The Back End is Only One Part of the Picture: Mobile-Aware Application Performance Monitoring and Problem Diagnosis

Katrin Angerbauer[1], Dušan Okanović[1], André van Hoorn[1], Christoph Heger[2]
[1]University of Stuttgart, Institute of Software Technology, Stuttgart, Germany
[2]NovaTec Consulting GmbH, CA Application Performance Management, Leinfelden-Echterdingen, Germany

ABSTRACT

The success of modern businesses relies on the quality of their supporting application systems. Continuous application performance management is mandatory to enable efficient problem detection, diagnosis, and resolution during production. In today's age of ubiquitous computing, large fractions of users access application systems from mobile devices, such as phones and tablets. For detecting, diagnosing, and resolving performance and availability problems, an end-to-end view, i.e., traceability of requests starting on the (mobile) clients' devices, is becoming increasingly important. In this paper, we propose an approach for end-to-end monitoring of applications from the users' mobile devices to the back end, and diagnosing root-causes of detected performance problems. We extend our previous work on diagnosing performance anti-patterns from execution traces by new metrics and rules. The evaluation of this work shows that our approach successfully detects and diagnoses performance anti-patterns in applications with iOS-based mobile clients. While there are threats to validity to our experiment, our research is a promising starting point for future work.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **General and reference** → Measurement; • **Human-centered computing** → Empirical studies in ubiquitous and mobile computing;

KEYWORDS

application performance monitoring, performance anti-patterns, iOS

ACM Reference Format:
Katrin Angerbauer[1], Dušan Okanović[1], André van Hoorn[1], Christoph Heger[2]. 2017. The Back End is Only One Part of the Picture: Mobile-Aware Application Performance Monitoring and Problem Diagnosis. In *VALUETOOLS 2017: 11th EAI International Conference on Performance Evaluation Methodologies and Tools, December 5–7, 2017, Venice, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3150928.3150939

VALUETOOLS 2017, December 5–7, 2017, Venice, Italy

ACM ISBN 978-1-4503-6346-4/17/12…$15.00
https://doi.org/10.1145/3150928.3150939

1 INTRODUCTION

In recent years, the number of users that use mobile devices has been increasing and has surpassed the number of desktop users [5]. However, users are still sensitive to issues when using mobile devices. A report by Google states that 61% of the users are unlikely to return to a mobile web site or application if they had trouble accessing it [2]. Also, the conversion rates are still higher for desktop users, but it is expected that conversion rates on mobile devices play a more significant role in the future. For example, Amazon reported that during 2015's holiday season, more than 70% of shoppers used Amazon's application on mobile devices to place orders [7]. Therefore, the success of businesses will heavily be influenced by the success of the mobile applications that support these businesses. The success and the acceptance of these applications will, in turn, be heavily influenced by their performance.

In order to maintain service levels in enterprise application systems (EASs), the usual approach is to monitor them using application performance monitoring (APM) tools [9]. Most of the currently available APM tools support end-to-end monitoring [8]. Data collection and analysis on the back end side, i.e., inside the data center, is a well known research field and common practice with many available tools and approaches [9]. However, very often, performance problems experienced by users with mobile devices are not caused by or visible inside the application back end. Examples include slow client-side rendering times due to the hardware and software technology stack on the client device, the characteristics and quality of the network connection (WiFi, cellular, etc.), the current geolocation, etc. In case of desktop clients accessing EASs via broadband networks, we can safely assume that the connection quality is constant, and the slow response of the system, can be attributed to bad software design decisions. In case of accessing from mobile device, we are often faced with variable network speed that depends on the network quality. The location of the device, which is considered fixed for desktop clients, can also be a factor. Therefore, additional metrics have to be included into root-cause analysis.

In our previous work [10], we presented the *diagnoseIT* approach for automated diagnosis of performance problems in enterprise applications. The approach is based on the research of performance anti-patterns [18]. It diagnoses anti-patterns, such as the N+1 anti-pattern when accessing remote services or databases, by analyzing execution traces [1] collected during operations.

In this paper, we extend our *diagnoseIT* approach to EASs that have mobile applications as front end clients. We collect the data both from the mobile application and the back end using agents for the respective platforms. The collection process utilizes underlying system APIs, and includes standard metrics known from EASs, such as timing data and resource consumption, but also mobile-specific

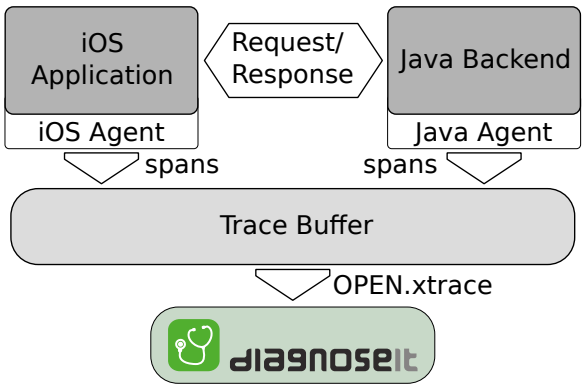


Figure 1: Proposed monitoring approach

metrics, such as geolocation and network information. Data from both platforms is then combined and analyzed using *diagnoseIT*.

