13.4 Practical Benchmark: Monitoring for EPN Farm

Previous run. In the previous *run* of the experiment ALICE [Aamodt et al., 2008], the equivalent of the **e**xtended **p**rocessing **n**odes (then called the high-level trigger "HLT") used a custom-designed monitoring based on Zabbix. Amongst others, the following reasons lead to the choice of another monitoring solution:

- Perceived lack of extensibility;
- insufficient performance for the anticipated higher data rates (in the previous run, the highest data rate was one measurement per minute which was increased to one measurement per ten seconds).

Modular stack. Therefore, the Work Package responsible for monitoring of the EPN farm decided to design a modular stack: Take the then-best solutions for each part of monitoring. The following parts and corresponding solutions have been identified:

- **measure data locally:** *telegraf*;
- **aggregate and store data centrally:** *influxdb*;
- **display data from a database:** *Grafana*.

This modular stack is also known as the "TIG stack". Had the decision been taken today, the last point might have been replaced by the two components chronograf and kapacitor (yielding the "TICK stack"), since all four tools are developed by the same group and therefore might interact more smoothly.

`anan` **for monitoring.** The component `anand` can be used similarly to telegraf, the component `anan` can take the role of influxdb. This was not possible without changes to `anan` because `anan` was designed for interactive command line usage and was unable to present Grafana the HTTP-based API it wants to query.

Requirements for Grafana. Grafana is used by everyone who is involved in the operation of the EPN farm. Each "dashboard" simultaneously shows about a dozen of "panels" of which each graphs the result of a query to e. g. `anan`. Therefore, `anan` has to choose one of the following solutions:

1. respond to queries very rapidly;
2. use multiple threads;
3. use an external program to answer queries.

Since solution (1) is naive and unrealistic, (2) was attempted. The report on this attempt will be very short since it ultimately failed. Solution (3) proved to be both more general, more reliable and easier than (1).

An *a priori* analysis might have found this because (3) means reuse of existing software, whereas (2) means introduction of complexity into a already somewhat tangled problem.

Failure at implementing Grafana's API. A few complications have been found (the more interesting are outlined briefly in the next few paragraphs), but the ultimate problem was generally bad performance for many simultaneous queries.

1. **JSON.** JSON is the JavaScript Object Notation [Bray, 2017]. It is the textual representation of structured objects in JavaScript and a commonly used notation for the exchange of arbitrarily structured objects over text-based protocols such as HTTP. Since Grafana shows its dashboard through a web browser and JavaScript runs in the web browser, using JSON for both query and response objects is the obvious choice. Parsing and creating JSON is easy in Lua; there are several libraries available for this. The problem is a slight mismatch between Lua's and JavaScript's data structures: Some (not very commonly appearing) structured objects are not representable as JSON, and some JSON objects are not easily represented as Lua object. (Example for the former: associative array with non-scalar keys; example for the latter: array with "holes".) Although it is exceedingly unlikely that either of these malformed objects appear in practice, there has to be code handling them. Naturally, this code is always brittle, since it is tested only rarely, but still taints the code base.

2. **API documentation.** Although Grafana's API is documented, the documentation did not fully explain which endpoints need to be implemented and how data is passed. Time-consuming reverse engineering revealed these details. Some parts remained mysterious. Luckily, they were not compulsory for a Grafana data source.

3. **Constructing strings.** The response object needs to be constructed as a concatenation of many small strings. The naive approach has quadratic runtime:

```
-- naive approach
local accumulator = ""
for i = 1, #pieces do
    -- join with ","
    accumulator = accumulator .. "," .. generate_piece(i)
end
print(accumulator)
```

This code has runtime $\mathcal{O}(|\texttt{pieces}|^2)$ because Lua's strings are *immutable*: Every concatenation needs to copy the entire previously constructed string. — A more efficient approach uses the fact that dynamic arrays in Lua cost only linear time:

```lua
-- linear-time approach
local accumulator = {}
for i = 1, #pieces do
    -- amortised over the loop, this costs O(1)
    accumulator[i] = generate_piece(i)
end
print(table.concat(accumulator, ","))
```

Although the runtime is $\mathcal{O}(|\texttt{pieces}|)$, this coding style adds complications wherever it is used.

4. **Passing data between threads.** It is well known that threading is another complicating factor. Lua does not bring its own libraries for threading. There are libraries which wrap POSIX threads in various ways. The library `luaproc` [Fernández, 2016, Skyrme et al., 2008] by the developers of Lua was chosen as it was found to be the only library (at that point in time) that could reliably use multiple cores and pass data between threads. It is based on *message passing*, i. e. copying all data from the sender's heap to the receiver's heap. An elaborate framework to do so efficiently (based on the principles mentioned in the previous point) was developed. Since it was finally decided not to use this framework, this thesis will not report more details.

   The performance was frustratingly weak: There was a constant overhead that made responses returning a few thousand points only a few seconds faster than responses returning a million points. Better algorithmic and implementation techniques (i. e. more complicated, therefore more likely buggy code) would certainly improve this behaviour. Still, the cost of these techniques was felt to be higher than the benefit. Therefore, approach (2) was decided to be a failure.

External program for Grafana's API. The plugin `sqlite-datasource` for Grafana allows the results of SQL queries against a *SQLite* [Owens and Allen, 2010] database to be visualised with Grafana. The plugin runs as a separate process. Grafana deals with the complexity of calling multiple instances of the plugin's process when many queries are processed. The plugin implements the relevant parts of Grafana's API. The only parts that need to be filled in are:

1. Store monitoring data in an SQLite database.

2. Formulate the necessary queries in SQL (instead of the previously used query languages InfluxQL and Flux).

The second part is well-understood; there are numerous high-quality manuals and tutorials for SQLite's dialect of SQL. Therefore, only a few interesting aspects for the first part shall be recalled.

Table names. Sensors have arbitrary names. The default names are `anon-1`, `anon-2` etc. SQLite permits identifiers containing almost arbitrary characters using the syntax `[whitespace and-hyphens]`. This would fail as soon as a sensor name ever contained unbalanced square brackets, e. g. the name `"1.]"`. Since this is absurdly unlikely, no attempt was made to map such names to a legal format. Instead, an error with an explanatory statement is logged.

Column values. Sensor results are arbitrary Lua objects. A look into a list of all sensors ever deployed showed that sensor results are tables with strings (or numeric values) as keys. The values are sometimes non-scalar but easily mapped to scalars. This mapping is best performed by conversion to JSON because SQLite has native support for JSON. As mentioned above, care has to be taken to make sure this conversion is successful. It therefore was decided to change all sensors to only send string or numerical results.

Missing/new columns. Theoretically, a sensor might change the set of keys it sends over time. It is possible to change an SQL table's schema. This operation, however, tends to be costly. If a sensor starts sending new keys it did not use in the initial message, these keys will be silently ignored. An examination of all sensors found that this could currently never happen. — On the other hand, missing columns will lead to the (absence of a) value `nil`.

Indices. An index on the pairs (hostname, timestamp) greatly increases the typical query's performance. Statements to create these indices were added automatically.

Summary. Monitoring and debugging are different tasks. It is not obvious that a tool for debugging is suitable for monitoring. With a small extension to harness the power of SQLite, `anan` proved to be suitable, if not ideal, as a monitoring tool. It remains to be shown if there are synergies between monitoring and debugging that can be used to improve the performance, scope or usability of either.

13.5 Practical Benchmark: A Tiny Case Study

Overview. The following report is somewhat artificial, although it abstracts an actual case of `anan`'s application. It may seem that the required setup until `anan` could be used productively is too high. This is certainly true; nevertheless, without `anan`, even this much setup might have not sufficed to find the root cause.

