Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Evaluating Mobile Monitoring Strategies for Native iOS Applications

Matteo Sassano

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Dr.-Ing. André van Hoorn Dr. Dušan Okanović, Dr.-Ing. Christoph Heger
Commenced:	July 19, 2017
Completed:	January 19, 2018
CR-Classification:	C.4, H.3.4

Abstract

The success of a company is often influenced by the service and by a product they offer. If the supplied service or the offered product is a software system, a good performance will be essential to achieve desired goals such as high product sales. Slow applications and server responses due to performance issues, may cause a negative chain reaction. The amount of actual and potential users will probably decrease, and so does the users' satisfaction and the number of product sales. Application Performance Management (APM) is necessary to avoid these cases. The usage of APM could help detecting eventual software problems and to remediate performance issues afterwards.

Meanwhile, the usage of mobile devices, e.g., smartphones and tablets, for accessing enterprise systems is increasing in every application category. This expands the space where a potential software problem might be located in. Performance of mobile applications is more influenced by external circumstances, e.g., user location and access from bandwidth limited networks. APM tools not supporting mobile monitoring, are not able to recognize the mentioned performance issues.

There are different implementation strategies for application monitoring agents such as call stack sampling and full source code instrumentation. The goal of this thesis is to research agent strategies for mobile devices, to develop an own version of each agent type, to analyze and evaluate the different agent approaches in combination of various mobile application types. The evaluation will be done with a series of experiments, by measuring the outcoming overhead of the developed agents, integrated into previously selected representative iOS open-source applications.

Kurzfassung

Der Erfolg eines Unternehmens wird oft durch den angebotenen Service oder das angebotene Produkt beeinflusst. Wenn der Dienst oder das Produkt in Form eines Softwareprogramms bereitgestellt wird, dann ist gute Performance wichtig, um die gewünschten Businessziele, wie hohe Verkaufszahlen, zu erreichen. Langsame Anwendungen und Serverantworten aufgrund von Performancedefiziten, können eine negative Kettenreaktion auslösen. Sowohl die Anzahl der aktuellen und potenziellen Nutzer, als auch die Zufriedenheit der Nutzer und die Anzahl der Produktverkäufe wird sehr wahrscheinlich sinken. Application Performance Management (APM) ist erforderlich, um diese Fälle zu vermeiden. Die Verwendung von APM hilft, eventuelle Softwareprobleme zu erkennen und anschließend Performanceprobleme zu beheben.

In der heutigen Zeit nimmt die Nutzung mobiler Geräte, z. B. Smartphones und Tablets, für den Zugriff auf Enterprisesysteme in jeder Anwendungskategorie zu. Dies erweitert den Raum, in dem sich ein potenzielles Softwareproblem befinden könnte. Die Leistung von mobilen Applikationen wird stärker durch äußere Umstände beeinflusst, z. B. sowohl durch den Benutzerort, als auch durch den Zugriff von bandbreitenbegrenzten Netzwerken. APM-Tools, die keine mobile Überwachung unterstützen, können die genannten Leistungsprobleme nicht erkennen.

Es gibt verschiedene Implementierungsstrategien für Softwareagenten, wie z. B. Call-Stack Sampling, Byte-Code Intrumentation und die vollständige Quellcode Instrumentierung. Das Ziel dieser Arbeit ist es, mobile Agentenstrategien zu erforschen, eine eigene Version jedes Agententyps zu entwickeln und die verschiedenen Agentenansätze in Kombination verschiedener mobiler Applikationstypen zu analysieren und auszuwerten. Die Evaluation wird mit einer Reihe von Experimenten durchgeführt. Die entwickelten Agenten werden in zuvor ausgewählte iOS-Open-Source-Anwendungen integriert, und der entstehende Performanceoverhead wird gemessen.

Contents

List of Figures

List of Tables

List of Acronyms

AOP Aspect-oriented Programming

APM Application Performance Monitoring

CPU Central Processing Unit

EUM End-User Monitoring

RAM Random-Access Memory

List of Listings

List of Algorithms

Chapter 1

Introduction

This document presents mobile agent design and implementation strategies for mobile devices. For detecting performance problems in mobile applications, source-code and device based data has to be **collected**, **managed and maintained** and **sent** for further analysis. In each of the mentioned essential work stages of an agent, there are several practicable approaches. The resulting approaches need to be evaluated in terms of functionality, performance overhead and usability.

The motivation of this work is introduced in Section 1.1. Section 1.2 describes our goals for this thesis. Section 1.3 lists a set of tasks that has to be performed in order to achieve the defined goals. The last section, Section 1.4, of this chapter specifies notations for this document and the organization.

1.1 Motivation

Slow enterprise systems, may negatively influence the end-user satisfaction and the company success due to performance deficits. For instance, an increase of load times may discourage the end-user to utilize the application. The consequence is the loss of actual and potential clients. Being prepared for performance problems occurring on mobile devices is even more difficult considering the high distribution of the application. The application development team is only aware of the problems when it is too late. This is the case when ratings become very low the application badly or the earnings of the company are decreasing. To be prepared for performance problems we have to recognize them as soon as possible. Therefore, we need to establish a bridge between the application and the developer or problem analysts. Application performance monitoring (APM) provides solutions for monitoring the application and for further investigating performance problems. Figure 1.1 illustrates how to typically adapt the application

in order to monitor the user activity. In the first phase, the application developer has to choose an agent configuration to integrate in the application. Dependent on the monitoring strategy, this task has a longer or shorter duration. After integrating the agent configuration, the developer has to rebuild and to publish the application. Henceforward, the agent runs within the core application and automatically sends collected monitoring data to a monitoring server. At the end, the data can be requested and analyzed by a monitoring client.

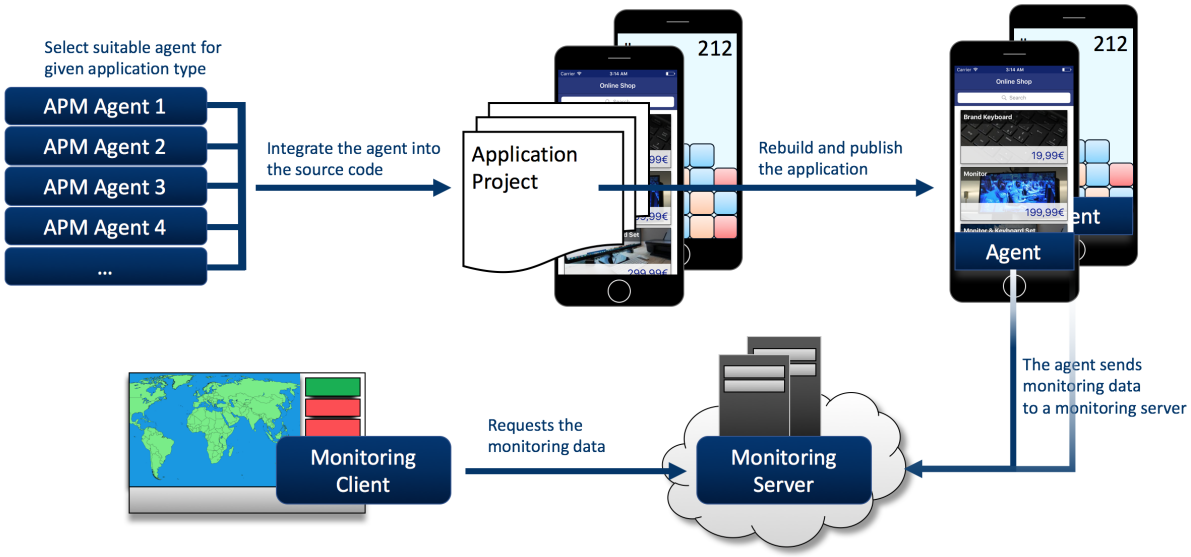


Figure 1.1: APM for Monitoring Applications

There are several mobile iOS application performance monitoring solutions on the market using slightly different design and implementation approaches. In addition to that, new automated approaches such as call stack sampling and automated instrumentation code injection, which are known but not existent for iOS solutions, will be developed in the course of this thesis. On the other hand we may have different application types using hardware resources differently from each other. In this thesis we will investigate, whether a perfect agent configuration exists for a certain application type.

1.2 Goals

In the following, we will list the goals of this thesis.

Application classification: The application classification is an important task needed for the evaluation. Each application type will be monitored with an agent configuration. This is needed to determine the best setting between agent and application.

Research of important system libraries: For instrumenting iOS system libraries we have the options to wrap them and to instrument the wrapped structure or to trace them. If we think about automated measurement strategies, there needs to be a recognition of certain class names and instance methods to place the instrumentation points on the right position.

Design and implementation of the strategies: A mobile agent working pipeline is made up of collecting data, manage data and dispatch data. Each strategy needs to be designed and implemented for iOS mobile devices.

Experimental valuation of the strategies: Since we have many strategies for each agent task, we have to select the implemented concepts for a set of agent configuration. Each agent configuration will be practically evaluated with at least two representative open source systems. By profiling the application we may notice differences in terms of device resource usage. Additional criteria are noticed performance impacts and the integration overhead of the agent configurations.

1.3 Tasks

The aim of the thesis is to research, to design and to implement different strategies for the different task types of an mobile agent and to evaluate them on various native iOS applications, to determine in which circumstances an agent concept is suitable for a certain application class. The tasks fulfilling the requirements are split in theoretical tasks and practical tasks. Figure 1.2 illustrates the task dependency graph. The following subsections will describe in detail the necessary tasks and their dependencies. In this section we will also present the work program containing all main tasks and subtasks.

1.3.1 Theoretical Tasks

This subsection lists all theoretical tasks needed for this work. Additionally, each task is described and the necessity is explained.

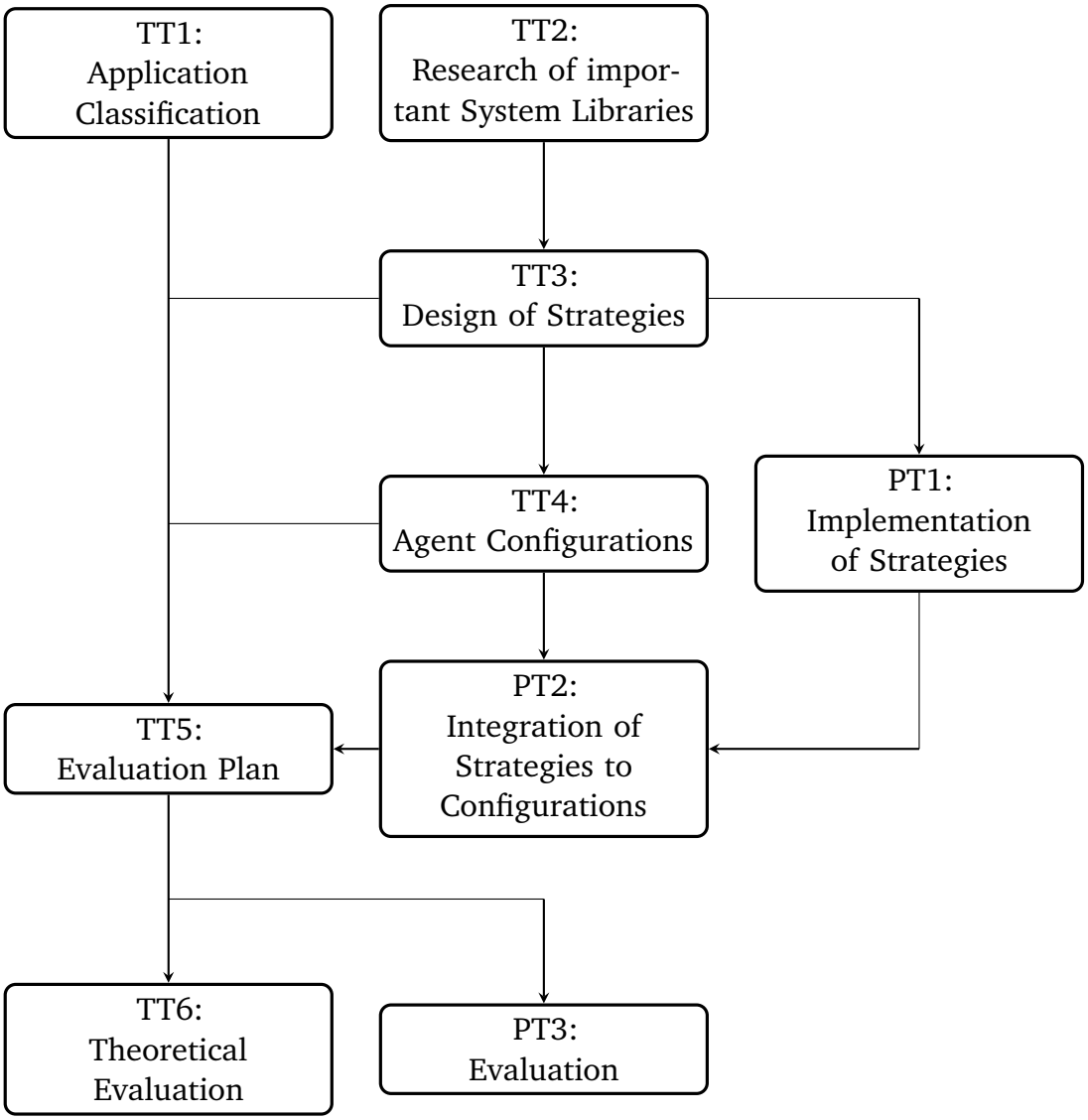


Figure 1.2: Tasks pipeline

ID	Theoretical Task	Dependency
TT1	Application Classification	-
TT2	Research of important System Libraries	-
TT3	Design of Strategies	TT2
TT4	Agent Configurations	TT3
TT5	Evaluation Plan	TT1, TT3, TT4, PT2
TT6	Theoretical Evaluation	TT1, TT2, TT4, TT5

Table 1.1: Theoretical Tasks

TT1 – Application Classification

Description: Research application types based on their resource and network usage, the used frameworks and the application category on.

Goal: We want to minimize the set of different application types. Apple provides 24 selectable application categories [APC] and 72 frameworks [APF]. The aim of this task is to reduce the set of application classes by finding common resource usage properties between different categories and frameworks.

TT2 – Research of important System Libraries

Description: For this task we will research the most used iOS system libraries provided by Apple. The results will be filtered by priority. A system method has a higher priority then other ones if performance problems could occur by performing it.

Goal: The aim of this task is to provide a set of instrumentable methods. Method swizzling allows to wrap these methods and to add instrumentation points. We could use this advantage and prepare the agent to instrument important system methods out of the box. If the developer uses the automated instrumentation approach, the code parser also need to recognize system methods to add instrumentation points to them.

TT3 – Design of Strategies

Description: This task a specification of the possible approaches. It will present how each strategy needs to work and the construction of it.

Goal: The result of this task is to clarify the structure of each strategy. This task forms the base for implementing the defined strategies.

TT4 – Agent Configuration

Description: Each agent configuration is made of a conjunction of different strategies but not all approaches can be combined together. Therefore, we need this task to analyze possible configuration and to define them.

Goal: We will define a set of available agent configurations. The resulting configurations are needed for the theoretical and the practical evaluation.

TT5 – Evaluation Plan

Description: The purpose of this task is to design an evaluation plan explaining how we want to compare the monitoring strategies.

Goal: The resulting plan defines the evaluating methodology. This will help evaluating the researched, defined and implemented monitoring strategies.

TT6 – Theoretical Evaluation

Description: Monitoring strategies with the same purpose will be compared theoretically in terms of efficiency and will be classified in a ranking list.

Goal: The theoretical evaluation will present a ranking list of agent configurations based on a theoretic comparison. These results are independent of the results of the practical evaluation and may not cohere to them.

1.3.2 Practical Tasks

This subsection lists all practical tasks needed for this work. Additionally each task is described and the necessity is explained.

ID	Theoretical Task	Dependency
PT1	Implementation of Strategies	TT3
PT2	Integration of Strategies to Configurations	TT4, PT1
PT3	Evaluation	TT1, TT4, TT5, PT2

Table 1.2: Practical Tasks

PT1 – Implementation of Strategies

Description: We will implement all defined mobile agent monitoring strategies for iOS devices.

Goal: The implementations are needed to integrate them in an agent configuration.