The proof-of-concept implementation presented in this paper uses iOS-based mobile clients accessing a Java-based back end. However, thanks to the use of the open and technology-agnostic APIs and formats OpenTracing [20] and OPEN.XTRACE [13] the approach is independent of the technology platforms and APM tools.

To summarize, the contribution of this paper is threefold:

- An approach for end-to-end monitoring of mobile applications based on the vendor-neutral OpenTracing and OPEN.XTRACE specifications. In comparison to existing APM tools, we also monitor what is happening inside the mobile application, not only its communication with the back-end.
- The diagnosis of performance problems in mobile applications using the *diagnoseIT* approach, extended to deal with performance problems typical for mobile platforms.
- A proof-of-concept implementation and evaluation based on an iOS/Java-based application system setting.

The remainder of this paper is organized as follows. Section 2 presents our approach, which is evaluated in Section 3. Related work is discussed in Section 4. In Section 5, we draw the conclusions and outline future work. Supplementary material for this paper, is available online.[1]

2 APPROACH TO MOBILE-AWARE PERFORMANCE MONITORING AND ANALYSIS OF APPLICATION SYSTEMS

An overview of our approach is depicted in Figure 1. The data is collected using monitoring agents for mobile and back end platforms, respectively (Section 2.1). The implementation of both agents is based on the OpenTracing standard [20]. These agents send the collected data to the *Trace Buffer*, which merges the data from different system tiers. This data is then converted into the OPEN.XTRACE format and analyzed using *diagnoseIT* (Section 2.2).

[1]https://doi.org/10.5281/zenodo.1012746

2.1 Collecting the Data

Monitoring of mobile applications provides a challenge because operating systems on mobile devices do not allow the modification of applications once they are published. Therefore, monitoring approaches that are well known from EASs, such as instrumentation or using external monitoring agents [9], cannot be implemented. This is why commercial, out-of-the-box solutions usually provide only a limited number of end-point metrics, e.g., number of remote calls or call durations, which are usually extracted by the monitoring data from HTTP communication.

In order to be able to collect the data from different platforms, and to be able to correlate and merge it for the analysis, we use the concept of *spans* from OpenTracing. Spans, delimited using pairs of start- and end-points inserted into application code by developers, are abstract representations of certain code regions that will be executed. They contain unique identifiers and timestamps, and can be nested inside each other, allowing for hierarchical structures. An example of how spans can be inserted into the code is shown in Listing 1. The resulting hierarchical structure is illustrated in Figure 2.

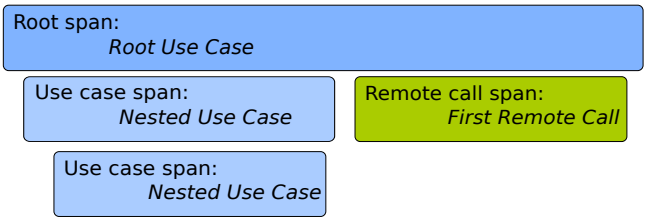


Figure 2: An example of a span hierarchy

Developers create spans using agents, which are singleton, and are responsible for their closing, in order to prevent an overlap between spans. In our case, the top span is always designated as the *root*, and it contains sub-spans, which can further be *remote call* spans and *use-case* spans. It also contains a unique identifier—*trace identifier*, which is passed to all the spans that are contained in it. Remote call spans represent calls from one application node (e.g., mobile device), to another (e.g., application back end), and contain the data about the call, e.g., HTTP request, as well as the trace identifier from the root span. In order to be able to trace a client's call from the mobile device to the back end, we inject the span identification data into HTTP requests, as well as into responses. More specifically, this data is added into the headers as additional attributes. When an HTTP response arrives, based on the identifier, the remote call span is closed by the agent. Use-case spans represent a specific execution within the application, and their creation triggers a data collection. The semantic of spans is not predefined by standard. It is up to the developer who creates them by inserting them into the source code to specify what they represent. A span can cover anything from one simple operation to a complex processing that includes several components.