The solution. For a more intelligible exposition, the solution of the problem will be presented before the setup and search towards the solution. Of course, the reader is urged to follow `anan`'s user as unknowingly as the user was while he sought the solution. — A program on one node (for argument's sake, say it is `epn100`) produces minimal (disk and network) IO, sometimes uses a lot of CPU and sends a few kilobytes of data through a socket. In short, it behaves like

```
cat /dev/random | nc epn200 12345
```

(Reading from `/dev/random` is slow and CPU-intensive. The command `nc <host> <port>` opens a TCP socket to `host` at `port`. — On more modern releases of Linux, `/dev/random` is as fast as the non-blocking device `/dev/urandom`.)

On the other machine (in this example, it is called `epn200`), there is a corresponding process that collects this data without any meaningful action, as in:

```
nc -l -p 12345 > /dev/null
```

(The switch `-l` puts `nc` into server mode, it listens at port `12345` and stores the data in the pseudo-file `/dev/null` which discards everything.)

Setup.

1. **Finding sockets.** The following pseudo-files are relevant:

   - `/proc/net/tcp`
   - `/proc/net/udp`
   - `/proc/net/tcp6`
   - `/proc/net/udp6`

   These four files list all TCP/UDP sockets using either IPv4 or IPv6. (The latter might be less important since the EPN farm shies away

from the usage of IPv6.) — Each socket is identified by its local/remote host/port and the protocol UDP or TCP (and the IP version). In Linux, every socket is internally identified via its *inode number*. This number is also given in these files. The process id (*pid*), however, is not given. Another (small, but annoying) complication is that the IPs and ports are given in hexadecimal; the addresses' octets are in little endian, whereas virtually everywhere else big endian is applied. This only adds a few lines of parsing code. — A sensor was constructed which lists all TCP or UDP sockets with corresponding addresses and inode numbers.

2. **Finding the corresponding process.** Every Linux process has a pid. Given the pid, symbolic links to all file descriptors being held by the process can be found in the directory `/proc/<pid>/fd/` on the pseudo file system `proc(5)`. Each of these files can be mapped to its inode number via the system call `readlink(2)`. — A sensor was constructed which returns a map of all pids to the inode numbers of the corresponding open sockets.

3. **Finding additional information about processes.** The pseudo-file-system `proc(5)` offers much more information than necessary for this debugging task. For this task, the file `/proc/<pid>/stat` is most interesting: It contains about 50 different tab-separated fields, among them

   - field no. 14: **utime**, the amount of *clock ticks* the process has been scheduled in user mode;
   - field no. 15: **stime**, the amount of clock ticks the process has been scheduled in kernel mode (**s**ystem mode).

   High values hint to processes that do a lot.

   Another interesting file is `/proc/pid/cmdline`. This `'\0'`-separated file contains the command line used to start the program with the given pid. It helps identifying the program that runs under this pid. The program may decide to change the value presented under this file and thereby "lie". If a system was ever suspicious of having programs that do this, `/proc/pid/exe` — a symbolic link to the program's executable file — might be more helpful.

Sensors reporting these three pieces of information have been deployed. Crucially, the Linux kernel presents all information in files ("everything is

a file") in formats which `anan`'s sensors can extract easily. — The data was correlated using SQL statements joining all three pieces of information. Interactive experimentation led to a query showing the exact level of detail, amongst others adding the following aspects:

- only consider connections inside the EPN (based on the address prefix);
- don't consider connections that belong to executables called `anand` or `telegraf` because they are known not to misbehave in this way;
- only consider programs which have a somewhat high `stime` — the value was found by eyeballing — because the incriminated program was assumed to spend a lot of CPU time of which a significant part was likely to be related to system calls.

This query found a list of processes containing the searched-for program. Although at no point, a tool like `anan` was required, it probably still helped to finish the search more quickly.

13.6 Application Note: Porting `anan` to an Embedded System

Introduction. For organisational reasons, an embedded system (a board centred around a TMS570LC53 Hercules™ Microcontroller Based on the ARM® Cortex®-R Core) could not be used for its intended purpose. It became a "solution looking for a problem". This problem was identified: debug custom hardware in the context of particle physics. The board was to be equipped with rich measuring electronics and connected to the system under consideration. A few boards have been built already. Since they were relatively small, they could be put in any relevant place. Communication would happen via Ethernet, although WiFi has been considered. The device was to be deployed in an environment with increased radiation which rules out WiFi.

This work was performed in cooperation with José Antonio Lucio Martínez.

Choice of operating system. Before the project was stopped (again, for organisational reasons), `anand` was cross-compiled for the above-mentioned microcontroller. If `anand` was required to run without an operating system, major restructuring would be needed. Therefore, an operating system

was run on the microcontroller. Since there was previous experience with RTEMS 4.1, that operating system was chosen.

Cross-compiling `anan`. Any `configure` script constructed by autotools should automatically support cross-compiling, provided the setup for autotools was done correctly. This requires precise specification which tool should run on the host (as part of the compilation) or on the target (as resulting artefact of the compilation). The build system used for `anan` does, in fact, require some parts to be built for the host and executed there to create some of the build artefacts.

Another complication was the incomplete support of POSIX' symbols by RTEMS' `libc` and other system libraries. Most of these symbols were probably not relevant because `anan` was not supposed to be used to debug software issues. Therefore, most of the support for POSIX APIs could have been chucked. For completeness' sake, bindings for most of RTEMS' APIs have been included.

Summary. It could be ascertained that `anan` could run on that board. The project was stopped before any actual measurements could be performed by `anan`. Nevertheless, this shows that `anan`'s clean and minimalist design makes porting it to new architectures relatively straight-forward. (As an aside, the author found that both `anan` and `anand` run unmodified on commodity Android smartphones equipped with a C compiler. The use case remains unclear.)

Chapter 14

Usability and other "Soft" Criteria

14.1 General Observations

"Eat your own dog food." The author attempted to use `anan` for as many different tasks as possible. (Another attempt aimed at managing configuration files with `anan` never left the planning stage for various non-technical reasons.) This approach was meant to improve the "soft" aspects of `anan`'s performance, chiefly usability and fitness for as many purposes as possible.

Smooth interactions. The user interface of `anan` is a classic *REPL* (read-execute-print loop): A prompt is presented, input should be given, the resulting value of this input is shown. REPLs gain in usability by adding some features. The following features have been planned, some of them implemented.

- **Line Editing.** Following the now standard paradigm of direct manipulation [Hutchins et al., 1985], the command line prompt is edited interactively. Without the usage of GNU readline [Ramey, 2015] or a similar library, only the most elementary line editing is available: Adding characters at the end of the input string; deleting the last characters; delete the whole line. — The usage of `readline` eased the implementation of automatic completion.

- **Syntax highlighting.** [Hansen, 1971] This has not been implemented. It was planned to highlight keywords, strings, string/number literals and nesting levels. The plan was abandoned for the following reasons:

  - Not much usability was added.

- Syntax highlighting interferes with line editing (`readline` expects to be in full control of the terminal).
- An almost full lexical analysis of the input string is needed to highlight it correctly.
- A semi-correct highlighting is probably worse than none.

Altogether, the possible gains did not justify the amount of added code.

- **Automatic completion.** This feature is so ubiquitous in text-based interfaces that it needs no explanation. It has two advantages:
  - Fewer keystrokes needed.
  - Better discoverability: If a command `x = complex_datum()` was executed, typing `x.`, followed by a tab, will show the subfields of `x`. Thus, even deeply nested hierarchies can be traversed easily.

  This feature required somewhat less code than the attempt at syntax highlighting (because it only supports a tiny subset of Lua's syntax). The increase in complexity was greatly set off by the increase in usability.