PT2 – Integration of Strategies to Configurations

Description: The implemented strategies will be integrated in the configurations defined in Section 1.3.1.

Goal: A set of different agent configurations is needed to perform the experimental evaluation.

PT3 – Evaluation

Description: Preselected open-source applications will be profiled with predefined use-cases. The different agent configurations will be integrated afterwards and the applications will be profiled again. While integrating the agents to the applications, we will measure the spent time of each integration. At the end the profiling results will be compared to each other and respectively to the defined criteria and the gathered data we will create a ranking list, which tells the application developer which configuration is the most suitable for his developed app.

Goal: The evaluation result explains the application developer, which agent configuration is the most suitable for his application.

1.4 Document Organization

This section describes the typographical formalia of this document and presents the thesis structure.

1.4.1 Formalia

In the following paragraphs we introduce typographical conventions used to emphasize certain passages or words of this document.

Paragraphs: Paragraph names are written in **bold**.

Emphasizing: To emphasize certain words or sentences in this document, we use the *italic* font weight.

`Classes/Methods/Attributes`: Source code-based words such as class names, methods or attributes are written with the `typewriter` font.

[Reference]: References are used for referring to external work such as papers, books, theses and web pages. Each reference links to its bibliography entry.

1.4.2 Thesis Structure

This subsection describes the document structure by mentioning and summarizing each chapter of this work.

Chapter 2 – Foundations and Technologies: For a better understanding, this chapter is providing descriptions of terms mentioned throughout this document and describes the current development level of the application performance monitoring tools focusing on mobile agents.

Chapter 3 – Application Classification: In this chapter we are displaying differences between iOS applications in terms of system properties and resource usage. By recognizing those differences we are able to define a set of application classes.

Chapter 4 – Requirements for Monitoring Agents: This chapter describes the requirements needed to fulfill the conception and implementation of an agent configuration (and for performing the evaluation). Additionally, this chapter describes the necessary tasks to be performed for creating an agent configuration.

Chapter 5 – Architecture of Monitoring Agents: The pipeline of a working iOS agent will be described in this chapter. Additionally we will present and explain in detail all strategies for each working state of an agent.

Chapter 6 – Implementation: All various strategy implementations will be presented and explained in detail in this chapter. Additionally we will present all agent configurations which are possible to implement with the researched approaches.

Chapter 7 – Evaluation: The evaluation is the most important part of this thesis. The outcome of the evaluation is a recommendation of an agent configuration for each application class. In this chapter, we will describe our evaluation approaches, our experimental settings and present the results.

Chapter 8 – Conclusion This chapter summarizes the results of this thesis and provides ideas for future works.

Chapter 2

Foundations and Technologies

The introduction gave an overview about this topic and argued the essentiality to evaluate the different agent approaches. Researching, developing, and evaluating existing and new strategies for mobile agents are the main goals of this thesis.
For a better understanding, Section 2.1 is providing descriptions of terms mentioned throughout this document. Section 2.2 describes the current development level of the Application Performance Monitoring tools focusing on mobile agents. Currently available iOS application monitoring agents are listed in Section 2.3. Section 2.4 introduces the basics of the development language *Swift*, in order to understand the implementations in Chapter 6.

2.1 Terminology

Application Performance Monitoring: End-user monitoring (EUM), runtime application architecture discovery, transaction profiling, component deep-dive monitoring and analytics are the five categories of application performance monitoring [GAR17]. The aim of APM is to achieve a solid level of application performance. [HHMO17]

APM tool: APM tools manage and monitor the performance and availability of software systems. Displaying application and resource information, requires run time data. In most cases an agent, instrumenting the monitored system, is used to collect performance measurements.

Agent: The APM agent is a software instance, which collects runtime application information such as resource usage, invocation sequences, execution times and remote call information. In addition to that agents' tasks are to store and manage the collected data and to transmit them to the central APM controller server. In the context of this thesis, it describes agents for monitoring mobile applications.

Mobile agent: In this context, a mobile agent is an agent, tracing on mobile devices. The difference between both agents is the collection of mobile based metrics of the mobile agent.

Performance measurement: In the context of APM, a performance measurement represents relevant data from a monitored software. Invoked method names, used arguments, server response times, execution times, thrown errors or exceptions and device information are examples which can be part of a performance measurement [HWH12]. In the course of this thesis, these measurements are collected from mobile devices. Therefore, it also contains cellular unique attributes, e.g., the geoloaction, network connection type and the carrier.

Performance problem: A performance problem is represented by an undesired performance decrease of the software, derivating from high hardware resource usage or from high response times [IHE15].

Execution trace: An execution trace is an invocation sequence through the monitored software. A trace composition is based on a tree structure of sub-traces [HRH+09]. This thesis focuses on execution traces of mobile applications.

Use Case: In this context, a use case resembles a certain application activity performable from the end user on the mobile device.

End-to-end trace: This term can be described with an associated use case, as represented in Figure 2.1. The end-user with a mobile device is able to perform certain application operations. These inputs executed by the user will produce a mobile trace. In case the user effectuates a remote call, for loading a specific content for the mobile application, the back end system may perform various method invocations to respond

the remote call. The execution trace tracked on the back end system and the mobile execution trace collected by the mobile agent can be connected and rebuilt to an end-to-end trace [CMF+14].

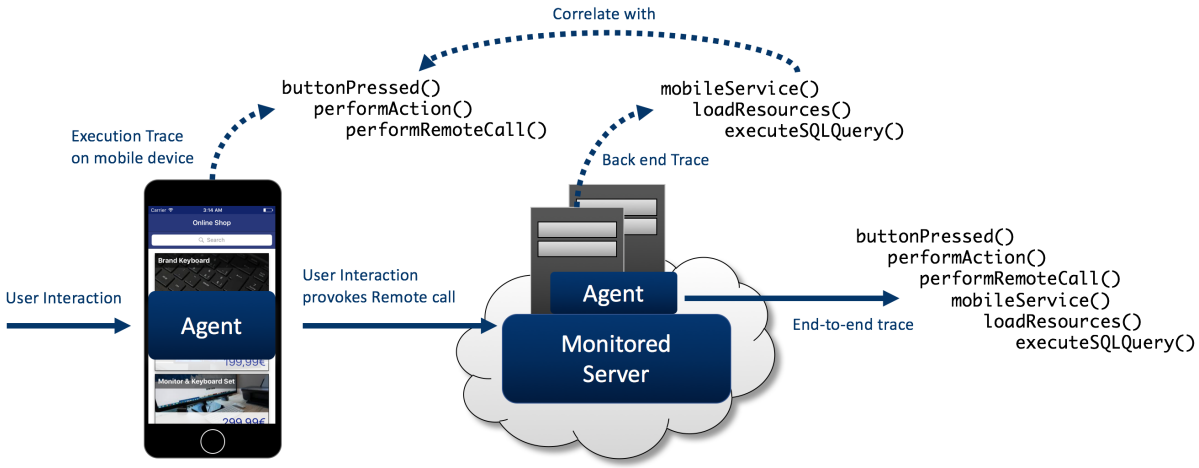


Figure 2.1: Building an end-to-end trace

The terms below belong to the technical language of native iOS development. These terms will often emerge in topics which are related to the agent concept or the agent implementation.

Selector: A selector [APL115] holds the name of a method and is used to select its associated method implementation for the execution.

Method swizzling: Method swizzling [NK14] refers to swapping a method implementation with another one at runtime. This makes it possible to change system functions, even without knowing and possessing the implementation of them.

CocoaPods: CocoaPods [COC17] is a dependency manager for native iOS applications. This service provides a Podfile, which one can link needed libraries with the specific versions. Podfiles need to be installed, with the command `pod install`, afterwards to make the changes effective. CocoaPods is providing more than 31,000 libraries.

2.2 Strategies for developing monitoring agents

In the following, we will review the state of the art with respect to the topic goals and specifically focusing on the currently available monitoring strategies for the agent tasks, existing mobile monitoring tools and the related APM tools. An agent needs to collect invocation sequence and resource usage information, to store the data, to manage the collected instances and to care about the measurements dispatches. For all these task, there are several different design and implementation strategies.

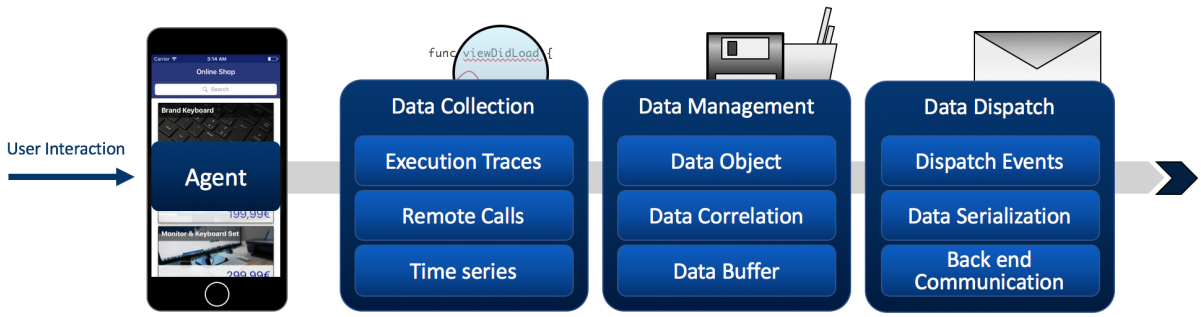


Figure 2.2: Flow of activities performed by the monitoring agent

Figure 2.2 illustrates the flow of activities in cases the user interacts with the application. In the first phase of the pipeline, the agent collects execution traces, recognizes performed remote calls and measures time series data. For some monitoring strategies, the agent has to reconstruct the execution trace from method invocation to method invocation. The data management phase provides this functionality. Creating measurement objects, correlating the measurement objects, in order to reproduce the execution trace, and buffering the data are part of managing the data. At the end of the pipeline, the monitoring agent waits for a dispatch event, serializes the collected data and sends them to a monitoring back end. The following sections demonstrate general approaches of the mentioned agent working phases by focusing on collecting and processing execution traces. Section 2.2.1 focuses on how to collect source code-based data. In Section 2.2.2, general approaches of managing collected data are explained and Section 2.2.3 focuses on data dispatch strategy.

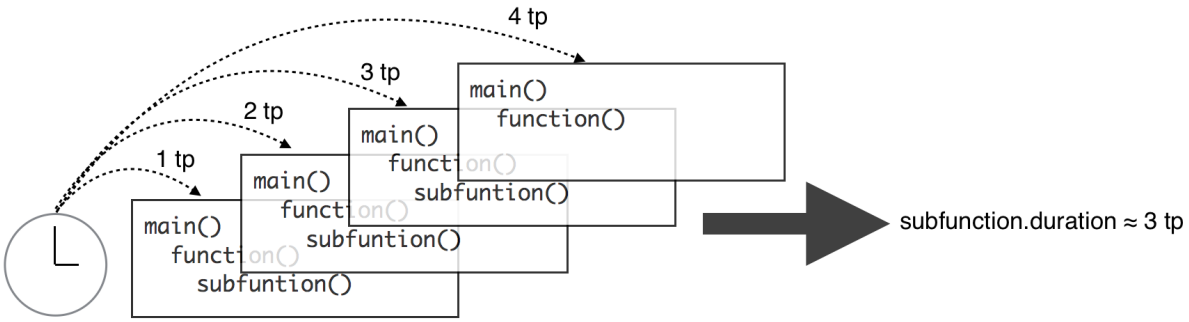


Figure 2.3: Simplified illustration of call stack sampling

2.2.1 Monitoring Strategies for collecting execution traces

Call Stack Sampling

The idea behind call stack sampling [BS14] is to run a timer-based process which has the task to collect the actual stack trace information of every thread. The collected data are going to be buffered and compared on the next iteration with the new stack trace. With the stack trace itself it is possible to rebuild the invocation sequence. By sampling and comparing the old stack trace buffer with the new one, the agent is able to calculate an approximate duration of invocations. For instance, if an invocation is tracked on three iterations on the same stack position, with the same parent invocations, and this invocation disappears on the fourth iteration, the agent assumes that this specific procedure's duration was at least three timer periods long. The higher the timer frequency is, the more accurate is the approximation of the execution time. On the other hand, if the timer periods are too short, the task does not have enough time to perform the buffering, the stack comparison, the invocation recognition and to set the invocation relations and their properties. In addition to that, with a frequency this small, the agent produces a huge overhead, which may affect the performance of the application negatively.

Tracing

This approach is used to instrument the written source code of the application. By manually invoking the agent at the beginning and the end of every method as shown in Listing 2.1, the agent can determine the start time, the end time and the execution time of any method.

Listing 2.1 Example of Tracing a Function

```
func doSomething() {
    // The agent starts tracking the doSomething function
    let invocationId = Agent.trackInvocation()

    // ... do something

    // The agent stops tracking the doSomething function
    Agent.closeInvocation(id: invocationId)
}
```

When a function ends, the agent sets the end time of the invocation, calculates the execution duration, by subtracting the start time from the end time.

Source Code-Level Instrumentation using AOP

The disadvantage of manual instrumenting function by tracing them, is the fusion of instrumentation invocations with the application functionality. This reduces the understandability and the maintainability becomes more difficult. The mentioned problem can be solved with aspect-oriented programming AOP, introduced by Kiczales et al. [KLM+97]. With AOP one is able to encapsulating cross-cutting concerns such as agent invocations, used for app monitoring, from core-level concerns but are automatically calling each other in the normal program flow [HKGH11]. This means that a method is wrapped when invoked and one is able to insert so called aspects to be executed before or after the method execution.

2.2.2 Data Management Strategies

Storing measured and collected data on the mobile devices from the application agent is not a trivial task performed by the mobile agent. One possibility is to save all measured objects on RAM as invocation, remote call or metrics measurements. This approach could cause application problems such as an overflow of memory, if we consider that the application developers application can scale as desired. We have to consider that low memory on a mobile iOS device causes an application crash [APL217]. The operation system gives the developer a chance to deallocate objects, by directly calling the `didReceiveMemoryWarning` [APL317] function on low memory. We could use this delegate to manage the collected data and remove unimportant information to clear the memory or to transfer objects on the hard disk. A disadvantage of this approach may be

the performance overhead generated by iterating through all objects and by removing or passing them to another storage. Another disadvantage is the memory overhead itself, which could cause content loading problems in the context of user experience. For reducing memory overhead on the mobile device we have to mention the approach of not saving all measured data. We could analyze the gathered information directly after the collection and store it if important and revert it otherwise. Another aspect to be considered is the retention of incomplete measurements. Incomplete measurements are caused by an unusual application termination such as crash or the end user terminated the app before the actual scope of the application ended.

Management processes are the next steps after data collection and retention. Data management is considering the data model of collected measurements, and the maintenance of the agent data. The data model represents the blueprint of agent measurements. We have firstly to consider approaches such as raw object storage and data serialization. Afterwards we have to analyze the management performance of such object types in terms of data comparison, data filtering and data size. These mentioned data tasks are important due to limited device memory and limited mobile data volume. High memory objects may have more analysis information than low memory objects, but are allocating more memory and are increasing the network usage of the end user. We face the same issues with a huge set of data and a reduced one. A strategy for data filtering is to run a background process, which analyzes the collected measurements with predefined performance criteria and marks them as important measurements and unimportant ones. For instance measurements with detected software anti patterns are important measurements. In the case a measurement is unimportant, it can be deleted. This would save memory and reduce the network usage afterwards. We have to take in consideration that the filtering process should not affect the performance of the application processes in the foreground. Therefore, this approach could be problematic for applications with high usage of hardware resources.

2.2.3 Data Dispatch Strategies

Collected data needs to be sent to a back end for data monitoring and further data and performance analysis. In the previous section we mentioned the limited mobile network data volume of an end user. This task is not a problem in cases the user is connected via Wifi. Data packages could be sent without mobile data volume utilization. The main challenge of this agent task is to reduce the network usage overhead as much as possible, if the application user is not connected to the Internet via Wifi. On the other hand important measured data could be lost, when not sent, due to clearing the storage in cases the device has low memory. The agent has the possibility to wait until the user is connected via Wifi or to reduce the amount of data for the dispatch and to store the

rest on the hard drive. We have also to consider that some user may not be connected to a stable network at the moment the agent would be ready to send measured data. In this case the agent could store the mobile measurements and try the data dispatch later. This depends on how long the data are stored. For instance data collected months ago are not as important as actual measurements, therefore the agent could decide to clear the storage on this term.