The Trace Buffer (Figure 1) performs the merging of the data from multiple nodes, in our case from the mobile device and the application back end, based on the unique trace identifier from the root span. This procedure within the Trace Buffer effectively

Listing 1: Example Code for a simple root use case with nested use case and remote call

```
1  // start the root use case
2  Agent.getInstance().startRootUsecase(name: "Root Use Case")
3  // ... (application logic)
4
5  // start the first sub use case
6  Agent.getInstance()
7         .startUsecase(name: "First Use Case",
8                       root: "Root Use Case")
9  // ... (application logic)
10
11 // nest one use case
12 Agent.getInstance()
13        .startUsecase(name: "Nested Use Case",
14                      root: "First Use Case")
15 // ... (application logic)
16
17 // close the nested use case
18 Agent.getInstance()
19        .closeUsecase(name: "Nested Use Case",
20                      root: "First Use Case")
21 // ... (application logic)
22
23 // close the first sub use case
24 Agent.getInstance()
25        .closeUsecase(name: "First Use Case",
26                      root: "Root Use Case")
27 // ... (application logic)
28
29 // start a remote call with timeout
30 Agent.getInstance()
31        .startRemoteCall(name: "First Remote Call",
32                      root: "Root Use Case",
33                      httpMethod: "GET", request: &rq)
34
35 // ... processing of the first remote call
36
37 // closing successfully completed remote call
38 Agent.getInstance()
39        .closeRemoteCall(name: "First Remote Call",
40                      root: "Root Use Case",
41                      responseCode: 200, timeout: true)
42 // ... (application logic)
43
44 // close the root use case
45 Agent.getInstance().closeRootUsecase(name: "Root Use Case")
```

removes the need to independently analyze and manually correlate data from different platforms. When the Trace Buffer receives the data and finishes merging, it converts it into the OPEN.xtrace representation, so it can be analyzed by *diagnoseIT*. A simplified object diagram based on the example in Figure 2 is shown in Figure 3.

As stated previously, the data is being collected during the execution of use case spans. The metrics collected in the back end of EASs are well known in literature [9]. Here we discuss metrics that we consider in mobile applications. They can be grouped into four categories. The list of metrics considered and collected by our approach is provided in Table 1.

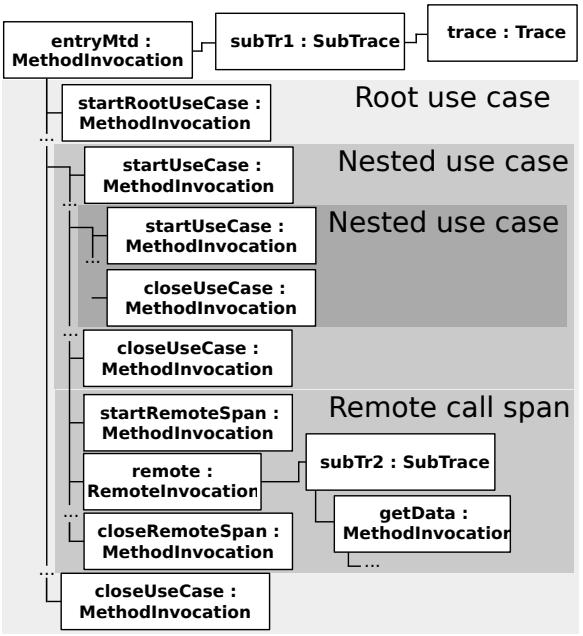


Figure 3: An object diagram of the trace in OPEN.xtrace format, based on Figure 2

System-related metrics. These metrics are collected from the operating system itself. Examples of these metrics are CPU, memory, and hard drive usage for the application. Some metrics from this group, such as overall CPU usage of the system, which are usually available in enterprise systems and are valuable in root-cause analysis, are not available in mobile systems.

Network-related metrics. These metrics are collected from the network interface. They include the information about the network provider and the network connection. Network provider information can be interesting, for example, to compare detected problems with problems reported by the provider.

Location data. These metrics are collected from the mobile device's location services. This is useful to detect if some performance problems are location-dependent.

The data from these three categories is collected using sampling when the span is created. The frequency of sampling is configurable, but this should be done with caution. A too low frequency may lead to missing important data, while a too high frequency causes unacceptable overhead.

Remote call data. is collected for remote call spans by intercepting calls towards the application back end, and contains the information extracted from HTTP requests.

2.2 Analyzing the Data

Our previous work on *diagnoseIT* successfully diagnoses performance anti-patterns in EASs. This approach can be applied to the back end applications, but our analysis of mobile systems shows that some anti-patterns known from EASs can appear in mobile applications.