- **Pretty-printing data.** Using `anan` often involves the inspection of raw sensor responses which is much easier when the datum is shown in a "pretty" (not necessarily aesthetically pleasing, but systematically indented) way. This problem is well-understood (e. g. [Hughes, 1995]) and there are software libraries available. Therefore, the benefit of pretty printing came with very little cost.

- **"Online" help.** A system with online help has a facility to show help for every object the user can interact with. For `anan`, a partial approach has been taken: Variables of types `string`, `number`, `table`, `nil` and `boolean` are easily inspected by pretty-printing and/or automatic completion. Conversely, `thread` and `userdata` values should never be seen by the user (with the possible exception of file descriptors) and therefore do not require any elucidation. The remaining type, `function`, was given an implementation of the meta-function `__tostring` which returns the function's documentation. Since this did not add discoverability and was a burden to maintain, this feature was removed.

A similar effect could have been achieved by the usage of `readline`: Additionally to presenting several possible completions, short help notices (such as argument names and types, possibly with additional explanations) can be shown. No attempt was made to implement this for the same reasons as in the previous paragraph.

- **Result variables.** Sometimes, the user gives a command which performs a lengthy calculation and the result is a complex datum. But once the datum is pretty-printed, it is lost, although the user might want to continue using it for further calculation or inspection. To prevent this, there are three "magical" variables `_`, `__` and `___` that always store the last three results ([Forster, 1996]; [Steele, 1990, symbols ***, +++, ///]). Although this facility is used rarely, its costs are negligible.

Debugging the GWC. The second "synthetic benchmark" was performed using an instance of OpenStack that was dubbed GWC. As reported in chapter 13.3, the results were inconclusive but slightly positive. Why was `anan` not used to debug the benchmark? Of course, this has been tried. The main obstacle was lack of access to critical information: The benchmark executed many thousands of containers running on dozens of virtual machines. The containers were under the GWC's user's full control. This did not hold for the virtual machines: The virtualisation technology KVM [Goto, 2011] has been employed using the user interface presented by OpenStack. This gave only very limited possibilities to inspect the virtual machines (besides everything the virtual machine can measure about itself). Therefore, `anan` did not have access to crucial information. For organisational reasons, this access could not be granted in any meaningful way. Nevertheless, measurements found hints to a possible root cause: Possibly, either storage itself or the storage system's network connection was too slow. Later communication with the GWC's administration confirmed that this issue was known, although it could not be proven conclusively because the GWC went into a phase of restructuring.

Summary. Almost every cluster-related issue can be debugged with `anan`. Any questions can be asked and answered, but none easily. The next subsection will present some educated guesses on possible reasons and corresponding solutions.

14.2 Reasons and Solutions

1. **Jack of all trades.** Maybe, `anan` is trying to do *too much*: A universal tool is bound to be universally weak in every discipline covered. Is this also true in `anan`'s case?

   Consider the following tools which are employed in the wider field of debugging:

   - **strace** (trace system calls);
   - **tcpdump** (dump network communication);
   - **valgrind** (debug memory issues);
   - **gprof** (software performance profiler).

   All of these tools are highly specialised and contain very sophisticated algorithms and data structures optimised for the exact use case. None of them perform tasks that `anan` could do in any meaningful way. In fact, if this was desired, the necessary plumbing could be easily added to `anan` so that it could tap into these tools' facilities. The programs `anan` and `anand` would not require any change; only corresponding sensors need to be deployed.

   Quite the opposite: `anan` does not try to do anything that another tool could do better. Instead, it tries to be the glue combining interesting sources of information.

2. **Too small standard library.** Maybe, `anan` can do *too little*: The standard library that ships with `anan` does not allow the user to do much. Therefore, he might be stuck reinventing the wheel.

   This is certainly true in some sense because `anan` does not even have a "standard library". This is not only due to the fact that `anan` is coded in Lua (a language "without batteries included"). On the other hand, the full abilities of POSIX' [spe, 2008] C library are available to all sensors. This is complemented by all tools of the debugged systems' user land. And of course, `anan` can be extended at built time by any library that can be linked to `anan`'s binary. Theoretically, arbitrary C libraries can be loaded and arbitrary C functions called from Lua. This facility has not been tested extensively, though, and is believed to be superfluous for `anan`'s use case (although it is still available for the rare case). Only a few combinations of operating system, `libc`, and compiler/linker support generating static objects

(anan always builds as a static object) which can load dynamic objects. The most popular case of Linux with glibc and gcc or llvm has been reported to work, though.

3. **Too low level.** Maybe, anan cannot provide a sufficiently high level of abstraction. The user has to fight with repetitive details which are irrelevant to the current problem. Of course, this problem is related to the previous point (too small standard library). Therefore, the response is similar to the previous response: Sensors can use all facilities that can be loaded by anan's run control file. This can contain all facilities mentioned in the previous point. Even programming paradigms, such as functional and object oriented programming, can be added as libraries [Ierusalimschy, 2016, chapter 20–21]. With this infrastructure, any desired workflow can be implemented.

   Alternatively: insufficiently opinionated. The level is not too low, all required facilities are reasonably easily accessible, there is not anything distracting the user, but he is still "unhappy": The tool anan does not gently lead him a specific way; it does not present sensible "best practices". There is no beaten path to follow. — This criticism is absolutely true. The reason is that there are no best practices yet. Even the author never found that there is a standard approach he should be following.

4. **Problem exists between keyboard and chair.** This is no serious criticism and should not be rewarded with an answer.

Summary. This remains broadly unclear. A set of answers may include non-technical issues, such as insufficient documentation; organisational issues; the complexity of questions that were meant to be resolved; random personal preferences etc.. Only future experience with anan and with similar tools yet to be developed can give definitive answers.

14.3 Monitoring and Debugging

Overview. It has been shown that anan is a possible makeshift monitoring tool. This does not mean that debugging clusters is similar to monitoring them. This fallacy can be likened to the following one: C++, a programming language considered relatively high-level, is just a super-set of assembly language. On a purely technical level, this is correct: Any assembly program can be "converted" to C++ by adding a few lines of boiler

plate code (the main function and the `__asm__` statement). By this, however, it does not become a high-level program. Similarly, abusing `anan` as a monitoring tool does not show anything about these two related tasks.

Universal or high-level tooling. Everything can be coded in assembly language. In fact, at a certain stage during compilation, every statement of a C++ program is translated into assembly language. Accordingly, every issue in a distributed system can be detected by properly set up monitoring and debugged by logging in at every machine and measuring manually. The efficiency of these two approaches is not comparable, though. In this situation, `anan` can at least be used to configure monitoring such that the critical variables are shown. On the scale of universal tools (cf. assembly language) to high-level tools (cf. C++), `anan` tries to tend to the latter side.

Chapter 15

Interlude: The notion of simplicity

> A complex system that works is invariably found to have evolved from a simple system that worked.
>
> —John Gall

Overview. A core result (mentioned in many place in this thesis) is that *simplicity* is important. It is a truism that complexity for complexity's sake and the "creeping features disease" do not go well with reliable and efficient systems, be it in software or any other field of engineering. This observation, however, hints to only a small part of the simplicity whose importance was found. That simplicity is on a conceptual level. Since this thesis does not primarily deal with philosophical questions, this interlude is kept as brief as possible.

Definition. Philosophy has been knowing the notion of simplicity for a long time:

> We may assume the superiority *ceteris paribus* of the demonstration which derives from fewer postulates or hypotheses [McKeon et al., 2009, p. 150].

This is commonly (probably incorrectly) known as Ockham's Razor. However, this idea is too imprecise to serve as a definition.

