EPJ Web of Conferences **245**, 01041 (2020) https://doi.org/10.1051/epjconf/202024501041
CHEP 2019

ANaN — ANalyse And Navigate

Debugging Compute Clusters with Techniques from Functional Programming and Text Stream Processing

Alexander Adler[1,*] and *Udo* Kebschull[1,**]

[1]Goethe-Universität Frankfurt

Abstract. Monitoring is an indispensable tool for the operation of any large installation of grid or cluster computing, be it high energy physics or elsewhere. Usually, monitoring is configured to collect a small amount of data, just enough to enable detection of abnormal conditions. Once detected, the abnormal condition is handled by gathering all information from the affected components. This data is processed by querying it in a manner similar to a database.
This contribution shows how the metaphor of a debugger (for software applications) can be transferred to a compute cluster. The concepts of variables, assertions and breakpoints that are used in debugging can be applied to monitoring by defining variables as the quantities recorded by monitoring and breakpoints as invariants formulated via these variables. It is found that embedding fragments of a data extracting and reporting tool such as the UNIX tool `awk` facilitates concise notations for commonly used variables since tools like `awk` are designed to process large event streams (in textual representations) with bounded memory. A functional notation similar to both the pipe notation used in the UNIX shell and the point-free style used in functional programming simplify the combination of variables that commonly occur when formulating breakpoints.

1 Introduction

Large-scale computation is indispensable, not only in high energy physics, but also in many other fields of physics, in fact in most fields of science. To simplify matters for *users* (i. e., scientists using computers in order to solve scientific or engineering problems) of such *compute clusters*, the users are not simply presented with bare computers. Rather, there are patterns (see, e. g., [1] for a common approach) of operating setups and software frameworks that enable users to spend more time thinking in the domain of their particular scientific problem and less time thinking in terms of general programming or system administration. Therefore, someone who is not identical to the user has to deal with issues of

1. setting up the cluster (operating system; user accounts; software packages in common use; site-specific software)
2. operating the cluster (ensuring most or all machines are actually running; network availability; patching operating system and other software; dealing with hardware upgrades; backup).

*e-mail: aadler@iri.uni-frankfurt.de
**e-mail: uk@rz.uni-frankfurt.de

Together, these two tasks enable *correct* and *efficient* behaviour of the cluster. Although correctness and efficiency are intertwined, simply put, proper setup leads to correct, but not necessarily efficient behaviour, whereas proper operation should aim at optimising efficiency without harming correctness.

Although both tasks are seemingly only necessary at the initial commissioning stage of the cluster, they remain relevant to a smaller degree throughout the cluster's operation. Systems that log information about (lack of) correct and/or efficient operations are called *monitoring* software. A broad overview is given in [2].

Often, there is a trade-off in terms of which and how much data to collect: if too little data is collected, no sensible conclusion can be drawn. If, on the other hand, too much data is collected, the very process of collecting, transmitting, storing or processing large amounts of data interferes with the cluster's workload — observing the system changes the system in adverse ways (sometimes dubbed "Heisenbug", see e. g. [3]).

Monitoring is a relatively static tool; after having decided on the variables to be monitored, changes in the frequency and precision of measurement require more than single-click actions. In fact, monitoring is not typically used as an interactive tool, but rather as a means of gathering data for later scrutiny. Therefore, the mode of interaction with monitoring is more similar to querying a database with past information than to inspecting a live system.

Finally, there are monitoring solutions that are ready to use in a matter of minutes. But since typical installations consist of many moving parts, significant changes can be required for production use, thereby negating all advantages of simple (initial) deployment.

These observations about monitoring mean: Once a user or the monitoring system notice degradation in correct and/or efficient operation, the first step includes looking at the data stored by monitoring. If this doesn't yield sufficient information for implementing an improvement strategy, monitoring — an often complicated subsystem with custom changes and configurations — needs to be changed in order to collect more relevant data. In plain terms: The cluster's operator burns to get numbers about what's happening on the nodes instantly, but he/she has to fiddle with monitoring! The proposed solution is a software dubbed "cluster debugger":

Definition. A *cluster debugger* is a tool for *interactive introspection* of data similar to that collected by monitoring and for *interaction* with nodes based on the outcome of that introspection. It does not necessarily introduce new capabilities relative to monitoring, instead, it caters to improved interactivity: The path between proposing a theory explaining the unwanted behaviour and finding evidence in favour or against the theory should consist of a few keystrokes at most.