Metric	Collected from
CPU usage	System
Hard drive	System
Memory	System
Battery power	System
WLAN active	System
Network provider	Network
Network connection	Network
SSID	Network
Geolocation	Location data
Response code	Remote call
Timeout	Remote call

Table 1: Overview of considered mobile metrics

One example of these anti-patterns is the **Blob** anti-pattern [14], a situation where one class contains most of the application's logic, while others serve only as data containers. In large systems, this class represents a performance bottleneck, and it should be broken down into several top-level classes. In this paper, we will not focus on these anti-patterns, as their diagnosis was already discussed in our previous work [10]. Instead, we focus on diagnosing five anti-patterns known from EASs, but with slight modifications, and two glitches—situations that appear as anti-patterns, but actually are not.

In case of the **Ramp** anti-pattern [19], we differentiate between the **Memory Ramp** and the **Hard drive Ramp**, i.e., increasing memory and hard drive consumption over time, respectively. Both anti-patterns can be diagnosed by observing memory and hard drive consumption over time. **High memory utilization** refers to the situation when the device slows down due to the fact that the memory is full and the system stores more data on the hard drive. This also influences the battery consumption. Similarly, **High hard drive utilization** forces the system to try to free more space by, e.g., deleting unused data, decreasing the performance and consuming more battery. These two anti-patterns are diagnosed by setting a threshold on the utilization.

Too many remote calls [23] from mobile device can have serious performance impact and is usually solved by merging many similar calls into one, to reduce the network overhead. There are two special cases of this problem that we identified: many remote calls to the same target (e.g., server, page) and many remote calls to the same URL (e.g., the same picture is loaded multiple times instead of loading it once and saving it). These anti-patterns are diagnosed by analyzing the number of remote calls from a method that is designated as slow.

There are also some situations that may appear as performance problems, but are actually caused by temporary problems in the device's environment. We designate these as glitches rather than as performance anti-patterns. If an application is contacting its back end, a performance issue may be reported, but it can be caused by the mobile network's **high latency**. In order to properly diagnose this issue, we need to check if the long response time was due to a network delay. The solution to this problem is for the application to check the connection speed before sending large amounts of data.

Listing 2: Excerpt of the rule that diagnoses a short timeout

```
1 rule "Short timeout"
2 ...
3       for(RemoteInvocation call:
4                       trace.getRemoteInvocations())
5               if(call.isTimeoutFlag()
6                       && call.getTargetSubTrace() != null) {
7                       /* detected! */
8               }
```

Listing 3: Excerpt of the rule that diagnoses a memory ramp

```
1 rule "Memory Ramp"
2 ...
3       Map<Long, Long> ramValues = ...; // timestamp, value
4       for(Trace trace: traces.get(startTime, endTime))
5               for(Callable callable: trace.getCallables())
6                       ramValues.add(getRamValues(callable));
7       if (calculateSlope(ramValues) > SLOPE_THRESHOLD) {
8               /* detected! */
9       }
```

Related to this issue is the situation when the application reports a timeout, but the response arrives later (**Too short timeout**). In this case, we check for the existence of the response. The solution is to increase the application's timeout settings, if the network connection is slow.

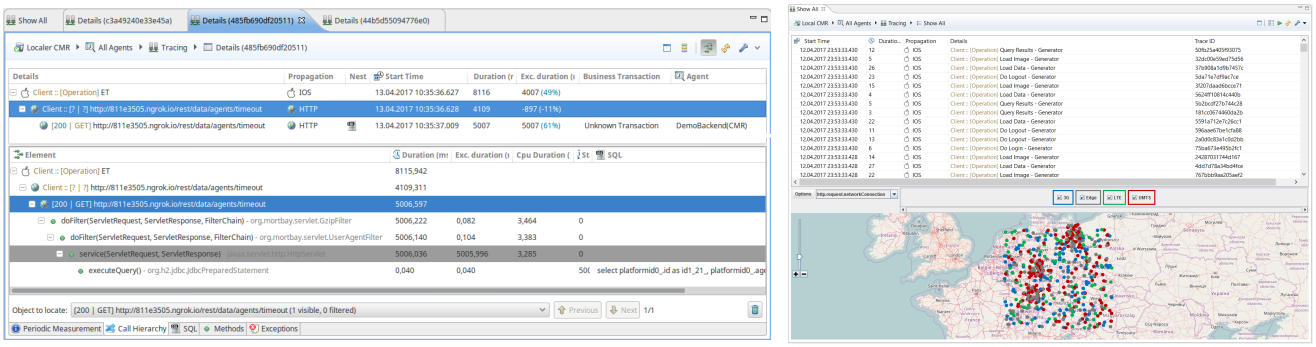
To implement the diagnosis of these performance anti-patterns and glitches, we implement an additional set of rules, and extend *diagnoseIT* with them. As *diagnoseIT* is designed to be extensible, no changes are needed, except for the rule set.