Simple things with complex behaviour. One should not be fooled into thinking that simple objects show simple behaviour. It has been long observed [Wolfram, 2002, p. 27 about "Rule 30"] that some obviously simple algorithms (only a few lines of low-level code) show complex behaviour.

This observation is central for the field of cryptography. Consider the following algorithm [Paul and Maitra, 2011](where `i`, `j`, `S[0]`, ..., `S[255]` are unsigned byte values that have been initialised previously; `S` is a permutation of the set $\{0, \ldots, 255\}$ of all bytes):

```
uint8_t tmp;
j += S[++i];
tmp = S[i]; S[i] = S[j]; S[j] = tmp;
printf("%u\n", S[i]+S[j]);
```

The output of this code — after repeated execution — is highly random. In fact, this algorithm has long been used as the core of the stream cipher RC4.

Admittedly, most technology is designed to be testable and therefore not to show erratic and random behaviour. Therefore, "simplicity" as in Rule 30 and RC4 can be disregarded.

Kolmogorov complexity. In algorithmic complexity theory, the Kolmogorov complexity of a string s is the size of the smallest universal Turing machine that outputs s and terminates. Since all universal Turing machines can simulate each other, the precise choice of Turing machine model and notion of size is immaterial. The Kolmogorov complexity of a string is incomputable. Only upper bounds can be given. For practical purposes, [Ziv and Lempel, 1977] entropy-based compression methods often fulfil this purpose. Even this class of methods does not yield a practical method to assess the complexity of anything but a string — ideas such as software architectures or programming languages cannot be compared.

A subjective approach. Simplicity of structured objects such as the two examples mentioned in the previous paragraph cannot be defined in a *reductive* way. Instead, the following tentative definition is proposed: If the whole object under consideration has one simple idea (or a few) at its core, then even loading it with almost arbitrarily much complexity cannot negate the simple basis. This definition requires a few clarifications:

- **Self-referential.** What is a "simple idea"? This can be quantified via Kolmogorov complexity because ideas are typically conveyed as text (i. e. strings). An idea is simple, if it can be described by a very short human-readable text.
- **What is too much complexity?** The definition claims that there is a limit of added complexity which can drown the original simple core. To a certain degree, this is subjective. But the following consideration might clarify: If the core idea stops being the core, it does

not really matter how many concepts are added — the simple core is gone. Once most of the interaction with the object under consideration happens through tangential ideas, there is not even a simple core left.

Consider the following example: The hallmark of relational database systems is working with operations on relations such as projections, cross products and joins. But in certain settings, all interactions with a database happen through "stored procedures" which abstract the relational operations away. Even though not necessarily much complexity is added, the simple core of relational logic is lost (although possibly, another simple core can be identified).

- **Core and tangential ideas.** An idea is a core idea if the object under consideration stops being that object once it is removed. The object cannot be explained without it. Often, it cannot perform most functions it usually performs. — The above-mentioned example of stored procedures also applies here: If the relational logic was removed from the database, then it could not operate at all.

Two examples from the realm of programming languages should illustrate this definition. Both languages and corresponding ecosystems are popularly assumed to be somewhat complicated.

15.1 Forth: Two Stacks, Tiny Functions, Direct Machine Access

Introduction. Forth is an old programming language [Biancuzzi et al., 2009, Chapter 4]. It stems from an era when every byte and every clock cycle was precious. Accordingly, although it wanted to be called "fourth" (as in "fourth generation language"), the system it ran on only allowed five-letter executable names, yielding the name FORTH (originally in capital letters).

Tiny functions and the first stack. The most fundamental idea is to use tiny functions. To simplify communication between functions, the (*data*) stack through which function arguments are normally passed is exposed to the user. Thus, a function does not need to (and, in fact, cannot) declare how many arguments it receives. Arguments cannot be given names or types. A function (in Forth parlance, a *word*) takes as many arguments from the stack as it wants. It may also return zero or more arguments. This almost automatically leads to the usage of reverse Polish notation:

```
(3+4)*(7-8)
```

is spelt as

```
3 4 + 7 8 - *
```

No precedence rules or parentheses are required. — A function consisting of a dozen words is already considered long. This leads to a "horizontal" style where most functions occupy one (short) line of code.

The return stack. Conventionally, control flow is implemented via the same stack: In C and C++, usually the return address is pushed on the stack, followed by the function's argument(s). In Forth, there is a separate *return stack* that only holds return addresses. There are facilities to copy items between the two stacks. This is commonly only done to implement special control flow words, e. g. exception ("condition") handling or sophisticated backtracking.

Direct machine access. The return stack is already a prime example of direct machine access. Other examples include:

- **Full multiplication.** On most CPUs that support multiplication, the product of two x-bit registers is stored in a double register with a total size of $2 \cdot x$ bits. In Forth, a special multiplication operator is available which returns both registers. (Interestingly, historical implementations did not give access to the overflow or underflow flag of addition or subtraction operations.)

- **Inline assembly.** Virtually all classical implementations contained an inline assembler. It was not uncommon to add ad-hoc words implemented in assembly language to use otherwise inaccessible features.

- **Block files.** On implementations with non-volatile mass storage, the following API was chosen for file IO: The disk consists of numbered *blocks*, each 1024 bytes in size. A block can be addressed by its number. This operation returns a pointer to a buffer with the contents of that block. After that buffer has been modified, the system needs to be informed of the modification. The buffer is written to disk either upon a request to do so or whenever the system runs out of buffers. — There is a convention that the 1024 bytes are to be understood as 16 lines of 64 bytes each, a size somewhat smaller than a contemporary screen could show.

Summary. This outline can only give a glimpse at all the ways Forth tries to give simple but direct access to the machine's full abilities. The simple core is given in this short section's title. But really, the core is even simpler: Whenever a compromise between ease of usability and ease of implementation is needed, err on the latter side. Forth is easily implemented (there is a multitude of hobbyist implementations) and somewhat easily used. Its semantics, however, is very complex, especially considering the complex interaction between control and data flow once both stacks are exercised.

15.2 Lisp: Linked Lists, Term Rewriting, Homoiconicity

Introduction. Lisp is even older than Forth [McCarthy, 1960]. This language was conceived as a tool for symbolic artificial intelligence. Therefore, Lisp has great support for discrete data structures such as trees, graphs and lists — hence the name **lis**t **p**rocessor.

Linked lists. The oldest and most fundamental data structure is the singly-linked list, represented as a pair of (1) the first element; (2) a pointer to the remainder of the list. For obscure historical reasons, these two pointers are called the `CAR` and the `CDR` of a `CONS` pair. E. g. the "proper" list $(1, 2, 3, 42)$ could be represented by the lisp code `(LIST 1 2 3 42)`. Conventionally, a `CONS` pair (a, b) is spelt `(a.b)`. The above-mentioned list is a short form for the `CONS` pairs given by `(1.(2.(3.(42.NIL))))`; here, `NIL` is a marker for the end of a list.

Although more data structures such as arrays, hash tables and records have been added, lists are the most flexible (though not always most efficient) data structure. Many internal structures are represented as lists and are supposed to be manipulated as such.

Term rewriting. In fact, the execution of a lisp program can be understood in terms of rewriting a list, namely the program's source:

- The value of any atomic (i. e. non-list) value is the value itself.
- The value of a list `(f x y z ...)` is given by the following procedure:
    1. Evaluate `f`, `x`, `y`, `z`, ...
    2. Call function `f` with arguments `x`, `y`, `z`, ... (Of course, `f` should evaluate to a function.)

Based on this, an interesting *functional* language could be built. On top of this, Lisp has a rich set of non-functional features such as global variables, exception handling and mutable data structures as expected for a typical imperative programming language.