1.1 Existing work

To the best of the authors' knowledge, there is no tool with the precise purpose of debugging a compute cluster. Anecdotally, a cluster debugger would have been of some use in certain situations that relate either to setup or maintenance of a cluster. Therefore, reviewing the state of the art has to focus on the two areas:

- setting up compute clusters
- general monitoring of information technology

General monitoring. A very extensive list featuring over 700 entries is hosted at the website of the SLAC National Accelerator Laboratory [4]. Most of these software packages come with complete documentation. The list dates back to the year 1996. The number of competing packages and the age of this branch of software makes it very plausible that monitoring is a mostly solved problem.

Setting up compute clusters. There are many tools for provisioning computers. Since every site has slightly different requirements and a basic provisioning tool can be developed in a few hours, new tools appear regularly. A detailed overview of the most popular orchestration and provisioning tools can be found in [5].

These two sets of tools are complementary: Monitoring is non-interactive, based on fixed configurations, and mostly useful after the initial setup. Conversely, orchestration has more interactive aspects (e. g., as a reaction to unexpected status message of the progress report, ad-hoc commands can be supplied), based on often-changing configuration or ad-hoc commands, and mostly used to perform the initial setup.

The cluster debugger has to supplement both sets of tools: It adds interactivity in order to deal with issues in deployment (initial setup) and operation (optimisation).

A similar idea appears when considering debuggers for software on single machines: Adding `print` statements at well-chosen points is very easy, tracing the execution of a program's relevant parts. On the other hand, this foregoes all interactivity that a symbolic debugger like `gdb` offers. This interactivity is a major advantage of symbolic debugging, which is a leading reason for the ongoing existence of symbolic debuggers. (The author found at least one symbolic debugger for each of the 20 leading programming languages listed on the TIOBE index as of January, 2020.)

A lot can be accomplished by "`print` debugging", although symbolic debugging (anecdotally) improves on that by adding interactive and introspective capabilities. On a similar note, it is expected that a more interactive monitoring tool adds to the capabilities of classical, non-interactive monitoring, yielding a tool similar in spirit to symbolic debuggers.

2 ANaN — ANalyse And Navigate

We present the cloud debugger ANaN. It is a software tool that tries to achieve the following properties:

- be easy to deploy to any Linux system (as high performance computing focusses on Linux)
- be reasonably easy to port to non-Linux, but POSIX-compliant systems
- interfere as little as possible with the system under consideration, ideally not at all (which, clearly, is impossible)
- enable powerful interaction with the systems under consideration
- be as easy to operate as possible
- be extensible by native libraries

To enable this, several software tools and techniques have been applied.

Choice of implementation language. The language Lua 5.3 was chosen. Lua is a multi-paradigm scripting language that is particularly *light-weight*; the full distribution weighs less than 20 000 lines of code. The small size is hoped to lead to a *small memory footprint*. The implementation language is the subset of C89 that also compiles as C++, therefore giving access to a rich set of *libraries*. The syntax and semantics of Lua are similar to other "scripting" languages, although not identical. It is therefore hoped that a casual user should be able to *rapidly learn* how to write tiny snippets of Lua code (mostly single expressions or single statements) in order to operate ANaN. Lua's *parser* can be used to deserialise large strings into data structures relatively fast (according to [6], Lua's original use case was configuration of scientific software, which necessitates a not-too-slow parser that can deal with deeply nested structures); there are sufficiently fast libraries for serialising Lua data structures into its textual format. Therefore, Lua's syntax for literal structures (strings, integers and structured data types) was chosen for the wire format.