Here we present two examples of these rules, while the others are available in the appendix of this paper. Too short timeout is diagnosed by checking if the remote call from the mobile application side, which did not receive a response in timely manner, has an existing target subtrace on the back end side (Listing 2). If the subtrace exists, that means that the application back end processed the request, but did not have enough time to send a response. If no subtrace exists, that would mean that the processing of the request failed and something else was the problem. An increase in memory utilization is detected by collecting memory utilization measurements from all the callables in traces from a certain time interval, calculating the slope from them and comparing it to a predefined threshold (Listing 3). For details on the OPEN.xtrace concepts mentioned here see [13].

3 EVALUATION

The goal of the evaluation is to find out to what extent our approach can diagnose performance problems in applications with mobile clients and properly diagnose them. We evaluate our approach by running different scenarios for anti-pattern and glitch diagnosis. We also need to properly assess our approach by investigating what are the correct parameters, e.g., thresholds, and data sampling rates.

To summarize, we want to answer the following research questions:



(a) Trace analysis

(b) Mobile-specific data

Figure 4: Analysis of traces collected with the iOS agent and mobile specific data in diagnoseIT

RQ1: Do the implemented rules correctly diagnose performance anti-patterns and glitches in mobile systems?
RQ2: What are suitable values for the configuration parameters of the rules?

The remainder of this section is organized as follows. Section 3.1 describes the experimental methodology, Section 3.2 reports the experiment results and Section 3.3 discusses threats to validity.

3.1 Experimental Methodology

For this evaluation, we use a sample iOS application, which allows clients to browse the data about books in a library. This data is retrieved from the Java back end. The application is instrumented with the iOS agent developed in this work, while the back end is instrumented using the existing inspectIT Java agent [3]. The application contains performance problems that can be turned on and off from the application interface. APM data was collected and sent to *diagnoseIT*, which was integrated into the inspectIT client (Figure 4).

To test the diagnosis of the previously described anti-patterns and glitches, we construct eleven scenarios detailed in Table 2. Scenario 1 represents the application without a known performance problem, Scenarios 2–5 represent performance problems being present in the application, while Scenarios 6-11 represent the problems in the communication between the application and the back end.

In all the scenarios, we collect the data and run *diagnoseIT* to test for anti-patterns/glitches, and compare the diagnosis results with the expected results from the respective scenarios. It is important to note that the thresholds, which were used in some scenarios, are based on our experience from working with the application and the back end.

3.2 Evaluation Results

In this section, the results of the different scenarios are presented. We provide only an overview of results. Complete results, with the experimental setup, are available in the supplementary material.

As it was expected, in Scenario 1, based on the data in Table 3, no rule was triggered, and therefore no anti-patterns were discovered.

To test whether performance issues stemming from a nearly full hard disk and high memory utilization in Scenarios 2 and 3 are diagnosed, we used the threshold values of 97% and 90% respectively. During the experiment, the measured hard disk utilization of 97.73% was higher than the specified threshold of 97%, and memory usage was 99.89% compared to the 90% threshold. In both cases, *diagnoseIT* correctly identified the slow methods. However, in the next step, high resource utilization is identified as the root-cause.

Based on the results for Scenario 4, shown in Table 4, increased memory utilization was detected. An average increase of 0.01% per second was higher than the specified threshold of 0.005% per second. In this case, memory utilization was sampled five times per second, during the execution of use case spans. This frequency provided enough data for the analysis, while keeping the monitoring overhead low.

In Scenario 5, memory ramp and high hard disk utilization were correctly diagnosed, based on the results shown in Table 5. The hard drive utilization threshold was set to 97% and 97.78% was detected. The threshold for the memory ramp was set to 0.001% per second, and an increase of 0.003% per second was detected. Additionally, due to increasing memory consumption, memory utilization also exceeded the threshold of 97%, which was also diagnosed correctly.

The diagnosis of the short timeout in Scenario 6, was correctly diagnosed by checking for the existence of the response in the back end, which was not registered by the mobile application.

It took the server at least five seconds to respond, while we set the timeout interval to four seconds.

In Scenario 7, the high latency was tested and diagnosed in a similar way with the predefined latency threshold, with one difference: the timeout was long enough for the response to be accepted by the mobile application.

In Scenario 8, the application performed too many (11) remote calls to the same URL, which was correctly diagnosed by setting the predefined number of calls as threshold to 4.

In Scenarios 9 and 10, both problems were diagnosed correctly using the approaches described in the previous scenarios—Scenario 8 for many remote calls, and Scenarios 6 and 7 for timeout and high latency, respectively.