2.3 Available iOS Application Monitoring Agents

Commercial Application Performance Monitoring tools like AppDynamics [APP17], New Relic [NEW17] and Dynatrace [DYN17] are supporting iOS mobile application monitoring. In the context of a development project [DEVP17] we implemented a mobile agent for the open-source APM tool inspectIT. The strategy used for the inspectIT mobile agent may differ from approaches used by other companies. The core of this thesis is to research strategies utilized by commercial tools, to create an overview of the used approaches and implement new iOS Agents, for evaluating the results on different kind of iOS applications. The following paragraphs list and describe commercial as well as open source mobile iOS APM tools with the focus on how these tools collect execution traces. Table 2.1 recapitulates the used concepts for monitoring execution traces.

AppDynamics Mobile Application Monitoring for iOS: AppDynamics [APP17] is providing an iOS Agent as a framework, written in Objective-C, that allows performance and activities monitoring of applications at runtime. To make the Agent operative, the framework has to be included in the project manually or by adding a CocoaPods dependency. AppDynamics offers the developer to manually instrument methods of the application. By instrumenting a method the agent counts the invocations and measures the execution time. Developers are able to instrument a method, by placing an instrumentation call, that starts tracing, at the start and one or more agent calls, to stop the measurement, at the exit points of the method. In addition, the agent allows to trace over more than one method. This feature can be used to trace a certain use case of the application. In the intervening time, the agent collects metric values such as carrier name, network connection, geolocation, application as well as device information and execution time. For monitoring system-based instructions or remote calls, AppDynamics uses method swizzling. The Agent is able to automatically detect and instrument HTTP requests done via `NSURLConnection` or `NSURLSessions`. To correlate back end traces with mobile traces (End to End Monitoring), the back end application adds and sends precalculated trace variables in the response header.

New Relic Mobile App Monitoring for iOS: New Relic [NEW17] offers their iOS mobile Agent as a framework, as was the case for AppDynamics. The Agent could be included in the project manually or by a CocoaPods installation. An automatic classes and methods instrumentation comes out of the box for some methods of classes such `UIViewController`, `UIImage`, `NSJSONSerialization` and `NSManagedObjectContext`. They allow users to trace self implemented methods, by calling the class methods `startTracingMethod` for starting tracing and `endTracingMethodWithTimer` for closing a trace. New Relic traces can be tagged with different categories, e.g., View Loading, UI Layout, Database, Images, JSON and Network.

Dynatrace iOS App Monitoring: Dynatrace [DYN17] offers their iOS mobile Agent as a framework, as was the case for AppDynamics and New Relic. The Agent supports a CocoaPods installation. The Dynatrace iOS Agent provides user-based data and recognizes new users, and HTTP request data such as the amount of calls, the error rate and the request time. The agent is also capable of reporting crashes of the application. They are offering a feature named Auto-Instrumentation. This feature traces automatically life cycle phases of the iOS application. The agent tracks for instance moments of starting the application, closing an application or loading a new view controller. This feature can be achieved by providing an instrumentation code injected application delegate file, which holds listener invoked when the application is entering in certain life cycle phases. In addition to that, they probably use method swizzling for instrumenting view controller based function such as the `viewWillAppear` method. Dynatrace's iOS agent collects the following application metrics:

- Agent Version
- Application Name
- Application Version
- Application Build Version
- Battery Status
- Connection Type
- Network Protocol
- Device Name
- Device Manufacturer
- Total Memory in megabytes
- Percentage of Free Memory

- Number of Running Processes
- The operating system
- Screen Resolution
- Device Orientation
- Device Carrier
- CPU Information
- Rooted/Jailbroken Device

Lightstep: Lightstep [LIG17] is providing an iOS tracer for tracing certain parts of a source code. This is based on the Opentracing [OPT17] framework, which offers to instrument code fragments as Spans. This framework is an open source project located on Github (https://github.com/lightstep/lightstep-tracer-objc) and can be imported to an application with CocoaPods. The Lightstep tracer communicates with the Lightstep collector back end to send and persist the collected data from the mobile device. To fulfill this task an access token is needed to have the rights for connecting with the collector back end.

inspectIT Mobile iOS Agent: In the course of a development project we implemented a monitoring agent for iOS applications. [DEVP17] The agent is implemented, as well as the other solutions, as a Singleton to provide one single instance to the application. The whole public API is provided by the Agent class. From here the developer can start as many spans as desired. Spans are an abstract representation of a certain amount of code. In the context of this specific agent one can see a span as a use case. Each span is identified by a unique 64 bit integer and by its name. In addition they store the identifier of the corresponding trace and parent. `UseCases` and `RemoteCalls` are used representations of a span. Those can be created, started and closed by the developer not only sequentially but also in parallel and they might be encapsulated (nested) in other spans. This allows the developer to start a span although another one is already running in the current execution. A graphical illustration of spans can be seen in Figure 2.4.

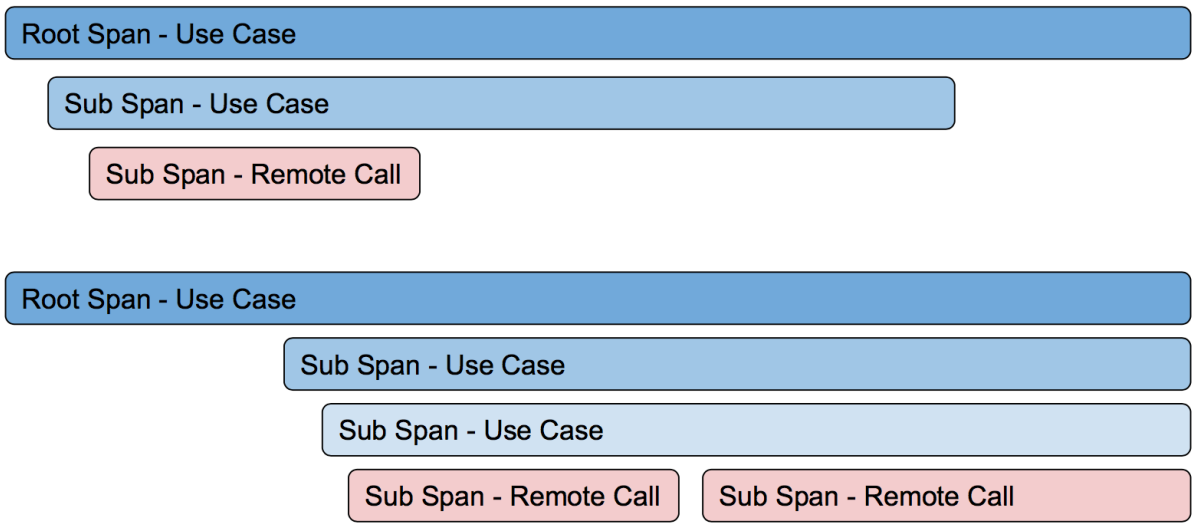


Figure 2.4: Spans example

To complete the mobile measurement, the agent also handles a pool filled with various metrical values of the device, such as CPU usage, memory and hard disk usage and battery power. The measurement values are retrieved in specific time intervals. By default the time interval is set to five seconds. The Agent also handles the serialization of gathered data, stores the result of executed use cases and prepares them to be sent to the monitoring back end.

Summary of the available iOS Monitoring Agents

This section reviews the available iOS monitoring agents in a tabular form with the focus on strategies for collecting execution traces.

	Tracing	Method Swizzling	Use case Mapping	Call stack Sampling
AppDynamics	X	X*	X	-
New Relic	X	-	X	-
Dynatrace	X	X*	X	-
Lightstep	X	-	-	-
inspectIT	-	-	X	-

Table 2.1: Used approaches for collecting execution traces of available iOS APM tools
X*: Only used for tracing system library methods

As shown in Table 2.1 most of the available monitoring tools are using the method level or the use case level tracing approach. Method Swizzling is used for monitoring system library based methods. For instance, for monitoring executed remote calls or the application life cycle. None of the mentioned iOS APM tools are using the call stack sampling approach for collecting execution traces.

2.4 Software Development Introduction for iOS

Since we will implement different mobile agent configurations, it is useful to have an introduction in the iOS development. This section will describe the most basic and important constructs for developing an iOS application. The focus of this section will be the programming language *Swift*. On top of that, when describing different data structure, this section will focus only on the ones that are used later on for the agent configurations. Swift forces the application developer to initialize attributes before using them. Attributes can be initialized as variables or as constants. A variable can be declared with the keyword `var`. A constant with the keyword `let`. Since Swift supports type inference there is no need to dictate the type of a variable. Listing 2.2 shows how to declare constants and variables.

Listing 2.2 Declaring Attributes in Swift

```
// Declaring a constant
let a = 1

// Declaring a variable
var b = 1
```

If the developer wants to force a different class type for a variable, he is able to do it by using the `:` syntax. By default, passing an integer numeric value will assign the `Int` type. Listing 2.3 shows a forced type assignment. Instead of an integer, the constant a is now an unsigned integer.

Listing 2.3 Forced Type Assignment

```
// Declaring a constant as unsigned integer
let a: UInt = 1
```

As mentioned before, before utilizing an attribute, the attribute has to be initialized. In cases we not able to do this, and we want to assign an attribute no value, the attribute can be defined as an optional with the `?` syntax. [SOP17]

Listing 2.4 Declaring an Optional

```
var a: Any? = nil
```

Methods known in the programming language *Java*, are named functions in Swift. In order to declare a function the developer has to write an optional modifier, the function identifier, an optional argument list and an optional return type. Listing 2.5 shows an example of declaring a function in Swift. The argument list is given within the round braces and the return type exists with the `->` syntax. [SFU17] One may noticed that the declared function was also called in line 8.

Listing 2.5 Declaring a Function (Factory Example)

```
func factory(n: Int) -> Int {
   if n < 0 {
      return -1
   } else if n == 0 {
      return 1
   } else {
      return n * factory(n: n - 1)
   }
}
```

In the following, we will introduce the data structures utilized for the mobile agent implementations. The following parts focus on arrays and dictionaries. Initializing an array as a variable means that the object is mutable. In other words, a variable array is a mutable list. In order to initialize a real array, the developer has to declare the attribute as a constant and define the array length when initializing the object or passing an entire array as shown in the example. Listing 2.6 illustrates how to firstly initialize a list and how to create an array. As shown in below, in order to declare a list or an array the `[]` syntax is needed. Within the braces the developer has to define the array type. In our example we created integer arrays. Array values can be retrieved and set through subscripts. The subscript type for an array is an integer, which is correlated to the position of a certain element in the array. For instance `array[0]` returns the first value of the array and `array[0] = 1` sets the first value of the array to 1. [SAR17]

Listing 2.6 Declaring Arrays

```
// Declaring a list
var list = [Int]()

// Declaring an array
let array = [1, 2, 3, 4]
```

In the scope this thesis, we also required a dictionary in order to organize collected data. A Swift dictionary is mutable. In order to create a dictionary, the developer has to define the key and value type of the dictionary. In the example below, Listing 2.7, we declared a dictionary with `String` as key type and `Int` as value type. Similar to an array, a value can be retrieved through the key. For instance, `dictionary["a"]` returns the value, if existing, mapped for the string "a".

Listing 2.7 Declaring a Dictionary

```
var dictionary = [String:Int]()
```

Chapter 3

Application Classification

This chapter presents differences between mobile applications. Additionally, it will argue if an application can be distinguished between other ones by the application category, the application architecture, the resource usage or the used system frameworks. The mentioned points are often coherent to each other. At the end of this chapter we will list the resulting application classes with their respective descriptions.

3.1 Application Categories

The most naive approach to classify applications would be to group application categories, since categories themselves are defined to group different apps dependent on the core concerns. Table 3.1 presents a set of application categories selectable from the developer for the implemented and publishing project provided by Apple [APC]. This section will review each category and research the average resource and network usage. The resource usage will be split in CPU, Memory and disk or database usage.

Books	Food & Drink	Medical	Reference
Business	Games	Music	Shopping
Catalogs	Health & Fitness	Navigation	Social Networking
Education	Lifestyle	News	Sports
Entertainment	Kids	Photo & Video	Travel
Finance	Magazine & Newspapers	Productivity	Utilities

Table 3.1: iOS Application Categories

Classifying mobile applications by their category, selected when deploying the application on the AppStore, will not result as a proper classification strategy. In the following we will demonstrate an example which proves that applications located in the same category set, often have different architecture properties and hardware resource usages. For this instance we chose the first category *Books*.

This category includes apps visualizing usual printed contents. Other apps included in this category are book portals such as, e.g., Audible which are also providing audio-book in a streaming format. In the most cases these apps are not high power consuming. Down-loadable reading content might be buffered in RAM or stored on disk. While e-books' average file sizes are around 2-3MB, audio-books recorded in high quality and published on the Audible platform could amount up to 28MB for an hour of audio [AUDA17]. For instance the popular audio-book *Harry Potter and the Goblet of Fire* has a recording length of 21 hours and 12 minutes [AUDB17]. The approximate file size of this book would be 610MB. Requesting files that big would highly affect the memory usage of the mobile device network connectivity.

Summary: Considering the mentioned instances, we are not able to attribute a class for all applications of a category. In the case of *Books* apps, the absolute memory and network usage depends on the amount of the requested files, which can fluctuate from a small amount to a huge amount depending on the usage of the user and depending on the core of the program. In conclusion we have to consider the application architectures themselves and focus on underlying system frameworks, to be the more accurate in defining mobile application classes.

3.2 Mobile Application Architecture

For developing a mobile application there are three main options. The first is to build a full native application. This option requires from the developer team a certain know how of the programming language, in this case *Swift* or *Objective-C*, used for build native applications. The advantage of using this option is that the programming languages and the provided frameworks are optimized for the specific operating system and are therefore in the most cases more performant. If the mobile application requires to access to hardware components such as the accelerometer or the built-in camera, there is no option to avoid a native program. An application ready to deploy, has to be uploaded to the related application store. The pipeline for uploading an application to the Apple AppStore requires an application to pass a review before the will be published application. In general the duration of the deployment process is in between two or five days, depending on how many applications have to be reviewed [APR17]. The same

process is also performed when the application has to be updated. Native applications itself is not suitable to consider as an application class in our context, since there are further differences between them in terms of performance usage. As mentioned in the section above, focusing on used system frameworks is essential for defining native application classes. An other option is building a web application. A web application is callable through a browser from each device, nonindependent which operating system is running. Common technologies used for modeling the application front end are HTML, CSS and JavaScript. The advantage of building a web application compared with implementing a native one lays on its portability and deployment velocity. Since native applications have to pass a review which takes several days to conclude, web applications can be deployed instantly by using other servers instead of the Apple AppStore platform. The main focus of this work is to evaluate monitoring strategies for native iOS mobile applications, therefore we will not take in consideration this application class. The last option is to build an hybrid application. By implementing hybrid applications, the developer is able to take the advantages of web applications and to bridge them into a native application.

In the following sections we focusing on three main native application classes. Section 3.2.1 explains the foundations of hybrid applications and describes the necessary components. In Section 3.2.2 we focus on native clients of a distributed system, and in Section 3.2.3 we review stand alone mobile applications.

3.2.1 Mobile Hybrid Applications

As mentioned above, by implementing hybrid applications the developer team is able to distribute the complete system and to take the advantages of native and web applications. For instance components which are expected to be updated often, may be migrated into a web application. Figure 3.1 illustrates a model of an hybrid application sample. The sample application includes native components at the top of the screen, which may allow the user to navigate through the app for instance by searching an article. The native application is also able to request and display a web application through a the web view, which is located under the native components in our example. A web view basically has the functionalities of an embedded browser. Apple provides the classes `WKWebView`, `UIWebView` and `WebView` for embedding web content in a native application. Considering monitoring hybrid applications, instrumenting web view based methods such as `load(Data, mimeType: String, characterEncodingName: String, baseURL: URL)`, `load(URLRequest)` and ones used for navigation is essential. Therefore we have to analyze the `WKWebView` class and its instance methods, for selecting the appropriate functions that should be instrumented out of the box by implementing a wrapping preset. For assuring the instant instrumentation without work overhead,

Figure 3.1: Hybrid Application Sample

the preset is made of a set of swizzled functions and it has to be integrated in the agent configuration. Method Swizzling is described in detail in Section 5.2.2. In the following, we are describing the functionalities of a `WebView` instance. Additionally we are mentioning methods which has to be instrumented to obtain enough diagnosis information for further analysis in cases of raised errors.