Homoiconicity. Lisp is an example (maybe the oldest in the world of computing) of homoiconicity: The program's code is a first-class value of the language (namely, a list). Therefore, programs can be constructed and manipulated algorithmically with ease: Code writes code. This is traditionally implemented by *macros*, Lisp code which alters the source at compile time. Whole new programming paradigms such as object oriented or aspect oriented programming can be added as sets of macros. No new meta-programming language is required, and all facilities for manipulating data structures are available.

Common Lisp. This standard [Steele, 1990] tried to standardise (almost) every feature available in any of then's major Lisp implementations. At the same time, it added all non-controversial facilities which were planned for then's next generation implementations. Altogether, it was seen as a huge programming language (although it is small in comparison to today's languages). — More features not conceivable at that time have been added to modern Lisps, such as networking, graphical user interfaces, threading, transactional memory, ...

Summary. This vignette could hardly even scratch the surface of Common Lisp and other branches of Lisp's language tree (notably, Scheme was completely ignored [Clinger and Rees, 1991]). Still, the case could be made for Lisp's simple core (consisting of the section title's three ideas) and the many useful additions. It is notoriously hard to implement Common Lisp in a fully conforming way; on the other hand, even the semantics of a full Common Lisp are relatively straight forward.

15.3 Simplicity in `anan`

Introduction. This section wants to apply the principle to `anan`. Its three core principles are

- `awk` sensors;
- Lua as extension language for sensors;
- relational queries between sensor results.

Unclear simplicity. All of these can be viewed as "large" technologies:

- The POSIX standard [spe, 2008] covers `awk` on about thirty pages — which do not include many other features such as regular expressions, localisation etc. discussed in other parts of the standard.
- The Lua reference manual is about 50 pages long [de Figueiredo et al., 2017]. It is mostly self-contained and relies only on the specification of the programming languages C and C++.
- The standard for SQL 92 [mel, 1992] runs over 600 pages (and of course, SQLite implements much more than the standard defines, although a small number of features is missing).

Obviously, page counts are arbitrary and depend on formatting and stylistic issues; additionally, standard documents tend to be more bureaucratic and therefore more verbose.

Not full `awk`. In `anan`, however, only the core idea of `awk` is used and implemented: Parametrise using variables; open files; read (generalised) lines, split into (generalised) fields; execute code when line matches pattern; gather results. These few words translate into a few hundred lines of code. Of course, this by far does not constitute a full implementation of `awk`.

Lua is small. The draft standard for C89 [c89, 1992] is about three times as large as the Lua reference manual when counting "pages" the same way. It is certainly written in a more pedantic style as it was submitted to ISO's standardisation process. But since C is known as a small language, Lua should be understood to be even smaller.

Not much SQL is used. In practice, very simple `SELECT` statements and rarely a few `NATURAL JOIN`s have been used. Theoretically, the full power of SQLite is available. Very little is necessary since it duplicates many of Lua's or `awk`'s facilities.

Summary. The notion of simplicity, based on the ideas of Kolmogorov complexity, can be successfully applied to software artefacts. From the right point of view, most apparent complexity becomes tangential, whereas the simple core can be seen clearly. This can be made somewhat plausible for the programming languages Forth and Lisp; a similar result has been attempted for `anan`. Since simplicity is only a tangent in this thesis, many questions remain, amongst them:

- Can this Kolmogorov-like complexity be quantified numerically?
- Specifically, can objects be ordered systematically by complexity?

- Can the root cause of complexity be found and fixed? Is this even required?
- No negative examples (i. e. seemingly simple, really complex) objects have been studied. The only (hardly relevant) exception is "Rule 30" and RC4 on p. 79. Without this, the proposed idea cannot be validated fully.

Chapter 16

Summary and Future Work

Introduction. The qualities of `anan` have been assessed, both using synthetic and practical benchmarks and non-functional (usability and similar) criteria. In this section, these results shall be used to assess the need for future work. Note that there could be two reasons why a certain aspect does not require future work:

1. It has been dealt with in a satisfying way. (*positive result*)
2. The aspect is not needed for `anan`'s successor. (*negative result*)

Naturally, previous results will necessarily be restated briefly.

16.1 Is `anan` Successful?

Benchmarks. The synthetic benchmarks hint and the practical benchmarks show that `anan` can be used in somewhat large data centres. Multithreading might be a useful addition to make sure `anan` can be used productively with several thousand systems under consideration. This, however, is a relatively straightforward addition. Other than this, this aspect is fully satisfactory.

Usability. As mentioned above, `anan` has been equipped with most of the amenities typical command line tools offer. This part of usability about as good as it can get for a command line tool. A graphical user interface might help usability, but was out of scope. Higher level tooling (large application-specific libraries of well-parametrised sensors; examples of meaningful SQL queries; tutorial-like examples etc.) would have probably improved `anan`'s ease of use greatly. This is, however, partially

out of scope, although during normal usage, such examples arise naturally. Therefore, this aspect might require some attention.

Actually found bugs. As mentioned above, besides the tiny case study in chapter 13.4, no real bugs have been found. That still is no negative result, be it only for the reason that the facilities used in the case study have been available only for a short time. The case study has certainly shown that `anan` can in principle be used to aid in finding even well hidden bugs, although the effort may still be considerable.

16.2 Is Cloud Debugging Successful?

Introduction. This thesis can only study a single instance of cloud debugging, `anan`. Perhaps, this instance was not representative of the (hypothetical) class of cloud debuggers?

Minimum requirements. The mere requirement of being a cloud debugger forces a few properties, among them:

- interactive (although not necessarily command line) use;
- client-server architecture;
- servers on each system under consideration;
- light-weight data transport layer;
- extensibility.

Together, many design choices described in chapter 11 are dictated by these requirements. Although many are not, any tool that wants to be a cloud debugger has to be similar to `anan`. Therefore, a result about `anan` almost automatically translates into a result about any cloud debugger of a comparable level of sophistication. Naturally, a tool into which much more effort has been invested cannot be compared to a relatively young tool such as `anan`.

Summary. A cloud debugger is clearly a success story, be it only as a do-it-all tool or as "better" monitoring tool. Since debugging is inherently a finicky task (and the more so for large systems with many moving parts), even the slightest improvement is helpful. But even the best tool (which `anan` is not, as of yet) can always be improved.

Bibliography

[rec, 1988] (1988). Specification of abstract syntax notation one (ASN. 1).

[mel, 1992] (1992). ISO/IEC 9075:1992: Information technology–database languages–SQL.

[c89, 1992] (1992). ISO/IEC 9899:1990: Programming languages—C.

[spe, 2008] (2008). Draft standard for information technology—portable operating system interface (POSIX®) draft technical standard: Base specifications, issue 7.

[int, 2013] (2013). Intelligent platform management interface specifications second generation [ol].

[Aamodt et al., 2008] Aamodt, K., Quintana, A. A., Achenbach, R., Acounis, S., Adamová, D., Adler, C., Aggarwal, M., Agnese, F., Rinella, G. A., Ahammed, Z., et al. (2008). The alice experiment at the cern lhc. *Journal of Instrumentation*, 3(08):S08002.

[Aceto et al., 2013] Aceto, G., Botta, A., De Donato, W., and Pescapè, A. (2013). Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115.

[Adler and Kebschull, 2020] Adler, A. and Kebschull, U. (2020). Anan—analyse and navigate: Debugging compute clusters with techniques from functional programming and text stream processing. In *EPJ Web of Conferences*, volume 245, page 01041. EDP Sciences.

[Agans, 2002] Agans, D. J. (2002). *Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems*. Amacom.