In Scenario 11, a high number of remote calls was correctly diagnosed, but because of the missing information from the back

Nr.	Scenario name	Description
1	No performance problems	Normal application execution when no performance problem is activated.
2	Hard disk utilization too high	The hard disk of the test device is filled with random data.
3	Memory utilization too high	When the application starts, the available physical memory size is computed, allocated, and initialized. After that, the application continues executing.
4	Increase in memory usage	Periodically, a certain amount of memory is allocated to simulate a memory leak, resulting in a memory ramp.
5	Combination of memory and hard disk issues	Both high hard disk utilization and increase in the memory utilization are simulated in a same way as in Scenarios 2 and 4, respectively.
6	Remote call with a short timeout	The mobile application performs a remote call with a specified timeout to the tracked back end system. The back end provides a response, but only after the timeout, so it is never registered by the application.
7	Remote call with a high latency	Similar to the previous scenario, but here we set the timeout to be long enough.
8	Too many equal URL calls to a tracked back end	Many remote calls to the same URL are conducted.
9	Many remote calls with a short timeout setting	The timeout issue from Scenario 7 is combined with performing too many remote calls from Scenario 6.
10	Many remote calls with a high latency	The high latency issue from Scenario 8 is combined with performing too many remote calls from Scenario 6.
11	Many remote calls to an untracked back end with short call timeout setting	The application performs many remote calls to the same URL, but the back end is not monitored.

Table 2: Overview of considered evaluation scenarios

Timestamp	Battery Power	CPU Usage	Memory Usage	Storage Usage
16:46:45.374	100 %	0.00 %	72.64 %	95.08 %
16:46:50.374	100 %	0.00 %	72.73 %	95.08 %
16:46:55.374	100 %	0.00 %	72.57 %	95.08 %
16:47:00.374	100 %	0.05 %	72.76 %	95.08 %
16:47:05.374	100 %	0.00 %	72.60 %	95.08 %
16:47:10.374	100 %	0.00 %	72.40 %	95.08 %
16:47:15.374	100 %	0.00 %	72.62 %	95.08 %
16:47:20.374	100 %	0.00 %	72.48 %	95.08 %
16:47:25.374	100 %	0.00 %	72.69 %	95.08 %
16:47:30.374	100 %	0.00 %	72.53 %	95.08 %

Table 3: Measurement samples from Scenario 1

Timestamp	Battery Power	CPU Usage	Memory Usage	Storage Usage
16:40:02.299	100 %	0.07 %	95.60 %	93.95 %
16:40:03.299	100 %	0.05 %	97.60 %	93.95 %
16:40:04.300	100 %	22.48 %	99.82 %	93.95 %
16:40:05.299	100 %	23.23 %	99.81 %	93.95 %
16:40:06.309	100 %	1.40 %	99.79 %	93.95 %
16:40:07.309	100 %	0.05 %	99.79 %	93.95 %
16:40:08.308	100 %	0.07 %	99.77 %	93.95 %

Table 4: Measurement samples from Scenario 4

Timestamp	Battery Power	CPU Usage	Memory Usage	Storage Usage
16:31:46.452	100 %	0.10 %	81.52 %	97.78 %
16:31:47.451	100 %	0.15 %	81.78 %	97.78 %
16:31:48.450	100 %	46.15 %	98.75 %	97.78 %
16:31:49.451	100 %	7.70 %	98.39 %	97.78 %
16:31:50.451	100 %	0.25 %	98.34 %	97.78 %
16:31:51.451	100 %	0.10 %	98.29 %	97.78 %

Table 5: Sample of measurements for Scenario 5

end, the actual cause of the problem (short timeout duration) was not diagnosed.

We can conclude that the evaluation has shown that the newly defined anti-patterns can be successfully diagnosed and that the configuration values seemed suitable for anti-pattern diagnosis. However, for future research the approach should be tested with real world data, e.g., using other applications, to ensure more validity for the results (as discussed in Section 3.3).

3.3 Threats to Validity

In this section we discuss the shortcomings of our experiments.

External Validity: The results of the evaluation might not be generalizable because the tested scenarios may not consider all relevant cases. Moreover, we used only one sample mobile application and the respective Java back end.

Internal Validity: We did not test if the iOS agent is reliable, i.e., we cannot guarantee that the collected mobile measurements are true. We can assume that the agent is reliable, since the collected measurements were as we expected, and because we rely on the underlying standard APIs.

Construct Validity: We injected problems to the demo application so that the rules take effect. We were biased during the implementation of the application. We did not test the anti-patterns with other mobile applications and cannot guarantee that the rules work for them.

Conclusion Validity: In some experiment runs the rules were not triggered, since the data from the back end, monitored by the Java agent, did not arrive in the buffer when the diagnosis was triggered. Therefore, *diagnoseIT* analyzed incomplete traces, which led to false negative results. We cannot guarantee that the rules work correctly for these experiment runs.