After instantiating a web view through the storyboard or programmatically, we need to load the web content. The developer is able to choose between loading local files, loading contents from the web or loading a web page by passing an HTML string. The methods providing loading content for web views are `loadFileURL(URL, allowingReadAccessTo: URL)`, `load(Data, mimeType: String, characterEncodingName: String, baseURL: URL)`, `load(URLRequest)` and `loadHTMLString(String, baseURL: URL?)`. The WebKit framework also provides functions for reloading contents such as `reload(Any?)` and `reloadFromOrigin(Any?)`. Additionally the developer is able to stop the loading process by performing `stopLoading(Any?)` [WKW17]. All iOS web views contain a UI delegate and a navigation delegate. The UI delegate provides the functionality of for presenting native user interface elements regarding to the web page. For instance `webView(WKWebView, runJavaScriptAlertPanelWithMessage: String, initiatedByFrame: WKFrameInfo, completionHandler: () -> Void)` displays a JavaScript alert view [WKU17]. The navigation delegate contains functions that are triggered at specific navigation occasions. If the load progress of the web page is complete, the delegate will fire the `webView(WKWebView, didFinish: WKNavigation!)` method. Navigation errors can be caught by overriding `webView(WKWebView, didFail: WKNavigation!,`

withError: Error) and loading failures can be recognized when webView(WKWebView, didFailProvisionalNavigation: WKNavigation!, withError: Error) is invoked [WKN17].

3.2.2 Native Client of a Distributed System

A mobile hybrid applications, which was described in Section 3.2.1, is a special type of a distributed system. While hybrid applications contain one or more web views, native clients do not. This means that the application is able to request data from other distributed systems, but it is only allowed to render them using native user interface elements. Since distributed systems are made off more than one system, communication between the different peer instances is an important task of applications of this type. For instance the mobile client may requests data from an external server or performs a web service. In consequence, this section has to concentrate on possibilities of performing remote calls. In our case a remote call is a URL request. In the following we will describe options for performing remote requests for iOS applications. For performing URL requests on iOS mobile devices, the developer need to instantiate an NSURLRequest object and send the request through an URLSession. The NSURLRequest instance encapsulates the request URL and the behavior to use when with cached responses. In Figure 3.2 the NSURLRequest class is specified in a class diagram. The httpMethod attribute stores HTTP request type. Dependent on the request task, the developer is able to assign GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE and PATCH as a string to the httpMethod attribute. The request URL is hold and provided by the url attribute. The data, which has to be provided to the receiver in order to get a response are stored in httpBody. The application developer is also able to set a request timeout by setting a time interval for the timeoutInterval attribute [URR17]. TimeInterval is public type alias for Double. The value(forHTTPHeaderField: String): String returns the value for a certain header attribute, which has to be passed by an argument. As one may notice NSURLRequest only provides to edit the HTTP body and only allows to read header fields but not to set ones. If the developer needs to add or set header variables, the developer has to use an NSMutableURLRequest instance. NSMutableURLRequest is a subclass of NSURLRequest, which adds the functionality to add header attributes. Figure 3.2 also shows the inheritance of NSMutableURLRequest. Header attributes can be added or set by calling addValue(String, forHTTPHeaderField: String) or setValue(String, forHTTPHeaderField: String). The first argument will be set as value for the header attribute passed by the second argument. The constraint for adding new header attributes is not to use Authorization, Connection, Host, Proxy-Authenticate, Proxy-Authorization and WWW-Authenticate as attribute identifiers [MUR17].

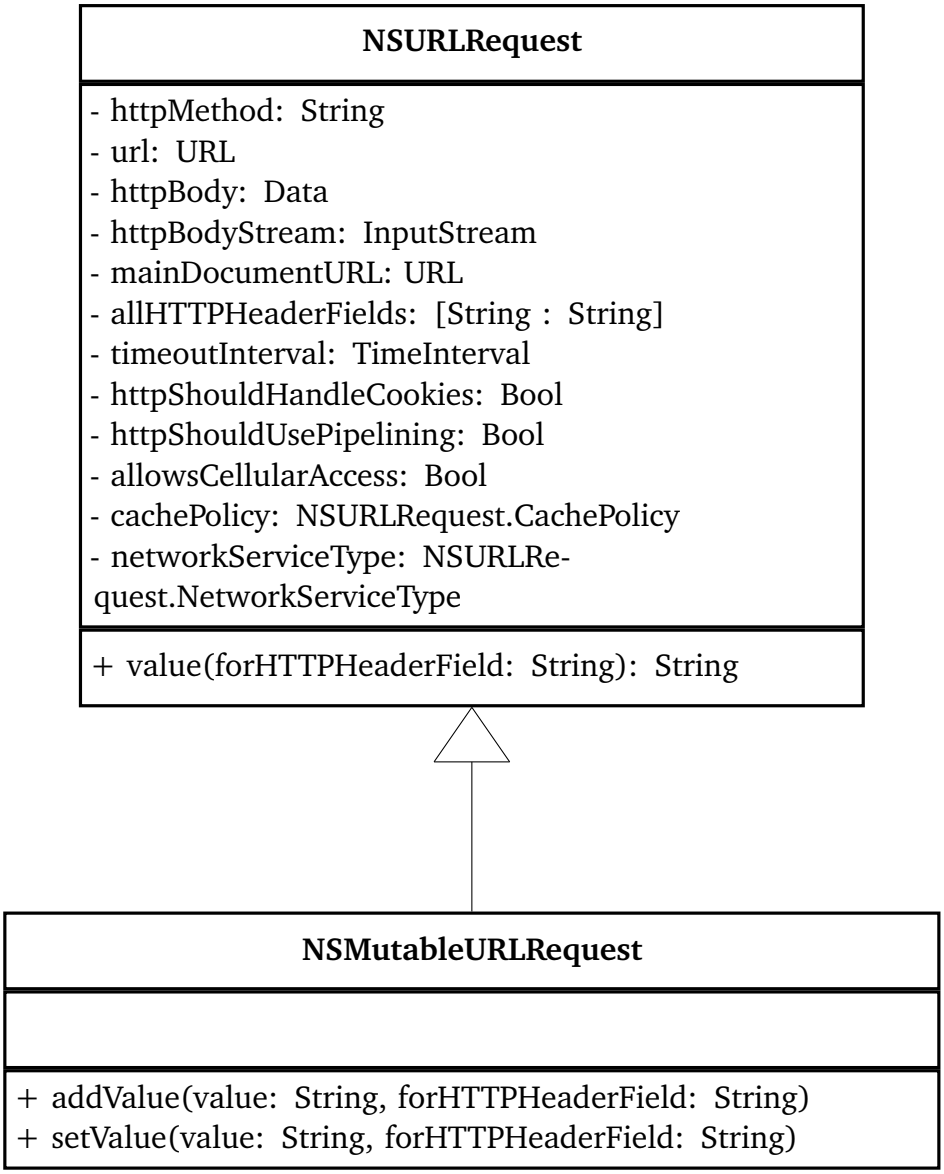


Figure 3.2: NSURLRequest Class

The next step, after setting the URL request, is to pass the NSURLRequest instance through a URL session. The `URLSession` class and the related classes are providing the functionality of requesting data from a back end and provides various download possibilities of the URL responses [USE17]. The `NSURLRequest` instance need to be converted into a data task. This operation is performable by calling `dataTask(with: URLRequest, completionHandler: (Data?, URLResponse?, Error?) -> Void)`. The passed URL request will be converted in a data task. Additionally, the developer is able to pass a callback function which will be called in the case the URL session receives a complete response from the back end. Created data tasks start in a suspended state. For changing

the state to active, the task need to be started by calling `resume()`. After starting a task, the URL session calls various delegate methods step by step. If a connection-level challenge is required, when performing the first handshake with the back end, the `URLSession` instance will call `urlSession(_:task:didReceive:completionHandler:)` or `urlSession(_:didReceive:completionHandler:)` on its delegate for requesting credentials. During the upload of the request body to the back end, a routine calls periodically `urlSession(_:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:)` for reporting the actual progress of the upload operation. When the server responds, the mobile application receives an initial reply from the back end. In that case the session calls `urlSession(_:dataTask:didReceive:completionHandler:)`. During the download of the requested data, the `urlSession(_:dataTask:didReceive:)` displaying the actual fragments of the requested data. When any kind of task is finished, `urlSession(_:task:didCompleteWithError:)` will be called to alert the end of the remote call. If an error occured during the download process, `didCompleteWithError` stores the error to catch the problem, otherwise it is set to `nil`. Additionally, we have to mention the existing different types of tasks. The process explained above is generally provided for a `URLSessionDataTask` [USD17]. A data task returns the downloaded data to the app memory. The `URLSessionDownloadTask` class is also provided. A download task normally wraps the downloaded data into a file. In addition, it has the ability to check the download status by comparing the written bytes with the expected amount of bytes [USL17]. The mentioned method is called periodically automatically from the URL session delegate. The last URL session task is the `URLSessionUploadTask`. The upload task also provides a function to check the current status of the upload. Developers are able to retrieve the upload status by overriding `urlSession(\_:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:)` [USU17]. As soon as the requested data arrive, the specific data task will perform its completion handler. The completion handler receives the remote call response as a `URLResponse` object. By instrumenting the mentioned instance methods of the different classes used to perform HTTP requests, the agent is able to recognize remote calls, to measure remote call-based metrics such as the upload or download status, to notice whether the remote call was timed out, to check the response code and to intercept occurred errors.

3.2.3 Standalone Mobile Application

A standalone mobile application is not dependent from an external back end to provide the core concerns. In the most cases these types of applications neither require a network connection. An instance of this application class is a calculator app. A calculator has no need to request specific data from a server. The core concerns of the application are all stored locally. An other instance is a to-do application without sharing functionalities.

The contents of the application are managed and stored locally. In some cases standalone mobile application are providing a local database to executing the mentioned tasks. Apple provides the `CoreData` framework for handling persistent object storage. Standalone apps could also display advertisements. Apple declared the iAd App Network form December 31 2016 as no longer available, and suggests to rely on third party networks and advertising sellers for this task [IAD17].

3.3 Summary

Since we focus on native iOS applications, for the further course of this thesis we will consider the following application classes:

- Hybrid Application
- Native Client of a Distributed System
- Standalone Mobile Application

The main difference between the listed application classes depends on the underlying system libraries. Each class uses different system classes than the other. As a result the agent has to be capable of monitoring the respective system methods. The next chapter argues in detail with the requirements of a monitoring agent.

Chapter 4

Requirements for Monitoring Agents

This chapter lists the requirements of an agent configuration and the prerequisites to perform the evaluation. Additionally, we will present a detailed view of our work program. Our work program consists of theoretical work packages and practical work packages. To fulfill a certain work package, various tasks need to be performed. Some tasks are dependent from other ones and need to be completed first. The detailed view of our work program includes the listing of the work packages, the presentation of the tasks and their aims of each work package and the dependencies of the tasks and of the work packages.

4.1 Software Requirements Specification

As illustrated in Figure 2.2, in order to monitor an agent-based instrumented application the agent requires at least three main working stages. Collecting monitoring data is the primary working stage of an agent. Monitoring data consist of runtime source code-based information such as method invocations with their invocation time and execution time. Further on time-series measurements of hardware resource usages such as CPU, memory and network usage are also included in monitoring data. The working phase after collecting data is managing data. Data management is considering the data model of the collected measurements, and the maintenance of the measured data. The last work phase of the agent is the data dispatch. Collected data needs to be sent to a back-end to be displayable on a monitoring client and for a further data and performance analysis.

4.1.1 Requirements of a Mobile Agent Configuration

Since we want to evaluate monitoring strategies for native iOS applications, we also need to consider mobile device specific metrics such as the network connection, the carrier name and the battery status. We also have to mention that in general mobile devices are less powerful in terms of hardware specification than personal computer or servers. Therefore the agent concepts and implementations needs to be as performant as possible in the context of memory usage and running time. The following subsections will present lists of functional, non-functional and external interface requirements.

Functional Requirements

FSR1 The agent collects data

- FSR1.1 Agent collects invoked source code-based data
- FSR1.2 The agent stores the timestamp referring to its method invocation
- FSR1.3 Remote calls are recognized by the agent
- FSR1.4 The agent tracks the users location when he allows it
- FSR1.5 The agent collects metric values with a timer-based process in a specific frequency
- FSR1.6 The collection frequency of the timer-based process should be editable by the developer

FSR2 The agent manages data

- FSR2.1 The agent reconstructs the right execution tree with the collected data
- FSR2.2 The agent buffers the collected data locally
- FSR2.3 Monitored back-end traces working with the Opentracing structure would recognize the caller invocation

FSR3 The agent sends the managed data to an APM back end

- FSR3.1 The back end address can be edited
- FSR3.2 The agent sends time-series measurements and trace information in a proper format
- FSR3.3 The agent has a unique ID

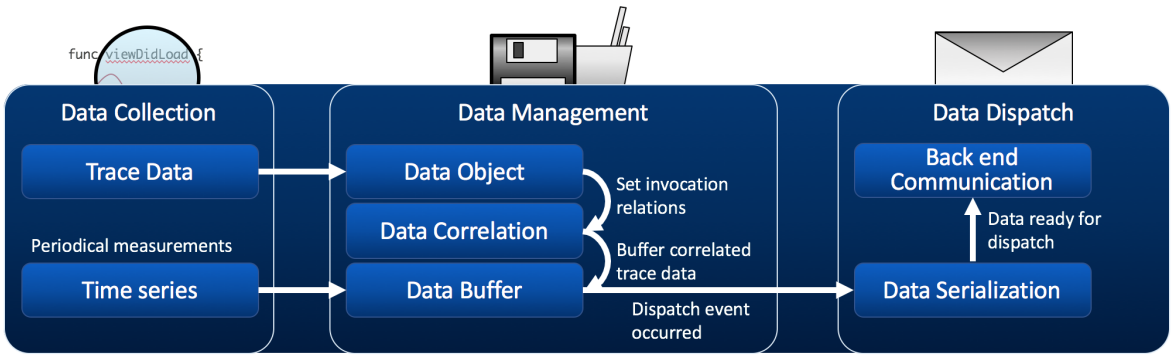


Figure 4.1: Monitoring Agent Activity Pipeline

Figure 4.1 illustrates the expected activity of the monitoring agent in detail. First of all, the agent has to collect trace data. Trace data comprehends the invoked source code data and the respective execution time. Remote calls are also included in trace data. This agent task has to fulfill FSR1.1, FSR1.2, FSR1.3 and FSR1.4. In order to monitor the hardware resource workload, the agent performs periodical measurements of the hardware metrics. In addition to that, the collection frequency has to be editable in cases the application developer requires more data in a smaller time interval. When trace data is measured, the agent creates a data object and correlates it with the caller object. This produces an hierarchical structure of trace data and ensures that the agent reconstructs the right execution trace. At the end, the collected monitoring data has to be send to a monitoring back end for further analysis. Therefore, the agent needs to serialize the measurements in a proper format. Since multiple agents are spread for the same application, the agent has to provide an id in order to distinguish the send monitoring data.

Non-Functional Requirements

NFSR1 The performance overhead produced by the agent is minimal

- NFSR1.1 The operational times of the agent tasks are as low as possible
- NFSR1.2 The memory overhead produced by the agent is as low as possible

NFSR2 The agent does not affect the functionality of the monitored application

NFSR3 The agent does not affect the usability of the monitored application

4.1.2 Optionals of an Agent Configuration

Optional requirements may improve the functionality of an agent configuration, but are not necessary for the base tasks. An additional module for an agent configuration might be a built in data analysis. This intelligence could be able to analyze the collected data locally on the mobile device and determine whether some collected and stored data are important and useful or not. For instance if the device runs out of memory due to the huge amount of collected measurements, the agent could start an analysis process which deallocates unimportant data from the memory. An other optional might be the integration of proactive elements. While the agent collects data, its intelligence could detect a specific error type. In this case the agent could notify the end-user by displaying an alert mentioning the occurring error type. This may improve the end-user experience and the mobile application user might be able to fix the problem himself. For instance if the user requests some content in a problem zone, the user would be aware of this problem due to the proactive error notification. At that point the end-user could solve this problem by moving to an other position with a better network connectivity.