Functional Programming [7, 8]. This is a programming paradigm that focuses on the mathematical notion of a function — a side-effect free computation — and views programming as the composition of functions. Sharing of mutable state is either impossible or needs at least to be made explicit.

The original plan for ANaN didn't call for any functional techniques. After a phase of experimentation, a powerful class of data sources ("AWK sensors"; to be introduced in the next paragraph) was found to be most useful when no or only minimal state was changed during collection of data. Furthermore, concatenations of tiny code snippets were found to be a flexible toolbox for evaluating sensor data. The first observation led to further inquiry into programming techniques; the second observation led to the UNIX pipe operator which directs one program's output into the next program's input. Both observations together make a strong point for functional techniques.

AWK. According to the POSIX programmer's manual[9], AWK is a "pattern scanning and processing language". The previous paragraph's AWK sensors are loosely modelled after AWK, especially its general mode of operation:

- execute all `BEGIN` code blocks
- process the given list of files, and for each (generalised) line of each file do:
  - split the line into tokens
  - if the line matches a given pattern, execute the corresponding action
- execute all `END` code blocks
- return data gathered during execution

In fact it was found that all data sources ("sensors") that have been implemented or planned prior to the implementation of AWK sensors can be implemented as AWK sensors, most of them using less code. Although the above outline shows many building blocks for AWK sensors, sensible defaults make almost all of them optional. It is found that typical AWK sensors use only two or three of the building blocks. A simple AWK sensor can be constructed ad-hoc at the command line; conversely, the behaviour of most short AWK sensors is simple and predictable. Therefore, AWK was chosen as the paradigm for the construction of sensors. In fact, all other types of sensors have been removed and replaced by corresponding AWK sensors.

AWK is not a functional programming language by any standard: Actions are allowed to create and mutate state. But most AWK sensors used so far don't mutate state, or could be changed not to do so, albeit using more computation time and network capacity as a result. It is unclear if a modified language for AWK sensors would lead to purely functional sensors, or how this modification would need to be performed.

2.1 Architecture

Overview. This briefly lists the main components and architectural decisions taken for the implementation of ANaN; see also figure 1. Each statement is explained in more details in the subsection *Rationale*.

1. ANaN consists of the two programs `anan` and `anand`: a *debugger* and a *demon* program. The debugger is the interactive tool. The demons run on each system under consideration.

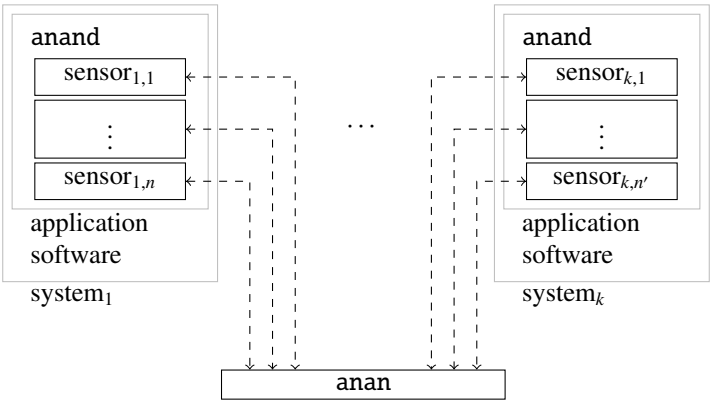

Figure 1. Overview of ANaN: Many systems run `anand` besides the main applications; each instance of `anand` executes a set of sensors; configurations and results are communicated to a central instance of `anan`.

2. Every demon is configured with zero or more *sensors*. A sensor is a small algorithm (see the previous discussion of AWK sensors) that gathers information. Each sensor is run approximately every second.

3. The debugger presents the resulting time series as *variables*. The debugger also has basic facilities to combine (e. g., correlate) and post process variables.

4. *Breakpoints* which act on the value of a variable can be defined. A breakpoint can have one or more associated *watch lists*: Lists of variables whose values are stored into circular buffers. After the breakpoint fires, the past watched values leading to the breakpoint are available for introspection. Additionally, a few more values are stored in the buffer.