4 RELATED WORK

An overview of APM tools by Haight and Da Silva [8] shows that there are various commercial (e.g., Dynatrace, AppDynamics, CA APM) and open-source (e.g., [3, 22]) APM tools available. According to this analysis, some of them, e.g., Dynatrace, AppDynamics and, New Relic support monitoring of mobile applications. However, this is usually limited to monitoring communication of the mobile application with the back end (the so called *end-to-end monitoring*) and collecting some system metrics. To the best of our knowledge, support for monitoring that will provide information of what is going on inside of the mobile application's components is not available, or is custom built by application developers [9]. Regarding the analysis of monitoring results, anomaly detection and/or alerting based on, e.g., baselines calculated from historical data or manually defined thresholds, is available, while problem diagnosis is limited to identifying the component that is designated "problematic," not the real root-cause.

In *diagnoseIT* [10], detection and semantification of performance problems is based on research on performance anti-patterns [18]. There are works in this field that focus on architectural performance problems [6, 21], but the general problem with model-based approaches is that they are limited by the lack of information about the real system. There are also approaches designed for the testing phase, that are not suitable for production scenarios [4, 12]. In the work by Wert et al. [23], the authors propose to systematically perform load tests and search for anti-patterns based on symptoms detected in their results. In our work, we use their classification of anti-patterns based on symptoms. Parsons and Murphy [15] and Peiris and Hill [16] propose approaches for anti-pattern diagnosis, but both have certain limitations. The approach by Parsons and Murphy is technology-specific, i.e., works only in Java EE environments, while the approach of Peiris and Hill diagnoses only the existence of the One-lane Bridge anti-pattern, without providing the root-cause.

Some newer (commercial) tools going into a direction to *diagnoseIT* have been announced [11, 17]. However, to the best of our knowledge they do not provide any semantics to performance problems, and their analysis strategies are not extensible.

5 CONCLUSION

In this paper we proposed an approach for monitoring of EASs that use the applications on mobile devices as front end. We also provided a proof-of-concept implementation for a setup that includes iOS applications using a Java back end. Based on the collected data, we performed the root-cause analysis using our *diagnoseIT* approach. For this, *diagnoseIT* was extended to be able to diagnose performance problems that are typical for mobile applications.

Future work includes a further investigation of other performance anti-patterns, particularly those whose detection requires metrics more specific to mobile environment, e.g., location and mobile network characteristics. We also plan to work on improving the current diagnosis rules, and evaluating the approach with real-world applications. The implementation of this approach for mobile devices using other operating systems is planned. Finally, we would like to find other ways to implement data collection, as the one presented here mixes application logic and monitoring, which causes problems in code maintenance.

A RULES

Here we provide the (simplified) implementation of the rest of the rules used in this paper.

A.1 Hard Drive Ramp

The Ramp in hard drive utilization is diagnosed in the same way as memory ramp shown in Listing 3. The only difference is that we collect hard drive utilization (`getHddValues()`) for each callable in the trace.

A.2 High Memory/Hard Drive Utilization

High memory utilization is diagnosed by collecting memory utilization values for callables and comparing them to the predefined threshold.

Hard drive utilization is diagnosed in the same way. The difference is that we collect "hard drive values" for each callable in the trace, and compare them to the `HDD_THRESHOLD` value.

Listing 4: Excerpt of the rule that detects a high memory utilization

```
1 rule "High memory utilization"
2   ...
3   Map<Long, Long> ramValues = ...; // timestamp, value
4   for(Trace trace: traces.get(startTime, endTime))
5     for(Callable callable: trace.getCallables())
6       if(getRamValues(callable)) > MEM_THRESHOLD) {
7         /* detected! */
8       }
```

A.3 Too Many Remote Calls

Too many remote calls is diagnosed by counting the number of remote calls in the trace and comparing the number to the threshold.

Listing 5: Excerpt of the rule that detects too many remote calls

```
rule "Short timeout"
...
  int noOfCalls = trace.getRemoteInvocations().size();
  if(noOfCalls > NO_OF_CALLS_THRESHOLD) {
    /* detected! */
  }
```

A.4 High Latency

High latency is diagnosed by measuring the duration of the call on the client side and the duration of the processing on the server side. The difference between these two values is then compared to the predefined threshold.

Listing 6: Excerpt of the rule that detects high latency

```
rule "High latency"
  ...
  for(RemoteInvocation call: trace.getRemoteCalls())
    long callDuration = calculateDuration(call);
    long subTraceDuration =
                calculateDuration(call.getTargetSubTrace());
    if(callDuration - subTraceDuration > LATENCY_THRESHOLD) {
         /* detected! */
    }
```

ACKNOWLEDGEMENTS

This work is being supported by the German Federal Ministry of Education and Research (grant no. 01IS15004, diagnoseIT), and by the Research Group of the Standard Performance Evaluation Corporation (SPEC RG, http://research.spec.org). Special thanks go to Tobias Angerstein, Alper Hidiroglu, Simon Lehmann, Manuel Palenga, Oliver Röhrdanz, Matteo Sassano, Christopher Völker — Master's students at the University of Stuttgart — for their support in the development of this approach, as well as to Stefan Siegl for his valuable input.