4.1.3 Required Metrics

For a further analysis of performance problems, the agent has to provide a set of important metrics. In this section we will mention mandatory and optional metrics collectible from iOS mobile devices by an agent configuration running within an application. In addition we will differentiate between execution information and time-series measurements.

Trace Data (Single Measurements)

A number of bad performance in software systems is often related to high execution times. Application monitoring tools have to recognize these kind of performance problems and have to visualize them to make the application analysts or developer aware of performance issues. To accomplish the mentioned awareness, it is important that the mobile application agent measures the execution times while tracking method invocations or remote calls. In the context of this thesis, the iOS mobile agents should be able to detect method invocations, defined use cases and performed remote calls as spans. Since each of the three span types have different attributes which are important for their respective span type, we decided to split this sub section to argue with the different metrics separately.

Method invocation: Important attributes to be collected while tracking a method invocation are the execution times, the method name and the span correlation with other spans. The method execution duration and the method name are important for identifying and analyzing slow methods. Recognizing the span correlation is important to detect relations in different method calls. In order to assure the span correlation, we decided to use the opentracing strategy. Spans the opentracing format, are holding three different ids in order to establish the relations: id, parent id and trace id. The parent id references to the span id of the span which invoked the current span. The trace id references to the execution trace the measured span is related to.

Metrics	**Mandatory**	**Optional**
Execution Time	X	
Method Name	X	
Correlation IDs	X	
Thread ID		X
Thread Name		X

Table 4.1: Mandatory and Optional Metrics for Method Invocations

Table 4.1 displays mandatory metrics and optional metrics for method invocations. The mandatory metrics have been already explained in the section above. Optional metrics are the thread name and the thread id. Since collected spans are organized in different execution traces, it is not required to additionally store the thread information of the spans.

Use cases: The execution time of a use case is as important as the duration for methods invocations. Instead of method names, the use case structure has to store the defined use case name. As shown in Table 4.2 it is not essential to additionally store the method names, where the use case started or ended considering that these information are not essential for recognizing performance problems. Use Cases might be related to other ones, therefore it is important to store the correlation attributes.

Metrics	Mandatory	Optional
Execution Time	X	
Use Case Name	X	
Correlation IDs	X	
Thread ID		X
Thread Name		X
Method Names		X

Table 4.2: Mandatory and Optional Metrics for Use Cases

Remote calls: When requesting a back end, many unknown problems could occur. A problem instance is an high loading time when requesting required application content. In order to firstly detect high loading times, the mobile agent has to measure the duration between the request and response time. In the case the agent detects a high loading time, it is still possible to understand the reason for this problem. In consequence the agent has to store more information about the remote call. Problems for high loading times could be that the user is requesting content from a problem location, that the end user is not connected to the network, that the user tries to request content from a problem router or the back end system is slow.

Metrics	Mandatory	Optional
Duration	X	
Request URL	X	
Geo-location	X	
Network Connectivity	X	
Mobile Provider	X	
Router SSID	X	
Response Code	X	
Correlation IDs	X	
Thread ID		X
Thread Name		X
Method Names		X

Table 4.3: Mandatory and Optional Metrics for Remote Calls

In order to cover all the mentioned cases, the mobile agent has to collect additional information about the geo-location, the network connectivity and the used provider of

the end users device, and how the back end responded, as shown in Table 4.3. Apart from that, it would be useful to store the request URL, in cases the back end is slow.

Time-series Measurements

Time-series measurements are repeatedly collected measurements in a certain frequency. Each measurement point stores its collection timestamp. This subsection presents the collectible mobile device hardware metrics in a tabular form and explains whether a certain metric is necessary to be collected from an agent or not.

Metrics	**Mandatory**	**Optional**
CPU usage	X	
RAM Memory usage	X	
Disk Memory usage	X	
Battery usage	X	
Geo-location		X

Table 4.4: List of Mandatory and Optional Metrics

One of the main task of an application monitoring tool is to recognize performance problems in a running application. Another reason for slow applications is high usage of hardware resources. In cases the device runs out of memory the running application crashes. As a consequence the agent has measure the hardware resource workload, in order to monitor these data. These metrics values are measured periodically in a specific time. This helps to relate an increase of hardware workload, which probably provocates a problem, to a performed method. Since the geo-location is only important for tracking remote calls, it is not necessary to retrieve this information periodically.

Chapter 5

Architecture of Monitoring Agents

This thesis investigates which agent properties fit best to a certain application class. For answering this question it is necessary to define which monitoring concepts are possible to model for each mobile agent phase. This chapter describes and explains the main functionalities available for monitoring an iOS application. It will start by presenting the regular working pipeline of a monitoring mobile agent. The following sections describe possible concepts of each agent working phase in detail in terms of functionality and idea. The agent working phases are illustrated in Figure 5.1. The first phase concerns collecting monitoring data. This chapter focuses on source code-based data. Managing the measured data is part of the second work phase of the agent. The last part concerns deploying the buffered information and to send them to a monitoring back end, in order to be further used for the monitoring client. Section 5.2 names and explains all performance data collection strategies. The processing and management concepts for collected measurements are presented in Section 5.3. Section 5.4 describes sending strategies of buffered traces and device metrics measurements. Additionally, reasons of using certain data structure and algorithms will be discussed.

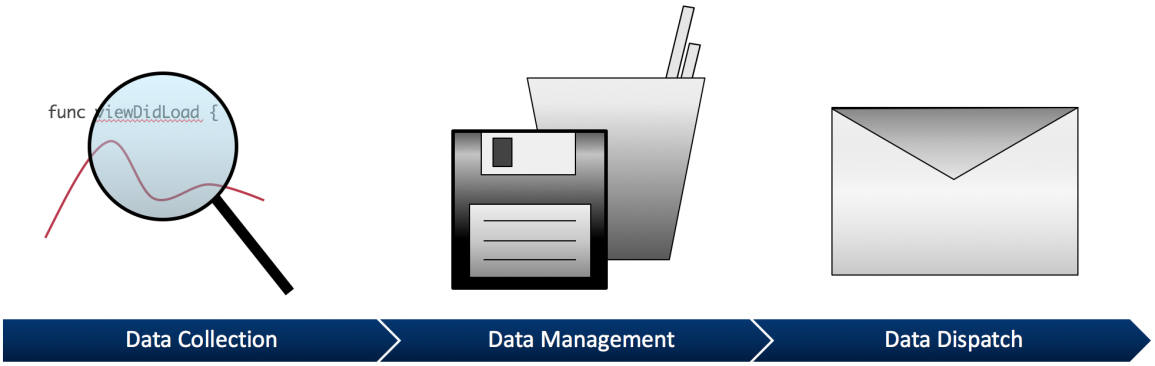


Figure 5.1: Agent Working Phases

5.1 Mobile Agent Pipeline

This section describes the functionalities of a mobile agent in general. Figure 5.1 presents the pipeline of the working phases of the agent. Collecting fundamental analysis data such as execution traces and metrics measurements is the principal task of an agent. The second work phase is the data management, where the agent buffers the collected data locally and merges correlated traces. The last stage is dispatching the managed data to an APM back end.

Optionally, an agent may include an analysis module which is able to filter out unimportant data. The analysis module may integrate a set of rules helping detecting performance problems based on the collected performance measurements.

5.2 Data Collection Strategies

This section lists performance data collection approaches for monitoring agents. Performance data need to be separated in source code-based data, resource usage and device information. Important for source code-based data are at runtime executed methods. The agent needs to recognize the executed method, to buffer the method name and invocation time and to measure the duration of the executing method. These properties are needed for a proper presentation of the execution traces on the APM monitoring client. Instead of instrumenting methods the agent could instrument a complete defined use-case of the mobile application. In terms of application monitoring a use-case may be formed in one or more method executions. Both options require from the agent to recognize an instrumentation starting point and an instrumentation end point.

5.2.1 Tracing on the Source Code-Level

This approach is used to instrument the written source code of the application. By manually invoking the agent at the beginning and the end of every method, as shown in Figure 5.2, the agent can determine the start time, the end time and the execution time of any method.

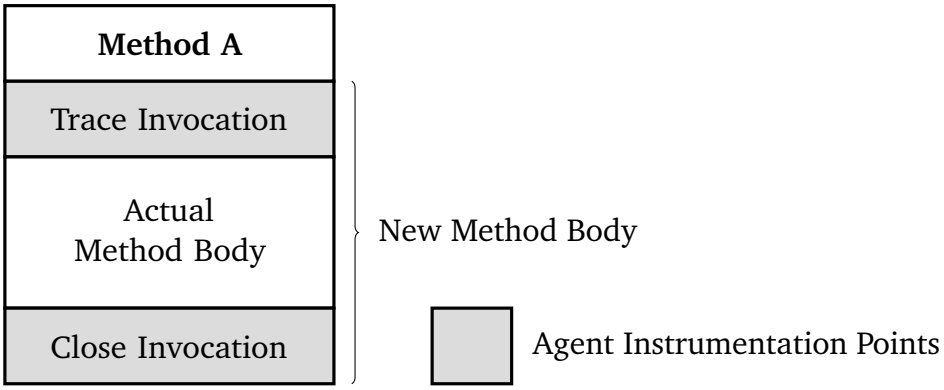


Figure 5.2: Agent Invocations in Method Body

By tracing a method invocation the agent should store the method name and the actual timestamp for remembering the execution start-time. Closing an invocation by the agent means to collect again the actual timestamp which defines the end-time of the method execution. The calculated difference between the two collected timestamps defines the execution duration of the instrumented method. Relations between traced invocations may be established though ids. Each measured invocation generates a unique id. For recognizing encapsulated calls the agent has to correlate the caller and the called method. This is realizable by adding a parent id attribute to each invocation measurement. The parent id relates to the id of the caller invocation. By using this approach the application developer has to deal with several problems. The first problem to mention is the manual setting of the instrumentation end-points. Methods may include return statements in their method body, therefore the termination of the method is not always at the end of the method body. To solve this problem the application developer needs to recognize the method termination points and to add the instrumentation end-points right before them. Otherwise an agent working with this approach would not be able to recognize the end of certain methods. Due to the fact that the developer necessarily adds the instrumentation points before return statements the agent would not be able to recognize recursive functions, calling themselves in return statements, without additional input.

5.2.2 Source Code-Level Instrumentation with Method Swizzling

One disadvantage emerging with manual source code-level instrumentation with tracing is that monitoring invocations are merges with the application code which reduces the understandability of the program code. Using aspect-oriented programming (AOP) would solve this problem. With AOP one is able to encapsulating cross-cutting concerns such as agent invocations, used for app monitoring, from core-level concerns but are automatically calling each other in the normal program flow [HKGH11]. In

Objective-C or Swift aspect-oriented programming (AOP) is realizable through Method Swizzling. Method swizzling allows application developer to swap the implementation of a certain selector with another one [NK14]. One could use this advantage to wrap methods he wants to instrument. This is performable by creating a new method which calls the original implementation. The new method can be injected with provided calls such as start tracing and end tracing to be invoked before and after a the real function call. After swizzling two methods, the implementation of the new one is executable by calling the old signature and vice versa. For instance we swizzle the `didReceiveMemoryWarning()` function which is called by the application on low memory, with `instrumentedDidReceiveMemoryWarning()` function. Assuming that the state of the device is on low memory, the application will call `didReceiveMemoryWarning()`, but the implementation of `instrumentedDidReceiveMemoryWarning()` will be executed. In this case we could inject the instrumentation calls in the method body of `instrumentedDidReceiveMemoryWarning()` as shown in Figure 5.3.

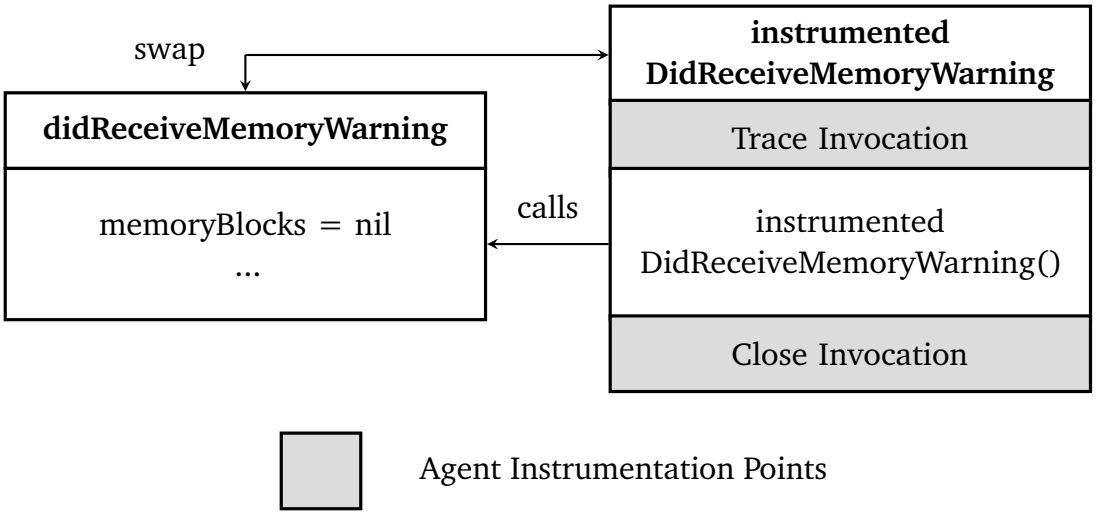


Figure 5.3: Method Swizzling Usage for Instrumentation

Method swizzling is programmatically achievable by creating two selectors holding both function names. Afterwards we have to distinguish whether we want to swizzle instance or class/static methods. Swizzling instance methods requires to retrieve the method implementation with the public `class_getInstanceMethod` method. Class methods implementations are accessible with `class_getClassMethod`. At last the implementations need to be swapped with the `method_exchangeImplementations` method.

5.2.3 Source Code-Level Instrumentation with Use Case Mapping

This instrumentation concept is similar to the source code-level instrumentation of invocations strategy. This approach is also used to instrument the written source code of the application, by manually invoking the agent. The difference between both strategies is the different mapping of the gathered data. While the source code-level instrumentation of invocations strategy creates invocation objects based on a single method invocation, the source code-level instrumentation with use case mapping creates span objects, which can cover more than a single method call. In general one can see a span as a use case. Since more than one use case can exist at the same time, a single stack is not sufficient. A dictionary, which maps the id of a root span to its span stack, is required. Starting a span as a root span requires only a name. Starting it as a sub span, the agent needs to know, which span is its parent and its root span, to be mapped on the right position. This is the reason, why span objects can be created, started and closed by the developer not only sequentially but also in parallel and they might be encapsulated (nested) in other spans. This allows the developer to start a span although another one is already running in the current execution.

5.2.4 Call Stack Sampling on iOS

The Problem with call stack sampling on iOS, is that there is no native method, which returns the call stack of all threads for iOS such as `Thread.getAllStackTraces()` for Java. Instantiating a scheduled timer for the sampling, returns a timer object, which executes a specific block of code on a sub-thread. The computed property `Thread.callStackSymbols` returns only the call stack symbols of the timer invocation thread, which does not help for further analysis. One may can try to force the timer invocation to run on the main thread with `DispatchQueue.main.async`. Letting the timer perform the code block on the main thread actually works, the agent misses all invocation information from all other threads. Therefore the needs an other approach for collecting all data. Since the agent has only the opportunity to get the stack trace of the current thread he was invoked from, we need to implement a routine which forces all threads on their own to call `Thread.callStackSymbols` periodically. The agent has to fetch all current running threads. Afterwards, the agent iterates through all threads and sets a signal action for each thread. A signal action is a callback method which is performed when the thread receives a specific signal. The callback method will be invoked on the same thread, the signal was sent to. This means that if thread 1 sends a signal to thread 2, the signal action will be called on thread 2. In our approach the signal action retrieves the stack trace of the current thread by calling texttttThread.callStackSymbols. Afterwards, the same routine, which fetches all threads and sets the signal actions, sends

a signal to all threads. This approach makes possible to read the stack traces of all threads. In order to perform this task periodically, the agent starts a new thread which invokes the explained function and sleeps for a certain time period repeatedly. This approach works only with a constraint. The signal action can only be performed, when the state of the thread is safe. For instance if a method is currently performed in a specific thread, the signal action has to wait until the end of the method invocation, before it gets pushed on the call stack. As a result the agent would always miss the last invocation of the current stack trace. As illustrated in Figure 5.4, the signal action would wait until the end of method B. The light gray block visualizes that the method is running. The signal action will be pushed only afterwards on the call stack and `Thread.callStackSymbols` will be called. At the end, the signal action will be popped from the call stack, the invocation of method A continues and the normal code flow will be resumed.