[Aho et al., 1979] Aho, A. V., Kernighan, B. W., and Weinberger, P. J. (1979). Awk—a pattern scanning and processing language. *Software: Practice and Experience*, 9(4):267–279.

[Anderson et al., 1990] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1990). LAPACK: A portable linear algebra library for high-performance computers.

[Angermann et al., 2020] Angermann, A., Beuschel, M., Rau, M., and Wohlfarth, U. (2020). *Matlab–simulink–stateflow*. De Gruyter Oldenbourg.

[Barry, 1992] Barry, P. (1992). Abstract syntax notation-one (ASN.1). In *IEE Tutorial Colloquium on Formal Methods and Notations Applicable to Telecommunications*, pages 2–1. IET.

[Becker et al., 1995] Becker, D. J., Sterling, T., Savarese, D., Dorband, J. E., Ranawak, U. A., and Packer, C. V. (1995). Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, volume 95, pages 11–14.

[Bernstein et al., 2014] Bernstein, D. J., Van Gastel, B., Janssen, W., Lange, T., Schwabe, P., and Smetsers, S. (2014). Tweetnacl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*, pages 64–83. Springer.

[Biancuzzi et al., 2009] Biancuzzi, F. et al. (2009). *Masterminds of programming: Conversations with the creators of major programming languages*. O'Reilly Media, Inc.

[Binnie, 2016] Binnie, C. (2016). Supercharged systemd. In *Practical Linux Topics*, pages 21–32. Springer.

[Bishop, 2006] Bishop, C. M. (2006). Pattern recognition. *Machine learning*, 128(9).

[Bray, 2017] Bray, T. (2017). RFC 8259: The javascript object notation (JSON) data interchange format.

[Bray et al., 2006] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2006). Extensible markup language (XML) 1.1.

[Cadar et al., 2008] Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224.

[Clark et al., 1999] Clark, J. et al. (1999). XSL transformations (XSLT). *World Wide Web Consortium (W3C).*, 103.

[Clinger and Rees, 1991] Clinger, W. and Rees, J. (1991). Revised report on the algorithmic language scheme.

[Codd, 1983] Codd, E. F. (1983). A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69.

[Collet and Kucherawy, 2018] Collet, Y. and Kucherawy, M. (2018). RFC 8478: Zstandard compression and the application/zstd media type.

[Cook and McKenzie, 1987] Cook, S. A. and McKenzie, P. (1987). Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8(3):385–394.

[de Figueiredo et al., 2017] de Figueiredo, L., Celes, W., et al. (2017). Lua 5.3 reference manual.

[Despain, 1979] Despain, A. M. (1979). Very fast fourier transform algorithms hardware for implementation. *IEEE Transactions on Computers*, 28(05):333–341.

[Dijkstra, 1968] Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148.

[Duff et al., 2002] Duff, I. S., Heroux, M. A., and Pozo, R. (2002). An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267.

[Fatema et al., 2014] Fatema, K., Emeakaroha, V. C., Healy, P. D., Morrison, J. P., and Lynn, T. (2014). A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 74(10):2918–2933.

[Fernández, 2016] Fernández, L. M. (2016). Concurrent programming in lua—revisiting the luaproc library.

[Ferraiolo et al., 2000] Ferraiolo, J., Jun, F., and Jackson, D. (2000). *Scalable vector graphics (SVG) 1.0 specification*. iUniverseo Bloomington.

[Forster, 1996] Forster, O. (1996). *Algorithmische Zahlentheorie*. Springer.

[Frigo and Johnson, 1998] Frigo, M. and Johnson, S. G. (1998). FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE.

[Gailly and Adler, 2004] Gailly, J.-l. and Adler, M. (2004). Zlib compression library.

[Gillmor, 1987] Gillmor, C. S. (1987). Memoirs of a computer pioneer.

[Goto, 2011] Goto, Y. (2011). Kernel-based virtual machine technology. *Fujitsu Scientific and Technical Journal*, 47(3):362–368.

[Günther, 2020] Günther, N. (2020). Evaluierung des Cluster Monitoring- und Debugging-Tools anan unter Berücksichtigung von CERN-Anforderungen. M. S. thesis, Goethe-Universität Frankfurt.

[Hager and Wellein, 2010] Hager, G. and Wellein, G. (2010). *Introduction to high performance computing for scientists and engineers*. CRC Press.

[Hansen, 1971] Hansen, W. J. (1971). Creation of hierarchic text with a computer display. Technical report, Argonne National Lab., Ill.

[Härder and Reuter, 1994] Härder, T. and Reuter, A. (1994). Principles of transaction-oriented database recovery. In *Readings in database systems (2nd ed.)*, pages 227–242.

[Herder et al., 2006] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. (2006). Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89.

[Hochstein and Moser, 2017] Hochstein, L. and Moser, R. (2017). *Ansible: Up and Running: Automating configuration management and deployment the easy way*. O'Reilly Media, Inc.

[Hughes, 1995] Hughes, J. (1995). The design of a pretty-printing library. In *International School on Advanced Functional Programming*, pages 53–96. Springer.

[Hutchins et al., 1985] Hutchins, E. L., Hollan, J. D., and Norman, D. A. (1985). Direct manipulation interfaces. *Human–computer interaction*, 1(4):311–338.

[Ierusalimschy, 2016] Ierusalimschy, R. (2016). *Programming in Lua, Fourth Edition*. Lua.Org.

[Karlesky et al., 2007] Karlesky, M., Williams, G., Bereza, W., and Fletcher, M. (2007). Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In *Proc. Emb. Systems Conf, CA, USA*, pages 1518–1532.

[King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

[Lieberman, 1997] Lieberman, H. (1997). The debugging scandal and what to do about it. *Communications of the ACM*, 40(4):26–30.

[Mathews, 2004] Mathews, B. S. (2004). Internet resources: Gray literature: Resources for locating unpublished research. *College & Research Libraries News*, 65(3):125–129.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195.

[McKeon et al., 2009] McKeon, R. et al. (2009). *The basic works of Aristotle*. Modern Library.

[Metzger, 2004] Metzger, R. C. (2004). *Debugging by thinking: A multidisciplinary approach*. Digital Press.

[Milojkovic et al., 2017] Milojkovic, N., Ghafari, M., and Nierstrasz, O. (2017). It's duck (typing) season! In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 312–315. IEEE.

[Moore, 1965] Moore, G. (1965). Moore's law. *Electronics Magazine*, 38(8):114.

[Neira-Ayuso et al., 2010] Neira-Ayuso, P., Gasca, R. M., and Lefevre, L. (2010). Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience*, 40(9):797–810.

[Oliphant, 2006] Oliphant, T. E. (2006). *A guide to NumPy*, volume 1. Trelgol Publishing USA.

[Omar et al., 2013] Omar, S., Ngadi, A., and Jebur, H. H. (2013). Machine learning techniques for anomaly detection: an overview. *International Journal of Computer Applications*, 79(2).

[Owens and Allen, 2010] Owens, M. and Allen, G. (2010). *SQLite*. Springer.

[Paez, 2017] Paez, A. (2017). Gray literature: An important resource in systematic reviews. *Journal of Evidence-Based Medicine*, 10(3):233–240.

[Pall, 2008] Pall, M. (2008). The luajit project. *Web site: http://luajit. org*.

[Paul and Maitra, 2011] Paul, G. and Maitra, S. (2011). *RC4 stream cipher and its variants*. CRC press.

[Perscheid et al., 2017] Perscheid, M., Siegmund, B., Taeumel, M., and Hirschfeld, R. (2017). Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1):83–110.

[Popper, 1989] Popper, K. R. (1989). Logik der Forschung.