REFERENCES

[1] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proc. ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation (PLDI '97)*. 85–96.

[2] Nora Aufreiter, Julien Boudet, and Vivian Weng. 2014. Why marketers should keep sending you e-mails. (2014). http://www.mckinsey.com/business-functions/marketing-and-sales/our-insights/why-marketers-should-keep-sending-you-emails

[3] Patrice Bouillet, Matthias Huber, Ivan Senic, and Stefan Siegl. 2017. inspectIT. (2017). http://www.inspectit.eu/

[4] Lubomír Bulej, Tomáš Kalibera, and Petr Tůma. 2005. Repeated Results Analysis for Middleware Regression Benchmarking. *Performance Evaluation* 60, 1-4 (2005), 345–358.

[5] Dave Chaffey. 2017. Mobile Marketing Statistics compilation. (2017). http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

[6] Vittorio Cortellessa, Anne Martens, Ralf Reussner, and Catia Trubiani. 2010. A Process to Effectively Identify "Guilty" Performance Antipatterns. In *Proc. the 13th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'10)*. 368–382.

[7] Lisa Eadicicco. 2017. More People Now Shop on Amazon Using Smartphones and Tablets Than Computers. (2017). http://time.com/4162188/amazon-holiday-shopping-statistics-2015/

[8] Cameron Haight and Federico De Silva. 2016. Gartner's Magic Quadrant for Application Performance Monitoring Suites. (2016). http://www.gartner.com/

[9] Christoph Heger, André van Hoorn, Mario Mann, and Dušan Okanović. 2017. Application Performance Management: State of the Art and Challenges for the Future. In *Proc. 8th ACM/SPEC on Int. Conf. on Performance Engineering (ICPE '17)*. 429–432.

[10] Christoph Heger, André van Hoorn, Dušan Okanović, Stefan Siegl, and Alexander Wert. 2016. Expert-Guided Automatic Diagnosis of Performance Problems in Enterprise Applications. In *Proc. 12th European Dependable Computing Conf., EDCC 2016*. 185–188.

[11] Instana. 2017. Instana - Dynamic APM for Microservice Applications. (2017). http://www.instana.com/

[12] Zhen Ming Jiang, A.E. Hassan, G. Hamann, and P. Flora. 2009. Automated performance analysis of load tests. In *IEEE Int. Conference on Software Maintenance, (ICSM 2009)*. 125–134.

[13] Dušan Okanović, André van Hoorn, Christoph Heger, Alexander Wert, and Stefan Siegl. 2016. Towards Performance Tooling Interoperability: An Open Format for Representing Execution Traces. In *Proc. 13th European Workshop on Computer Performance Engineering, EPEW 2016*. 94–108.

[14] Palma, F. and Nayrolles, M. and Moha, N. and Guéhéneuc, Y. G. and Baudry, B. and Jézéquel, J. M. 2013. Soa antipatterns: An approach for their specification and detection. In *International Journal of Cooperative Information Systems, 22(04), 1341004.*

[15] Trevor Parsons and John Murphy. 2008. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology* 7, 3 (2008), 55–91.

[16] Manjula Peiris and James H. Hill. 2014. Towards Detecting Software Performance Anti-patterns Using Classification Techniques. *SIGSOFT Softw. Eng. Notes* 39, 1 (Feb. 2014), 1–4.

[17] Ruxit. 2017. Ruxit All-in-one Application Performance Management. (2017). http://ruxit.com/

[18] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proc. 2nd Int. Workshop on Software and Performance (WOSP '00)*. 127–136.

[19] Connie U Smith and Lloyd G Williams. 2002. New software performance antipatterns: More ways to shoot yourself in the foot. In *Proc. Int. CMG Conf.* 667–674.

[20] Open Tracing. 2017. opentracing.io. (2017). https://www.opentracing.io

[21] Catia Trubiani and Anne Koziolek. 2011. Detection and Solution of Software Performance Antipatterns in Palladio Architectural Models. In *Proc. 2nd ACM/SPEC Int. Conf. on Performance Engineering (ICPE '11)*. 19–30.

[22] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proc. 3rd ACM/SPEC Int. Conf. on Performance Engineering (ICPE '12)*. 247–248.

[23] Alexander Wert, Jens Happe, and Lucia Happe. 2013. Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments. In *Proc. of the 2013 Int. Conf. on Software Engineering (ICSE '13)*. 552–561.