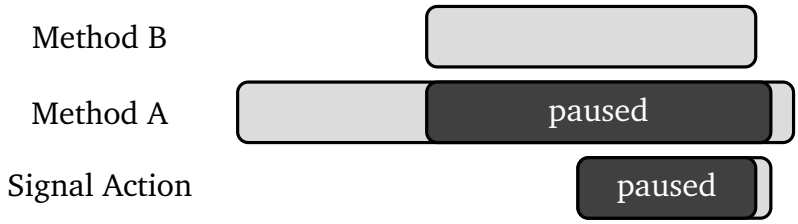


Figure 5.4: Invocation Schedule

The described issue leads to losing important information about the call stack and the agent is not able to reproduce the right invocation sequence. In the worst case a problem occurs on the latest invocation, which does not perform other nested calls.

An other option is to simulate the concept of call stack sampling by appending a call stack invocation on every written function at the start and at the end. If the time passed from one call stack invocation to the other one is less than the predefined time period, we ignore the last invocation, and wait for the next one. Fast methods as getter and setter will be ignored in this case, which reduces the overhead of the agent and resembles the real call stack sampling.

5.2.5 Automatic Source Code-Level Instrumentation of Invocations

The manual source code-level instrumentation strategies are producing high overhead in terms of integrating the agent in the project. The application developer needs to manually rewrite all starting and end points of all methods, in order to make them traceable by the agent. This reduces the code quality for reading and costs time. Since this concept is based on tracing functions, we could write a precompile process program.

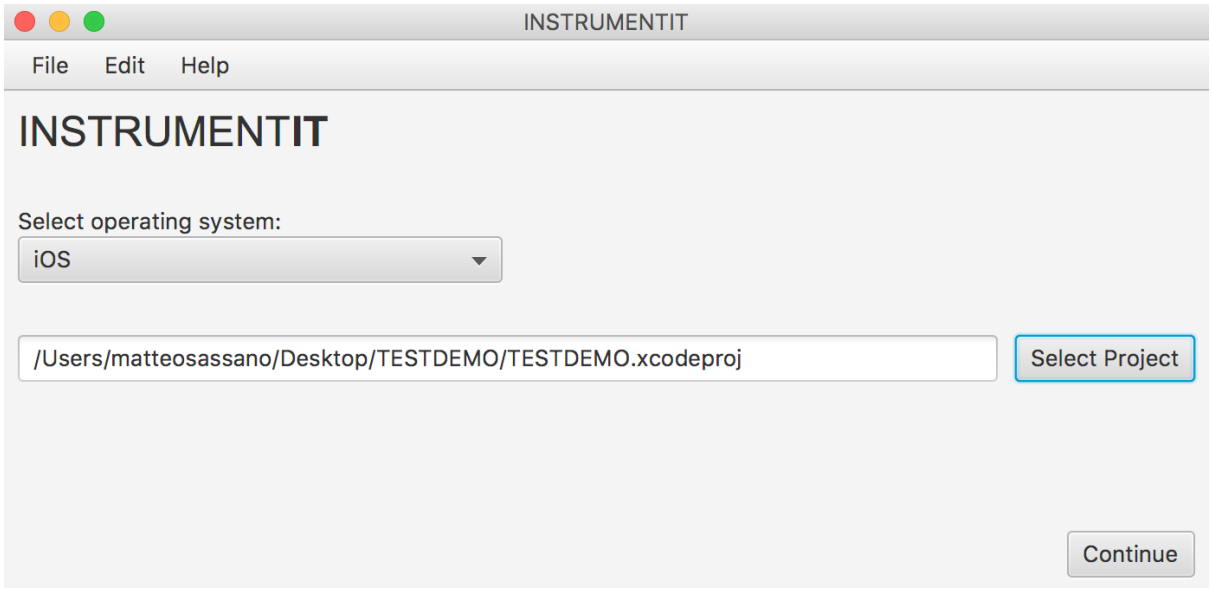


Figure 5.5: iOS project selection

For this strategy we implemented *instrumentit*. This process is built up of two parts. The first part performs a static analysis of the source code. It comprehends parsing the source code and the recognition of Objective-C and Swift specific keywords such as `func`, `class` and `static`. Afterwards by analyzing the parsed signatures, instrumentit is able to convert them into selectors needed for method swizzling. Optionally it stores the eventual instrumentation position for automated tracing. The second part comprehends the instrumentation code generation and placement. For each recognized and selected method to instrument, the process performs the swizzling of methods with help of a template. The template defines the source code grammar used for method swizzling. At the end the generated source code will be added in an appropriate place in the project to be compiled and executed later on.

In below there is a walk-through describing the usage of the instrumentit concept. As shown in Figure 5.5 the developer initially needs to select the iOS project. By clicking on the *Continue* button the instrumentation view, illustrated in Figure 5.6, appears. At this point the developer has the possibility to decide, whether he wants to trace all methods or only specific ones by selecting them on the user interface. In addition to that, he is able to decide the instrumentation strategy for each selected method.

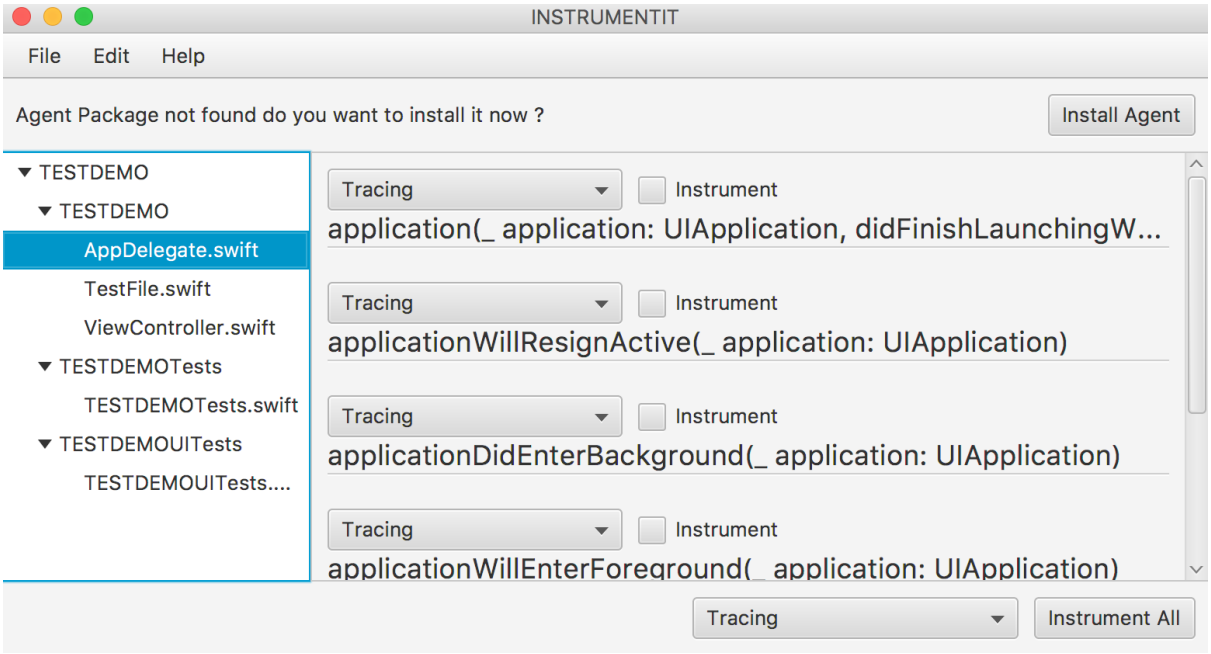


Figure 5.6: Prototype of the Agent Invocations Writer Program

After the method selection, the instrumentation application will modify the iOS source code automatically, in order to make the selected methods traceable.

5.3 Data Management Strategies

This section concerns about management approaches performable by mobile agents. It includes the various data model blueprints, *Span Instance* and *Span Message*, the data holding and controlling structures, dictionary, stacks and tree organization, and presents the correlation possibilities between measurement probes. In order to understand which functionalities the data management phase has to implement, the data management pipeline is provided in below.

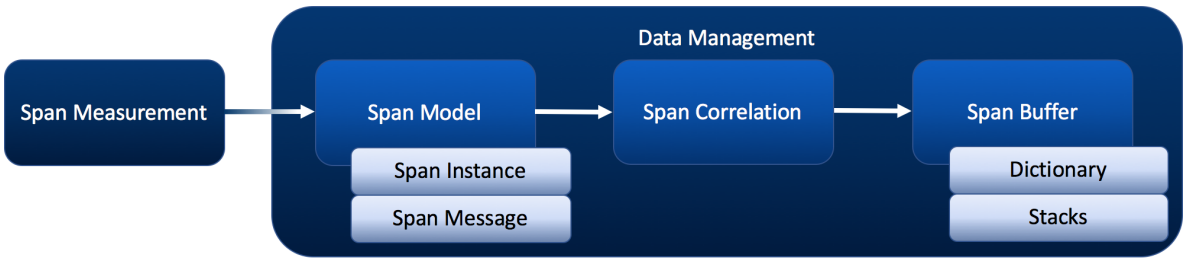


Figure 5.7: Data Management Pipeline

As illustrated in Figure 5.7, after collecting measurement probes, the agent creates an object related to the source code measurement, provides correlation settings between related spans and buffers the instance for further processing. Invocation instances as explained in Section 5.3.1 or log strings as demonstrated in Section 5.3.2 may be used as information blueprints. The emerging problem of correlating trace data is explained in Section 5.3.3. Opentracing provides a solution, which is explained in Section 5.3.4, for correlating measured spans. The following sections, Section 5.3.5 and Section 5.3.6, describe how to integrate the opentracing solution to the presented data models. In the last section we will present data organization strategies needed to buffer the measured trace data. Apart from buffering data, the organization strategies have also to be able to speed up the correlation between spans. This section additionally demonstrates and explains the expected running time of each methodology.

5.3.1 Model: Invocation Instance

As mentioned in Section 5.2 apart from executed source-code blocks we need to measure the execution duration and the starting time. The first possibility for the information model is to create an invocation object holding the mentioned attributes. For storing the execution start time one can use an unsigned 64-Bit UNIX timestamp, describing the time from the 1. January 1970 to now in milliseconds. The execution duration is the difference of the ending timestamp and the stored start timestamp in milliseconds. This attribute can be stored in an unsigned int. An unsigned 32-Bit int can store values from zero to $2^{32} - 1$. This would mean that we assume the maximum duration not to be more than $\frac{(2^{32}-1)ms}{1000}s = 4,294,967.295s = 71,582.78825min$. A unsigned 16-Bit short may not suffice the hold the duration. The maximum duration before a roll-back would be $\frac{(2^{16}-1)ms}{1000}s = 65.536s = 1.09226\overline{6}min$. An other argument for using a 32-Bit integer instead of an 16-Bit is the stack spacing. The size of one word of 32-Bit machines is 4 bytes. If the compiler pushes a 16-Bit variable on the stack, which is a normal occurrence when a object is invoked, it reserves 32 bits instead of 16. The reason for

that is to maintain the stack aligned. On 64-Bit machines the compiler performs an 8 byte alignment, due to the fact that words are 64 bit long. Therefore we allowed to use an unsigned 32-Bit integer on a 32-Bit architecture and an unsigned 64-Bit long on a 64-Bit architecture with the same memory usage. The primitive types `NSUInteger` [NSUI17] for objective-C and `UInt` [UIN17] for Swift are able to distinguish whether the compiled code is running on a 32 or 64-Bit machine. For the first option `NSUInteger` or `UInt` is an unsigned 32-Bit integer otherwise 64-Bit. The name of the execution is stored as a String. The `Invocation` class holds the string reference, which is the size of a word of the device architecture. The invocation properties may be set with the `startInvocation` method a the beginning and `closeInvocation` method at the end. To summarize the mentioned facts from above, Figure 5.8 illustrates the invocation class in a UML diagram with Swift types. The defined minimalistic construct allocates 40 bytes on 64-Bit architecture in total. Eight bytes for the `name` property, eight for the `startTime` attribute, eight for the `duration` variable and 16 bytes for the object meta-data. On 32-Bit machines the allocated space for an invocation object amounts to 28 bytes in total. Four bytes for the `name` property, eight for the `startTime` attribute, four for the `duration` variable and 12 bytes for the object meta-data. Additionally we have to add the string size which the `name` pointer references to in order know the exact amount of memory allocated by an invocation object.

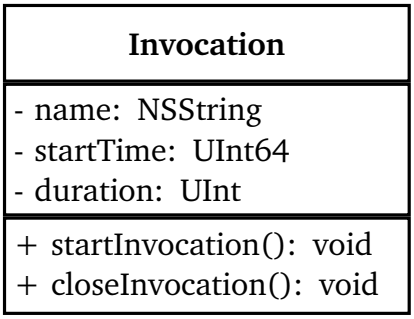


Figure 5.8: Invocation Class

5.3.2 Model: Log

Instead of having a real invocation model the mobile agent could log occurred events. In this case, creating objects for each invoked procedure is not a necessary task. Alternatively, the agent could use strings as event protocols which are buffering the invocation name, the execution start time and the duration of each invocation. A log message needs a certain pattern to distinguish the various invocation attributes from each other. Therefore, we need to introduce a rule for positioning and separating the different

attributes as presented in Figure 5.9. We define the *space* symbol as the separator. The positioning order of the attributes is: invocation name, starting time and end time.

```
name␣startTime␣duration
```

Figure 5.9: Log Message Preset

The ranking order of the mentioned attributes was established es defined to incrementally append new information. The first information the mobile agent might receive is the performing function name and the execution start time. At the end of the method execution the mobile agent calculates the duration and appends the result at the end of the log message.

The memory usage of this construct will amount up to at least 146 bytes on 64-Bit machines. Two bytes are required for the separators, 128 bytes to binary represent the execution starting time and the execution duration and 16 bytes (8 bytes for 32-Bit) are required to hold the string address. We could optimize the log message in terms of memory usage by converting the numeric attributes in hexadecimals. This convenience would reduce the amount of memory to at least 50 bytes for 64-Bit architectures and to 34 bytes for 32-Bit systems.

5.3.3 Correlation of Locally Instrumented Invocations

The sections before discussed how to collect information about invoked methods and how to model them. At this point the mobile agent would collect measurements separately from each other without establishing relations between each other. As mentioned in Section 2.1 an execution trace is a monitored sequence of method invocations. A method may be called within an other method. Considering the ignorance of setting relations between instrumented invocations may lead to false information description. The agent could have the possibility to recognize the execution trace based on the collected start time and duration of each method. The agent could recognize the caller of other methods as shown in Algorithm 5.1 if the start time of the callee is more than the start time but less than the start time added with the duration of the potential caller. One contra argument for this approach is the bad operational time. The algorithm with quadratic operational time $\mathcal{O}(n^2)$ has to be repeated after every closed invocation. An other problem is that the described approach will even fail due to an other lack of awareness of the agent: parallel execution-traces. On multi-threaded systems there might still exist more than one running execution trace. The iOS architecture supports multi-threads [ATH14], therefore it is obvious to find an other solution. Opentracing [OPT17] defines

Algorithmus 5.1 Basic correlation

```
procedure CORRELATEINVOCATIONS
    for all p ∈ I_closed do                    // I_closed is the set of closed invocations
        for all c ∈ I_closed do
            if p ≠ c then
                if p.startTime < c.startTime then
                    if c.startTime < p.startTime + p.duration then
                        p.PARENTOF(c)
                    end if
                end if
            end if
        end for
    end for
end procedure
```

a structural solution for the mentioned problems. The following section will describe the Opentracing model construct and explain its benefits.