[Postel et al., 1980] Postel, J. et al. (1980). RFC 768: User datagram protocol.

[Ramey, 2015] Ramey, C. (2015). Gnu readline library.

[Rescorla and Dierks, 2018] Rescorla, E. and Dierks, T. (2018). RFC 8446, the transport layer security (TLS) protocol version 1.3.

[Rosenberg et al., 2010] Rosenberg, J. et al. (2010). Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal for offer/answer protocols. Technical report, RFC 5245, April.

[Ruckert, 2015] Ruckert, M. (2015). *The MMIX Supplement: Supplement to The Art of Computer Programming Volumes 1, 2, 3 by Donald E. Knuth*. Addison-Wesley Professional.

[Schweizer, 2016] Schweizer, W. (2016). *MATLAB kompakt*. De Gruyter Oldenbourg.

[Sen, 2007] Sen, K. (2007). Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572.

[Shafranovich, 2005] Shafranovich, Y. (2005). Rfc 4180: Common format and MIME type for comma-separated values (CSV) files.

[Skyrme et al., 2008] Skyrme, A., Rodriguez, N. d. L. R., and Ierusalimschy, R. (2008). Exploring lua for concurrent programming. *J. Univers. Comput. Sci.*, 14(21):3556–3572.

[Steele, 1990] Steele, G. (1990). *Common LISP: the language*. Elsevier.

[Stein, 1995] Stein, S. (1995). *Stein on writing: a master editor of some of the most successful writers of our century shares his craft techniques and strategies*. Macmillan.

[Stonebraker and Rowe, 1986] Stonebraker, M. and Rowe, L. A. (1986). The design of postgres. *ACM Sigmod Record*, 15(2):340–355.

[Summers and Engineer, 2008] Summers, T. and Engineer, S. A. (2008). Hardware based gzip compression, benefits and applications. *CORPUS*, 3(2.75):2–68.

[Sutton et al., 2007] Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[Tamburri et al., 2020] Tamburri, D. A., Miglierina, M., and Di Nitto, E. (2020). Cloud applications monitoring: An industrial study. *Information and Software Technology*, 127:106376.

[Vaughan et al., 1999] Vaughan, G. V., Elliston, B., Tromey, T., and Taylor, I. L. (1999). Autoconf, automake, libtool.

[Vladyka, 2018] Vladyka, O. (2018). Kompresní metoda zstandard. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum.

[Weissenbacher, 2012] Weissenbacher, G. (2012). Explaining Heisenbugs. In *Runtime Verification*, volume 9333.

[Wolfram, 1991] Wolfram, S. (1991). *Mathematica: a system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc.

[Wolfram, 2002] Wolfram, S. (2002). *A new kind of science*, volume 5. Wolfram media Champaign, IL.

[Yuan et al., 2018] Yuan, Y., Yang, Y., Wu, L., and Zhang, X. (2018). A high performance encryption system based on AES algorithm with novel hardware implementation. In *2018 IEEE International Conference on Electron Devices and Solid State Circuits (EDSSC)*, pages 1–2. IEEE.

[Ziv and Lempel, 1977] Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343.

Chapter 17

Appendix: Reference Manual for anan

17.1 Installation and Setup

Since `anan` is using the GNU build system, it can be built by a simple invocation of

```
./configure && make
```

The usual conventions of configure scripts apply, e. g. the variables `CC`, `LD` override the compiler/linker path and `CFLAGS`, `LDFLAGS` append to the compiler's/linker's command line.

Cross compiling is supported via the option `--host`. The build process of `anan` needs to amalgamate all relevant binary and source objects into a C program. A tool called `luastatic` is used for that. It needs to run on the host. Building with the host compiler in a setting of cross compilation is not easily supported by autotools. Therefore, the following work-around is recommended for cross-compiling: (The option `-static` is optional.)

```
# in the directory from opening anan's tarball
LDFLAGS=-static ./configure
cd lua ; make ; cd .. # liblua.a is a requirement for luastatic
make luastatic ; mv luastatic luastatic.host ; make distclean
LDFLAGS=-static ./configure --host=aarch64-linux-musl # aarch64 is an example
cd lua ; make ; cd .. # trick dependency tracking into believing ...
make luastatic ; mv luastatic.host luastatic # ... luastatic is up to date
make
```

The build process should generate the following executable files:

- `anan`,

- `anand`,
- `new_keys`.

The latter is used to create key pairs for `anan`:

```
mkdir anand.keys
mkdir anan.keys
./new_keys
# keys are in anand.keys/ and anan.keys/
```

The binaries `anand` and `anan` expect their keys and their run control files to be in the current directory. The key creator `new_keys` is set up in such a way that `anan` would start happily in the directory `anan.keys` and `anand` in the directory `anand.keys`. The file `id` is the tool's private key, the `.pub` file is the other tool's public key.

The run control file is called `init.lua`. Using this facility is highly recommended. It is executed at startup and is expected to return a *table* of symbols (i. e., a module) to be added to `anan`'s global environment or to the sensors running on `anand`. An example of an `init.lua` for `anan`:

```
local module = {}
-- helper function for quickly loading content of small files
-- interactive use: slurp "init.lua"
-- no error handling
module.slurp = function(filename)
  local f = io.open(file)
  local rv = f:read("a")
  f:close()
  return rv
end

-- global variable
-- interactive use: storage[2]
module.storage = {"primus", "secundus", "tertius"}

-- there are no constants in Lua
-- interactive use: math.pi
module.PI = 4 -- approximately

return module -- compulsory statement
```

The database used for `anan` is at the fixed path `"/tmp/monitoring.db"`. If this should be inconvenient, it can be easily changed in the source code (variable `DBPATH`; appears twice (!)).

The source code of `anan` (including the build system's source) is in the public domain. The remaining sources ship their corresponding licenses either at the beginning of the source or in the file `LICENSE` in their source directories.

17.2 General

The command line interface understands Lua code. Both full statements (including branches and loops) and expressions are allowed. Every line has to be either a complete statement or a complete expressions; new lines cannot be escaped. (If lines grow too long, some of the material can be put into `init.lua`.)

Editing via `readline` is available, `.inputrc` is honoured.

The input prompt is `<-`, the prompt before output is `=>`. Normal output is green; error messages are red and include the full error message and the stack trace (if applicable). Multiple values are displayed with tabs as separators. Values are pretty-printed by the library `serpent` in the most verbose mode. This is hard-coded. If the user wants to avoid this, the value can be assigned to a variable (statements have no return value).

Lua's standard library is available; additionally, the following modules are compiled into `anan`:

- `readline`
- `zlib`
- `luatz` (time zone handling)
- `rxi-json-lua` (JSON parser and dumper)
- `luaposix` (bindings for POSIX functions)
- `nacl` (bindings for NaCl; implements only TweetNaCl)
- `serpent` (dumper for Lua's native format)
- `luaproc` (threading)
- `clutch` (high-level bindings for SQLite)

Arbitrary non-native modules can be loaded in the usual way. Native modules need to be plugged into the build system (for an example, see how `luaproc` is added to `Makefile.am` and `configure.ac` and see also `luaproc/Makefile.am`), unless they happen to be built with the exact same compiler and the same settings against the same `libc`.