5. Sensors are formulated in an analogous way to AWK scripts. In place of AWK, the language for statements and expressions is Lua 5.3. Both `anan` and `anand` have *run-control files* `init.lua` which are executed at each start. They can be used to load additional modules for sensors and for use in the debugger.

6. The debugger is a command-line tool. The user communicates with it via Lua commands.

Rationale. Here, each of the details listed in the *Overview* is motivated and compared to possible alternative implementations or techniques. Each numbered paragraph corresponds to the paragraph given there.

1. There has to be at least one place that sees all the data which is to be presented to the user. The program `anan` does this. It also implements the user interface. Additionally, each system under consideration needs to run some software that communicates the necessary information to `anan`. This is implemented as the separate program `anand`. An alternative would use persistent connections via `ssh` or similar. Although virtually every system exposes `ssh` in some way, it was considered unwise to depend on this service because it can be (and often is) configured in complicated ways which may necessitate more configuration ANaN. Furthermore, `ssh` is critical for the cluster's

operation, and the possibility of ANaN degrading this service can never be completely excluded.

2. The frequency of about one evaluation per second was chosen as a compromise between overloading the systems under consideration (by gathering more often) and missing interesting events (by gathering more rarely). Since a typical sensor does only very little file input/output and little computation, even this relatively high frequency (more typical frequencies are once every 10 or 60 seconds) shouldn't have too much of an impact. The waiting is implemented by a call of `sleep(1.0)`. Although this makes the frequency slightly imprecise, it also prevents `anand` from cannibalising the machine's resources, as long as the sensors don't run for prolonged time.

3. The facilities mostly follow the MapReduce paradigm [10]: Each variable can be transformed algorithmically, and multiple variables can be correlated or combined in other ways. There is even a rudimentary graphing routine using Unicode Block Characters for a simple visualisation of a single time series. All more advanced analysis and graphing should be delegated to more specialised tools.

4. The combination of breakpoints and watch lists allows the following workflow: A theory is proposed, claiming that a failure mode observable by the variable v is being caused by the variables $w_1, w_2, \ldots$. To find evidence for this theory, a breakpoint depending on w is constructed with watch list $w_1, w_2, \ldots$. Once the breakpoint fires, the theory can be verified by inspecting the recorded data from the watch list. The sizes of the circular buffers are configurable, so slowly developing processes can be observed.

5. The run-control files allow site-specific modifications, as well as a standard set of sensors and evaluation pipelines that are deemed to be universally helpful. Since the entire standard library of Lua is available, additional dynamic libraries may be loaded. The extensibility comes at the price of possibly reduced *security*: The program `anand` needs to run with elevated privileges. Otherwise, it is unlikely to be useful because much interesting information cannot be accessed by unprivileged users.

   As a security measure, all communication between `anan` and `anand` is encrypted and signed via NaCl [11]. To the authors' best knowledge, this should settle the question of security: If the attacker controls the communication between the two programs, it is infeasible to forge new messages (unless there is a security flaw in NaCl). The attacker could replay old messages. Thereby, he could reset the configuration of demons to old states (which were originally set by the debugger's operator), or have the demons send old messages. It is unclear how an attacker can take advantage of this, although both attacks can be prevented by requiring increasing sequence numbers on each message.

   If the attacker controls the run-control files or the binaries of the programs, the system under consideration is already compromised.

6. The debugger's operator is not expected to develop code at the command line. Commonly used sensors and evaluation steps can be provided in the run-control files. Several syntactic shortcuts available in Lua can make it seem that the command language for `anan` is not a real programming language. Consider the following sensor:

```
swaps = awk {                       -- an AWK sensor
    files = '/proc/swaps',          -- operating on this file,
    rules = {                       -- yielding a mapping of
        '', 'yield{F[1] = F[3]}' -- swap file name to size
```

```
}                                   -- (field 1 maps to field 3)
```

Sensors of this complexity can be considered "extended one-liners"; more so, since this line is more likely to be typed as

```
swaps=awk{files='/proc/swaps',rules={'','yield{F[1]=F[3]}'}}
```

Nevertheless, this verbose syntax is not considered optimal and therefore likely to change.