5.3.4 Opentracing

For bypassing the problems mentioned above, Opentracing [OPT17] defined a model for helping establishing relations between invocation measurements. Execution-traces are defined by their so called spans. One can imagine a span as a source code-based measurement, independent if it only a method measurement or a measurement expanding more than one invocation. A trace is than defined as a tree of spans. In Figure 5.10 one can see a sample trace with span nodes. In this case *Span A* is the caller of *Span B*, *Span C* and *Span D*. The three callee might have other child spans. In our case, *Span B* calls *Span E*. The edges of the tree are representing the relation between the spans. If a span above is related to a span below, the upper one is the parent span of the lower one. For providing this functionality, Opentracing expanded the span model. In addition to an operation name, the start time and the end time, their model holds a map, named span tags, for storing custom information, a so called SpanContext and references over the SpanContext. The SpanContext holds a unique id of the respective span and a trace id. The trace id correlates an amount of spans that were executed in the same execution-trace. The other reference property is the ability to set a span as a parent of another one. The benefit of this model structure is the ability to set the relations at the span creation. Dependent on the organization of the spans, the agent could look up whether there is a running trace and a parent span and set those properties when instantiating a new span. By performing the correlation in this way, the agent

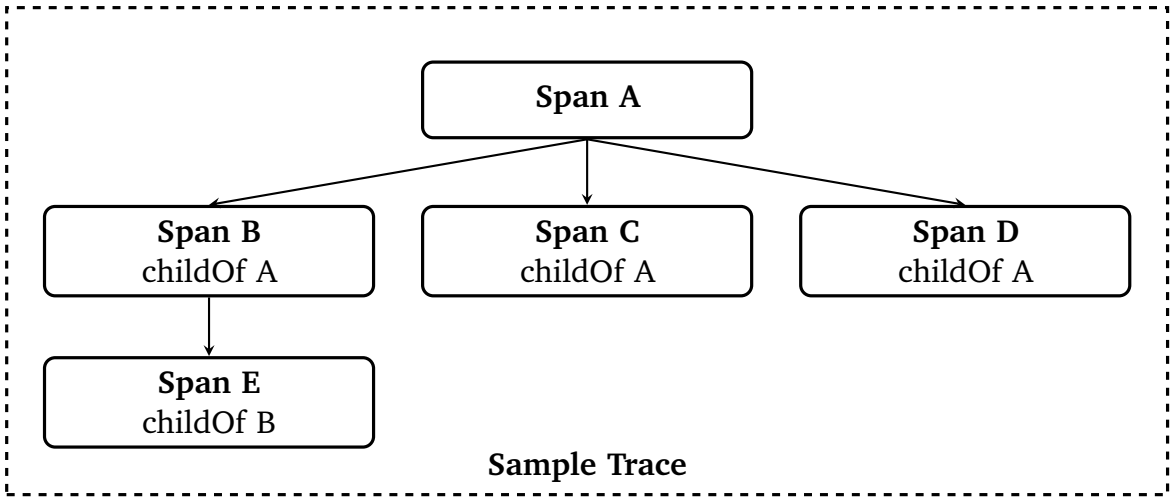


Figure 5.10: Span Tree representing a Sample Trace

avoids to repeatedly run the poorly performant Algorithm 5.1. The performance of this correlation strategy is highly dependent on the organization structure of the collected spans. The following sections [REEF] will describe the adaption of the Opentracing structure in our models defined in Section 5.3.1 and in Section 5.3.2. The sections [REEEF] will additionally define organization and management strategies and discuss the performance of the correlations.

5.3.5 Model: Invocation Instance with Opentracing Adaption

The model defined in Section 5.3.1 need to be extended to fulfill the requirements defined for the Opentracing construct. The invocation class has to include properties such as `id`, `parentId` and `traceId`. These three attributes are resembling the SpanContext defined in the Opentracing format, used for establishing relations between invocations or spans. As the id of each span has to be unique we suggest to use an 64-Bit unsigned numeric long for the mentioned id properties. The invocation class can optionally hold a map holding the span tags for storing additional information regarding to the measured span. In terms of memory usage, the amount of allocated memory for an invocation object grows to 72 bytes on a 64-Bit architecture and 56 bytes on a 32-Bit architecture. The three ids are allocating three times eight bytes. The span tags map is stored by reference, therefore it consumes eight bytes on a 64-Bit machine and four bytes on a 32-Bit. Figure 5.11 presents the new invocation class which is adapting the functionalities of the Opentracing format.

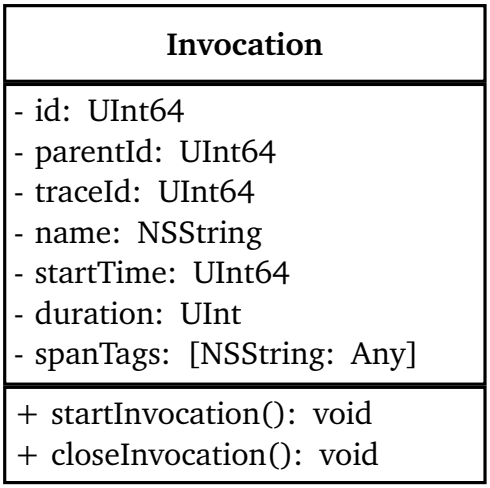


Figure 5.11: Invocation Class adapting Opentracing

5.3.6 Model: Log with Opentracing Adaption

If one decides to utilize log messages as span model, it is also necessary to include Opentracing specific properties. The log message can be extended with four more values: the three ids as hexadecimals and the span tag reference address as hexadecimal. Figure 5.12 illustrates the structure of the new log message.

```
name␣startTime␣id␣parentId␣traceId␣spantag-address␣duration
```

Figure 5.12: Log Message Preset with Opentracing Adaption

On 64-Bit architectures, this constructs' memory usage will amount up to 118 bytes (94 bytes on 32-Bit). The log message need four more separators (four bytes), includes three 64-Bit hexadecimal values for the ids (three times 16 bytes) and stores the span tag address as an hexadecimal value (16 bytes on 64-Bit architectures, eight bytes on 32-Bit). The memory size of the string is equivalent to the number of characters in the string. Searching the required property would be performable in expected linear time $\mathcal{O}(n)$, by reason of scanning each property position. Since the length of the log message is static, we can optimize the estimated lookup for each attribute. The lookup consists of searching the right string position and scanning the right amount of bytes. The agent could start scanning a certain amount of characters of the log message at a predefined index offset. For instance scanning the invocation id would require to start scanning 16 bytes from index 34. The described lookup methodology is only performable if the attribute indices are stored at some point of the source code.

5.3.7 Management: Span Organization and Span Correlation

Spans are normally instantiated when the instrumentation start point is invoked. Those spans need to be buffered locally in order to add and calculate the missing attributes of each invocation. The instrumentation source code point at the start creates the invocation object, generates a unique id, declares the name and measures the start execution time of the invocation. The end point measures the end time and calculates the duration of the invoked method. The setting of the attributes `parentId` and `traceId` is not automatically performable without a well defined span organization. Assuming the opposite case and the agent detected four spans as illustrated in Figure 5.13 and just stored them in a stack, the invocation sequence is not trivial. On one hand the stack could resemble the execution sequence, as shown on the right hand-side of Figure 5.14, on the other hand certain spans may be invoked in an other thread than other ones. For instance the left hand-side of Figure 5.14 shows that *Span A* was invoked on the main thread and invoked *Span C* and *Span C* invoked *Span D* and *Span B* was called in an other thread. Considering the cases, presented on Figure 5.14, the agent is not able to automatically set the parent span and the trace id.

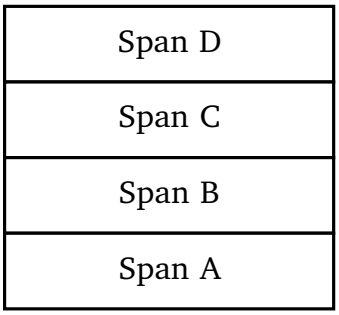


Figure 5.13: Span Stack without Organization

The only opportunity for setting the proper ids is to manually inject them. The agent could easily be extended to perform this task. The problem emerging using this kind of correlation strategy is the increase of work overhead. Manually setting the relation properties is requiring a huge understanding of the written source code. An other problem is the delivery of the span id in cases of nested calls. The span id is generated at the start of each method invocation. If the invoked method calls an other method the span id of the caller somehow has to be passed to the callee method. One possibility is to rewrite the method signatures to provide other arguments such as the `parentId` and the `traceId`. This approach will increase the workload even more considering that the developer would be obligated to edit every method signature he wants to instrument. The solution for avoiding the increase of workload is to utilize a proper organization structure. The other benefit emerging using a defined organization is the automated correlation

between spans. The next subsections present possible organization strategies usable for mobile agents. Each approach is described and analyzed in terms of functionality and performance.

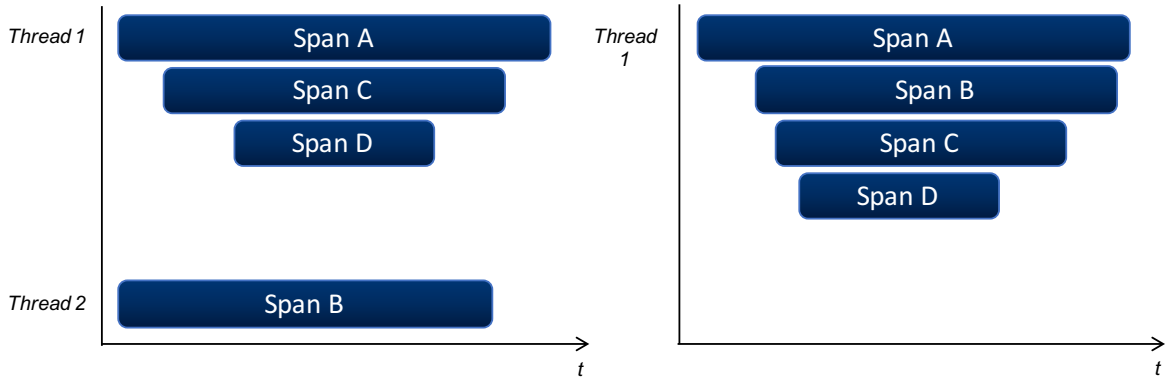


Figure 5.14: Non-Deterministic Trace Options

Organization in Stacks

As described in Section 5.3.4, an organization with a single stack would not fulfill the functional requirement FSR2.1 defined in Section 4.1.1. A possible option to solve the mentioned problems is the organization of the spans in multiple stack. Invocations measured in different threads are buffered in different stacks.

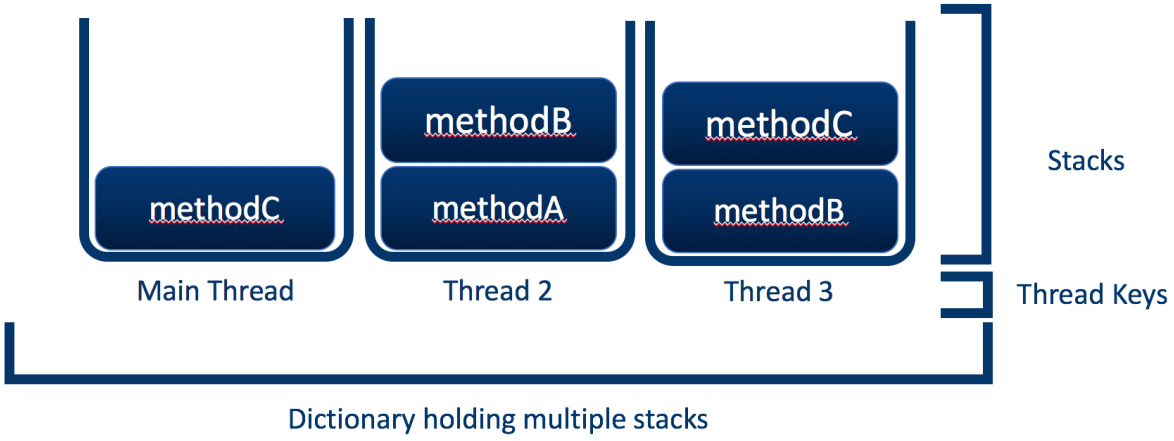


Figure 5.15: Multiple Stacks Organization

By using the organization structure as illustrated in Figure 5.15, the mobile agent has the possibility to check and retrieve the thread id or name and afterwards to select

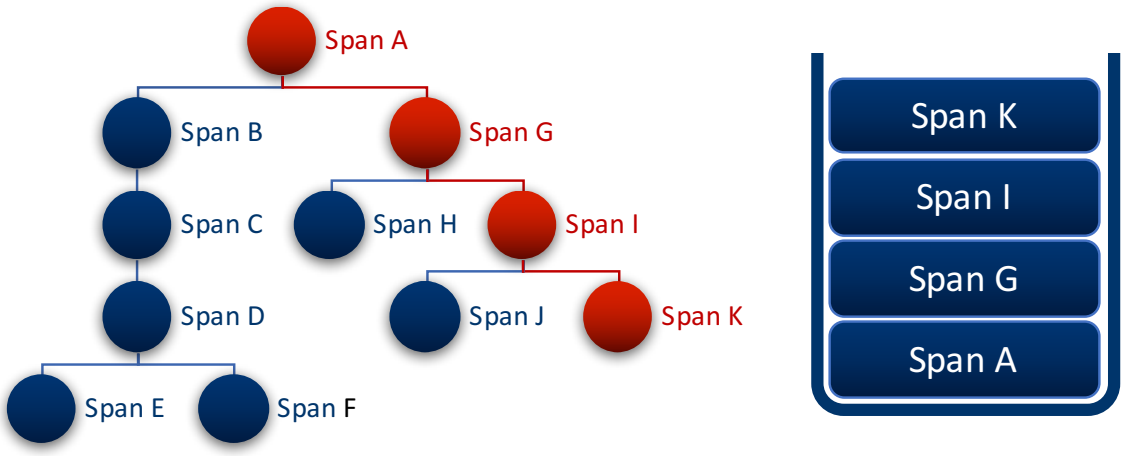


Figure 5.16: Monitored Span Tree and currently running Span Stack

the proper stack to push invocation object into. The moment snapshot of the different stacks resemble the various existent execution-traces with the running invocations. Pushing a span object in a stack means that a certain method was invoked. Popping an invocation element from the stack means that the invocation is closed. The stacks are not holding the complete execution tree, but only the currently running execution path. In Figure 5.16 one can see a monitored sample execution-trace as a tree with the respective trace stack on the right hand-side. The currently running execution path is marked red. As one can see, the last element of the stack is equivalent to the deepest currently running invocation of the execution trace, not trivially the deepest from all. This important fact can be used to set the correlations between spans. Assuming the methods identifier correspond to the span names and that invocation K does not perform any nested calls while its life span, ending the invocation would imply popping *Span K* from the stack. As result the parent *Span I* will become the last element of the stack. If the invocation I calls an other method after method K, for instance method L as shown in Listing 5.1, the stack organization would automatically correctly assume that *Span I* is the caller or the parent of *Span L*.

Listing 5.1 Example of Invocation I

```
func I() {
    // Span I is parent (caller) of J, K and L
    J()
    K()
    L()
}
```

The agent could therefore take the advantage of the multiple stack organization to set the correlations of the spans at the creation time. Before pushing the span into the stack, the agent retrieves the id and the trace id of the last element of the stack and copies the retrieved id as the parent id of the new span and sets the trace id identically to the trace id of the last span. An execution trace is complete, if all elements have been popped from the stack. At that point, a new trace id will be generated as well as each time the stack becomes empty. Since we do not know which and how many threads the application would use at runtime, the agent is not able to recognize how many statically defined stacks are required. The solution for this problem, is to use a dictionary of stacks. The dictionary key is defined as the thread id, which can be retrieved calling the static attribute `Thread.current.description`. The current description returns the name and the number (id) of the current thread. In this case, the agent uses the thread number as the dictionary key. Using the defined dictionary will allow the agent to dynamically allocate multiple stacks for each currently running thread. The agent checks, whether the key exists. If it is the case, the agent accesses the corresponding stack, if not a new key would be added and the corresponding stack would be instantiated. In terms of performance the stack accesses are performable in expected $\mathcal{O}(1)$ [CFD98]. The stack provides $\mathcal{O}(1)$ for accessing, pushing and popping the last element. This increases the overall performance of the agent, considering that each open span can be organized and correlated to its parent span in $\mathcal{O}(1)$. Dependent on the data dispatch strategy, there are cases where the closed spans need to be buffered locally. For the local buffer the agent could use an other stack, since it is irrelevant which span is serialized and sent first. In this case, popping an invocation from the running stack would mean to push the closed invocation into the other stack holding the closed spans. The overall memory usage of this approach could maximal amount to the count of all spans, therefore $\mathcal{O}(n)$.