17.3 Sensors

There is only one type of sensor which can be executed by `anand`. The following (nonsensical) example shows most features of sensors:

```
awk {
  id = "don't use weird characters in IDs\nit will confuse everyone/everything",
  files = { -- may be a list, a single string or empty
    "/etc/*", -- trailing commas don't hurt
  },
  variables = {
    FS = ":", -- field separator | built-in variable as in AWK
    KILL_PATTERN = "l33t p4tt3rn", -- user variable
    GLOB = true, -- expand globs in the section `files`
  }
  rules = {
    --
    'BEGIN', [[ -- before any file is opened
      fcntl = require "posix.fcntl";
      posix = require "posix";
      ]];
    -- condition #1
    'F[1]:match "roo." or #line > 42', [[ -- code #1
      yield{datum = line, msg = "weird line"}
      ]];
    -- condition #2
    'F[4]:match(KILL_PATTERN)', [[ -- code #2
      yield{datum = line, msg = "Your line was my last word!", f4 = F[4]};
      die(); -- but still run to completion and die only then
      ]];
    'END', [[
      yield{msg = "ups, forgot to use fcntl", fcntl = posix.F_SETLKW};
      ]]
  }
}
```

Sensors are constructed by the function `awk` (the value returned by it is not relevant). Sensors consists of up to five sections:

- `id`: If not given manually, then `anan` chooses names like `"anon-42"`.
- `file` is a list (or a single string) of file names to be opened during the main run.
- `hostnames` is a list (or a single string) of host names where the sensor should be deployed to. If this is not specified, the sensor is deployed everywhere.
- `variables`: Sensors can be given *user variables* to parametrise the sensor easily. Any value given here can be used as a global variable in the `rules` section. The following variables are built-in:

  - `RS`, `FS` are the record and field separators. A file is split into *records* (lines by default) according to the Lua pattern `RS`; each record is split into (space-separated by default) *fields* according to the Lua pattern `FS`. At the beginning of every record, `RS` and `FS` are checked; changing `FS` inside an action or condition does not split the current record into new fields, though.
  - If `GLOB` is set to a `true` value, filenames are expanded by `glob(3)`.

- `rules`: This is a list of pairs `(condition, action)`. The `condition` can be one of the following:

  - `BEGIN` or `END`. The `action` is to be executed before or after the files are processed.
  - The empty string. Equivalent to `true`.
  - A Lua expression. If the expression evaluates a `true` value, the action is executed.

  The action is a piece of Lua code. The following additional symbols are available:

  - `line`, `F[1]`, `F[2]`, ... are the current line and its fields. Assignments to `line` do not "automagically" affect `F` or vice versa.
  - `FNR`, `NR` count the number of records; `FNR` is reset to 1 at every new file.
  - `FILENAME`. The current filename, or `nil` in `"BEGIN"` rules. In `"END"` rules, the last values of `FILENAME`, `FNR` and `NR` are retained.
  - All symbols returned from `init.lua` are available. This is the only way for sensors to persist state between executions. (This use is not recommended.)
  - `yield()` adds its arguments to the returned object. Scalar values and arrays are appended; non-numeric keys are copied. Therefore, `yield{k=1}`, followed by `yield{k=2}`, will result in a returned object with a field `k=2`.
  - `die()` deactivates the current sensor. It does still run to completion and returns the aggregated result.

  As a shortcut, the sensor `awk(string)` is equivalent to:

```
awk {
  id = "quick-42",        -- possibly with a different number
```

```
3     hostnames = hostnames,  -- all host names,
4     files = {},
5     variables = {},
6     rules = {
7       'END', string         -- execute `string`
8     }
9   }
```

This is especially convenient with a string ending in `"die()"` for one-time measurements.

17.4 Use Sensor Data

Data is stored in memory for a few seconds in the variable `messages`, indexed by IP address and message number:

```
<- add_hostname{"primus", "secundus", "tertius"}
=> {             -- reports whether the addition was successful
true,
true,
true
}
<- awk { id = "testName", ... }     -- deploy some sensor
<- messages[ips.primus][1].awk.testName.msg
=> "value of msg"
```

Since this interface is very inconvenient, a somewhat better interface is given by the function `query()`.

```
1   -- returns an array of results
2   query "SELECT msg, max(time), hostname FROM testName"
3   query("SELECT hostname, time, msg FROM testName WHERE msg LIKE $pattern",
4         {pattern = thePattern})
```

The query language is SQL as understood by SQLite 3. The tables are sensor names, the columns are the sensor's results. (Sensors should refrain from changing their column sets.) There are automatically constructed columns `time` and `hostname`. Since SQLite does not directly support structured column values, sensors need to encode complex data sets as strings. JSON is preferred because of SQLite's built-in support. Alternatively, the database can be normalised manually to at least the first normal form.

Due to batched execution, `query` might see a delay of up to a few seconds. The same holds when the database is directly opened with SQLite's command line client.

17.5 Other functions

1. Specifying host names

   - `alpha(a, z)` generates a list of letters spanning from `a` to `z`.
   - `iota(i, j)` generates a list of numbers spanning from `i` to `j`; omitting `j` is equivalent to `iota(0, i)`.
   - `template(string, l1, ...)` interpolates the elements from l_i in the i—th place of the character `#`. The character `#` itself is escaped by doubling. Example:

```
<- template("#-#", {"storage", "compute", "infra"}, iota(2))
=> {
  "storage-0",
  "storage-1",
  "storage-2",
  "compute-0",
  "compute-1",
  "compute-2",
  "infra-0",
  "infra-1",
  "infra-2"
}
```

   - `map(f, l1, ...)` is useful for many things. It applies the function `f` to elements from the lists `l1`, `l2`, ... Instead of a list, a non-list element can be given. This is equivalent to a list containing infinitely many repetitions of that element. Example:

```
<- map(function(a, b, c) return a*b+c end, iota(4), iota(4, 8), 3)
=> {
  3,
  8,
  15,
  24,
  35
}
```

   - `filter(f, list)` returns a list with elements from `list` for which `f` returns true.

2. Processing sensor results

   - `lambda(string)` returns a function which evaluates `string`. The variables `x`, `y` and `z` are set to the first arguments of that function. Example:

```
1 increment = lambda "1+x"
2 plus = lambda "x+y"
3 lookup = lambda "y[x]"
4 map(lambda[[string.format("str[%q]", x)]], {"'\"'", "\0"})
5 map(lambda "x.complicated[1].value[4]", someSensorResult)
```

   - `place(where)` is for cases like the last example in line #5. The call to `lambda` could have been replaced by:

```
place ".complicated[1].value[4]"
```

   - `find(array, element, comparator)` returns the first index of `array` where `element` can be found. The default `comparator` is `lambda "x == y"`.
   - `zip(l1, l2, ...)` returns an array with the values of `l1`, `l2`, ... juxtaposed ("zipped together"). This can act as a poor man's `JOIN`, if the lists are time series with the same offset and spacing.

3. Managing sensors

   - `clear()` deletes all local sensor configurations and corresponding running sensors.
   - `ls(what)` lists all sensors, if `what` is `nil`. If `what` is a string, all sensors with names matching the Lua pattern `what` are returned.
   - `rm(what)` is similar, but removes the sensors from the local configuration (without updating the corresponding running sensors).
   - `update_configuration(hostname)` updates the `hostnames`' configuration to match the local one. This is only needed after `rm()` was called.

Chapter 18

Appendix: Lebenslauf des Verfassers

- geboren am 12. August 1989
- Abitur im Jahr 2009 am Lessing-Gymnasium Frankfurt
- Bachelor-Studium der Informatik mit Nebenfach Mathematik an der Goethe-Universität Frankfurt 2009–2012
- Abschlussarbeit zum Thema exakte Arithmetik; Prüfer waren M. Schmidt-Schauß und C. P. Schnorr
- Master-Studium der Informatik ohne Nebenfach an der Humboldt-Universität zu Berlin 2012–2016
- Abschlussarbeit zum Thema aktives Lernen regulärer Sprachen; Prüfer waren J. Köbler und S. Kuhnert.