3 Results and Conclusion

First synthetic benchmarks. A synthetic benchmark with 1000 systems under consideration was performed; another benchmark with 10 000 is being prepared. The systems under consideration are Docker containers [12], where 100 containers (throttled to 1% of a CPU) run on each of ten virtual machines. The virtual machines are executed by KVM [13]. The virtualisation is dimensioned such that each virtual machine runs with a single CPU at 2100 MHz with 2 GB of RAM. (The operating system both on the virtual machines and in the containers is Ubuntu 16.04.) An arbitrary CPU-intensive, but almost IO-free task runs in each container. Additionally, `anand` is executed. A set of six simple sensors similar in complexity to `swaps` above is executed. A manual introspection with `top` saw only fluctuating values around 0% for the CPU usage of `anand`. — The debugger was run on another virtual machine in a container. Although it was processing messages from 1000 demons, the system load stayed around 0.1. No message loss was observed. No further analysis pipelines were executed.

Deployment in production. A deployment in the next run of the ALICE experiment [14] is being prepared. First, ANaN should run alongside the regular monitoring and closely mimic its data gathering pattern. It will be tried to use primarily ANaN instead of monitoring to search for unwanted behaviours, unless a problem in ANaN makes this impossible.

Future work. The niche of *cluster debuggers* is so-far unpopulated — if rightly so, will be shown by the results of the above-mentioned experiments. Besides, the following future goals seem relevant:

- simpler user interface (sensor and evaluation language)
- a set of standard sensors that is sensible to deploy almost anywhere
- the possibility to visually inspect variables
- real-time sensors that plug into kernel facilities such as Netlink [15], thereby continuously gathering information that is submitted in second-spaced bunches

Source. The source code is currently hosted at
`https://chiselapp.com/user/kedorlaomer/repository/anan`

Conclusion. This paper presented ANaN, a debugger for compute clusters. It is found that the reason for the lack of cluster debuggers is likely not technical; the tools presented here have sufficient performance for being usable with hundreds, if not thousands of machines, without interfering too much with the system under consideration. It is shown that it can be used analogously to monitoring. The added value ove monitoring remains to be demonstrated.

References

[1] G. Hager, G. Wellein, *Introduction to high performance computing for scientists and engineers* (CRC Press, 2010)
[2] G. Aceto, A. Botta, W. De Donato, A. Pescapè, Computer Networks **57**, 2093 (2013)
[3] G. Weissenbacher, *Explaining heisenbugs*, in *Runtime Verification* (2012), Vol. 9333
[4] *Network monitoring tools*, `https://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html`
[5] K. Torberntsson, Y. Rydin, *A study of configuration management systems: Solutions for deployment and configurationof software in a cloud environment* (2014)
[6] R. Ierusalimschy, *Programming in lua* (Roberto Ierusalimschy, 2006)
[7] H. Abelson, G.J. Sussman, *Structure and interpretation of computer programs* (The MIT Press, 1996)
[8] M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi, *How to design programs: an introduction to programming and computing* (MIT Press, 2018)
[9] *The Open Group Base Specifications Issue 7, 2018 edition* (2018)
[10] J. Dean, S. Ghemawat, Communications of the ACM **51**, 107 (2008)
[11] D.J. Bernstein, Networking and Cryptography library **3**, 385 (2009)
[12] D. Merkel, Linux journal **2014**, 2 (2014)
[13] Y. Goto, Fujitsu Scientific and Technical Journal **47**, 362 (2011)
[14] K. Aamodt, A.A. Quintana, R. Achenbach, S. Acounis, D. Adamová, C. Adler, M. Aggarwal, F. Agnese, G.A. Rinella, Z. Ahammed et al., Journal of Instrumentation **3**, S08002 (2008)
[15] P. Neira-Ayuso, R.M. Gasca, L. Lefevre, Software: Practice and Experience **40**, 797 (2010)