Organization in a Dictionary

Rather than using multiple stacks this approach uses two dictionaries to organize measured spans. The first dictionary is to hold all spans. The key is the span id and the value returns the span object containing the span id. The second dictionary (correlation dictionary) is used to establish the correlations between spans. The basic task of this dictionary is to hold the deepest currently running span (running child) of each currently running execution-trace. Therefore the keys of the second dictionary are the thread ids and returns as value a span object.

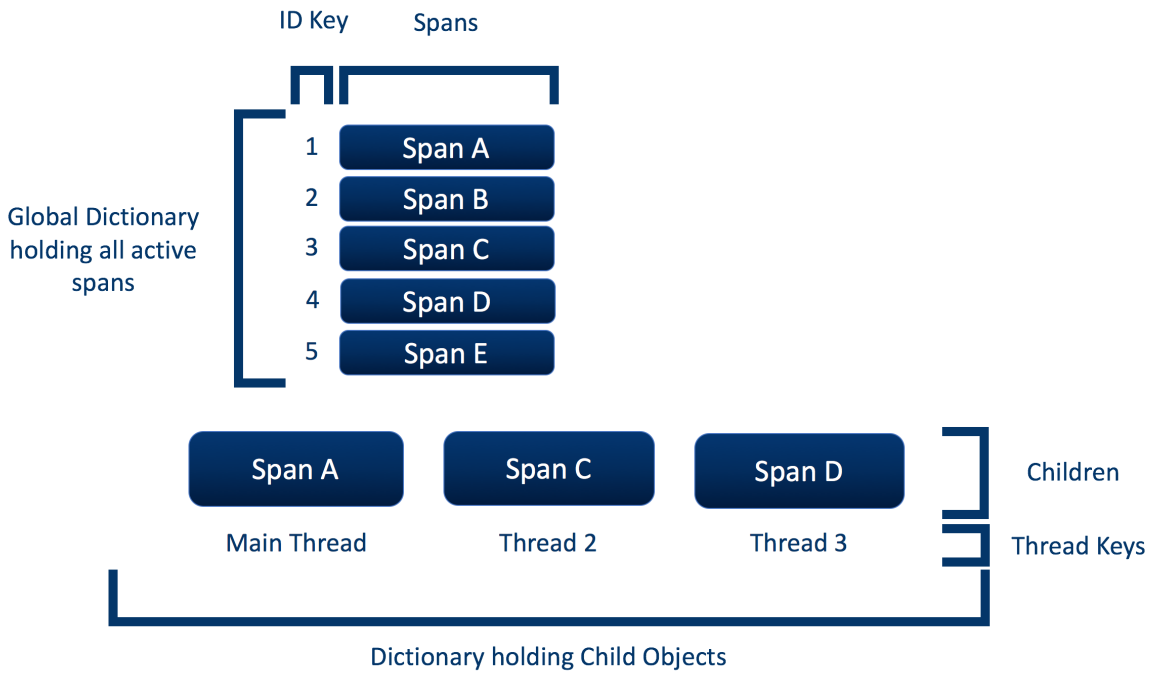


Figure 5.17: Organization with Dictionaries

Figure 5.17 illustrates the mentioned dictionaries with, containing sample objects. Monitoring an invocation with this organization would mean creating a span object, retrieving the current thread id and looking up whether the thread id is existent as a key in the correlation dictionary. If the key is existent, the agent will access the span object and copy the id as the parent id of the new span. Additionally the old span will be replaced by the new one, considering that the new one became the currently running most deepest child of the trace. If the key is not existent, the new span will be marked as a root span, so the trace id, the parent id and the span id have the same value, and mapped with the thread id key on the correlation map.

In both cases the new span will be mapped on the first dictionary, which holds all spans. This is important considering that method invocations could end without performing other nested calls. The consequence is that the correlation dictionary has to work also in the back direction. Closing a span which is hold by the correlation dictionary, would mean that the currently running deepest child is changed. At that moment the caller, which is the parent span, becomes the deepest running child. To perform this task, the agent can retrieve the parent span by getting the parent id of the old running child and collect the object stored with the copied parent id as key. As the agent uses a dictionary as the data structure for holding all spans this task is performable in expected $\mathcal{O}(1)$. The memory consumption of this organization strategy is dependent from the amount of

measured spans. The size of the second dictionary is also dependent of the count of the running thread multiplied by one span object. Assuming that the running threads are constant the overall memory consumption would be $\mathcal{O}(n)$

Organization in a Tree

Since an execution-trace is composed based on a tree structure, as defined in Section 2.1, the mobile agent has the possibility to organize the trace as it is in a tree. The root node of the tree would resemble the root span of the trace and children of a specific node are the callee of the node. For making the traversal of the tree possible, each node object has to hold the corresponding span object, a stack holding its children and its parent node. To support multi-threading, the agent has to adapt a dictionary which maps each thread to its execution tree. Since we store the hole tree for each thread, the agent is only able to trivially recognize the currently running invocation sequence. For this problem we have to find a correlation between the tree and the invocation sequence currently running. We could add the convenience that child elements are sorted by its execution time. In that case only spans located on the most right hand side of the tree are possible spans contained by the currently running invocation sequence, but not consistently all. The agent has to check at each level, whether the child node is closed or not. The other remaining nodes are all closed. Since we need the currently running invocation sequence or the last element of the sequence, the agent has to propagate through the tree to set the parent properties of the new measured span. Considering that the tree is not balanced, the worst case of finding the running child is $\mathcal{O}(n)$. The absolute amount of comparisons are equal than the amount of elements in the invocation sequence which makes this organization strategy less performant than the other ones in terms of execution times.

5.4 Data Dispatch Strategies

This section presents and discusses the various dispatch strategies for mobile agents. Since this work should evaluate monitoring strategies for iOS mobile devices, we have to take certain constraints in consideration, which normally can be ignored when programming an other agent monitoring enterprise application running on non mobile devices. The mobile agent has to consider the network connectivity of the mobile device and the network usage overhead has to be as minimal as possible. The reason for that is the possible slow bandwidth connection and the mobile data volume is usually limited. This section is separated in dispatch time strategies, dispatch circumstances and failure handling. Further more, serializing the buffered spans and hardware information is also

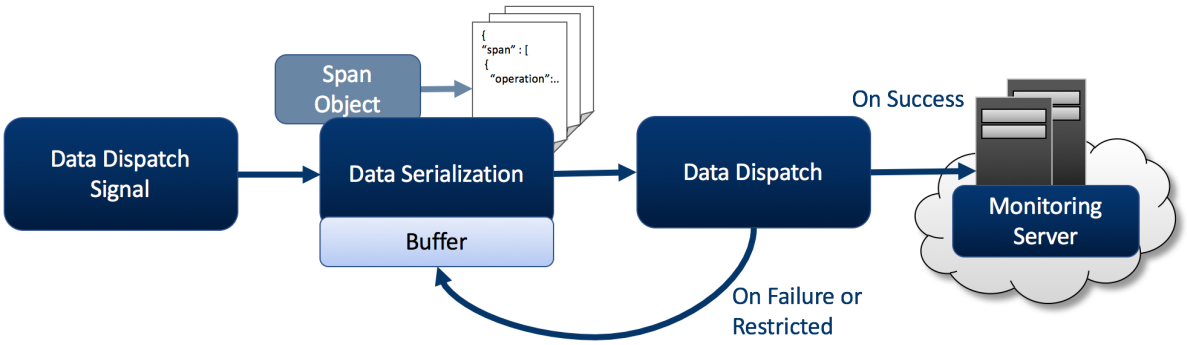


Figure 5.18: Dispatch Pipeline

part of the data dispatch phase. As shown in Figure 5.18, the mobile agent receives a certain dispatch signal, which is related to the dispatch options, collects the buffered spans, serializes them and sends them to the monitoring back end. In cases a failure occurs, when sending the data, the serialized spans will be re-buffered in their serialized state. For dispatching measured data to an appropriate back end, a communication interface has to be established. This includes planning and defining the data model and the communication interface composition itself. The used data model is specified in Section 5.4.3.

5.4.1 Dispatch Options

This section regards to approaches for sending monitoring data to an APM back end. The focus of this section lays on the different possible events which are conducting to dispatch the data. The dispatch options, the following subsections will analyze, are *Dispatching each Single Span*, *Dispatching complete Execution Traces* and *Periodical Dispatching*.

Single Span Dispatching

When utilizing this span dispatch strategy, the agent waits until it finished monitoring the method. Afterwards the agent gets the signal to serialize the span object and to send it to the back end right away.

Complete Trace Dispatching

The event for this dispatch strategy is fired in the case the agent closes a complete execution trace. Closing a complete trace means that the agent reached the root span when closing a monitored invocation. Each time the agent closes a span, it checks whether the id of the closing span is identical to its parent id and trace id. If this is the case, the agent recognizes that it is closing a root span. The consequence is that the agent collects the buffered elements, serializes them and sends them to the monitoring back end.

Periodical Dispatching

If the agent implements the periodical dispatching, the agent checks, each time it takes measurement probes of hardware resource workloads, if the span buffer contains closed spans, the agent collects them, serializes them and sends the serialized spans to the monitoring back end.

5.4.2 Dispatch Constraints

Considering that the various events of dispatching monitoring data are not influencing the total amount of sent bytes to the back end, it is more important to focus on the dispatch constraints. The dispatch options are only ruling the distribution of sending the data. For instance, if we compare the first option, *Single Span Dispatching* and the second one, *Complete Trace Dispatching*, in total the agent will send nearly the same amount of data but in different iterations. Therefore it is more important to focus on dispatch constraints. Dispatch constraints are defining whether the agent is allowed to send the serialized data, independently on which dispatch option it uses. The defined dispatch constraints are *Send Always* and *Send only via Wifi*. Rather than a constraint, *Send Always* is a flag that allows the agent to send monitoring data to the back end, independently on the connectivity of the mobile device. The constraint, *Send only via Wifi*, forces the mobile agent holding the serialized data and not sending them to a monitoring server, in the case the mobile device is not connected to the network via Wifi. This constraint helps reducing the overhead of the network usage performed over the mobile bandwidth. In addition to that, in this case the agent would not affect the provided data volume of the end-user, which is often limited. An other benefit of this constraint is, that usually slow bandwidth connections, due to problem locations or limited data volume of the end-user, are not utilized by the mobile agent. Counter arguments for activating this constraint is the deferred dispatch of monitoring data, the

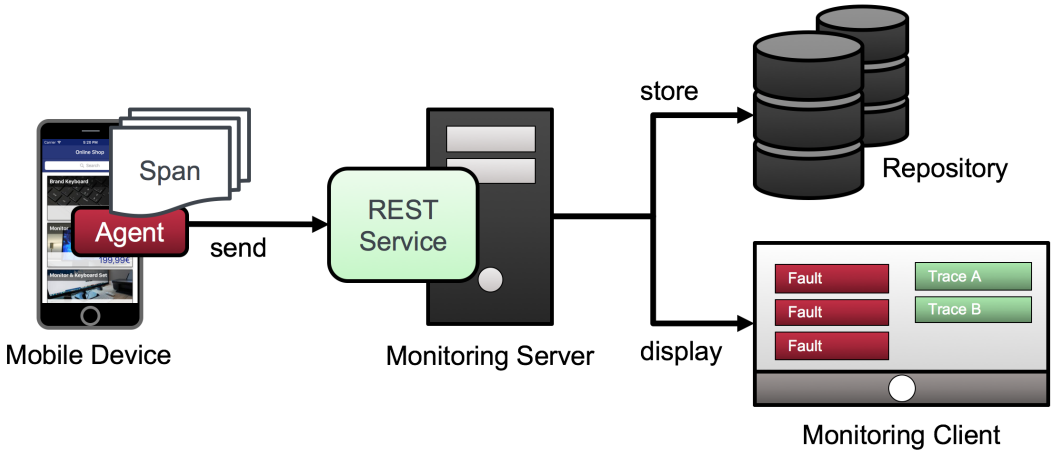


Figure 5.19: Dispatch Pipeline

increased memory usage of the local span buffer and lose of information in case memory allocated for buffer has to be freed, due to memory resource shortage.

5.4.3 Back end Communication

If the back end integrates a REST API, the agent would be able to submit the monitoring data with HTTP post requests. Therefore, the agent and the back end need to establish a proper model to exchange the data. Figure 5.19 shows the traditional pipeline for dispatching collected data. The agent on the mobile device collects and serializes code and device-based data and sends them through a REST interface to the monitoring server. Afterwards, the monitoring server stores the data and provides the monitoring client with data. For the purpose of this thesis we will use the data export model defined in the development project *Mobile-aware Diagnosis of Performance Problems in Enterprise Applications* [DEVP17]. Listing 5.2 displays the equivalent JSON object, which is sent, when the agent has traced one invocation. The JSON objects contains three main parts. The first part is a list of spans with the identifier `spans`. Spans are containing their identification ids, the execution time of an invocation, the method name and other span based properties. In this example, the method that was invoked was `viewDidLoad()`. The key `operationName` holds the name of the invoked method. The execution time of the method was 200 micro seconds. This value is hold by the key `duration`. The inner object named `tags` holds span-based properties. For instance if a method execution was traced from a mobile device, the span kind is from a client. In addition to that, the key `ext.propagation.type` stores, which operating system runs on the caller device. The span context object holds the id, parent id and the trace id. The

key `startTimeMicros` holds a UNIX timestamp in micro seconds. The second part of the export object contains the device id. The device id is important to distinguish different mobile devices from each other. The last section of the JSON object is the mobile device resource measurement list. Each object of this list holds a UNIX timestamp in micro seconds. The key `cpuUsage` holds the percentage usage of the CPU at timestamp time. In below the RAM memory usage and the persistent memory usage are also stored in percentage. The last resource metric is the battery power. The value of `batteryPower` is also stored in percentage.

Listing 5.2 Example of a JSON object with one Invocation

```
{
  "spans" : [
    {
      "operationName" : "viewDidLoad()",
      "duration" : 200,
      "tags" : {
        "ext.propagation.type" : "IOS",
        "span.kind" : "client"
      },
      "spanContext" : {
        "id" : 10965962947820502759,
        "parentId" : 10965962947820502759,
        "traceId" : 10965962947820502759
      },
      "startTimeMicros" : 1506257375819915
    }
  ],
  "deviceID" : 12167283130449148877,
  "measurements" : [
    {
      "cpuUsage" : 0.001375,
      "memoryUsage" : 0.9234822,
      "storageUsage" : 0.4457262502440192,
      "timestamp" : 1506257376826065,
      "batteryPower" : 0.90,
      "type" : "MobilePeriodicMeasurement"
    }
  ]
}
```

The second example shows the object difference, when the notices nested calls. In this case, we consider that `viewDidLoad()` calls `loadContent()` as shown in Listing 5.3. As a result the agent should notice two functions and create two spans. Since `viewDidLoad()` is the caller of `loadContent()`, the viewDidLoad span has to be the parent span of the loadContent span.

Listing 5.3 Example of a nested Invocation

```
override func viewDidLoad() {
   let id = IITMAgent.getInstance().trackInvocation()
   self.loadContent()
   IITMAgent.getInstance().closeInvocation(id: id)
}

func loadContent() {
   let id = IITMAgent.getInstance().trackInvocation()
   // load some content
   IITMAgent.getInstance().closeInvocation(id: id)
}
```

The difference of the exporting JSON object is shown in Listing 5.4. The spans list holds an other span object. As one would assume, the id of both spans is different. The parent relation is set trough the span context. As one can see, the parent id of the loadContent span is the id of the viewDidLoad span. As shown in the second example, a nested span is also allowed to be a remote call. A remote call can be distinguished from a method invocation through `ext.propagation.type` value. The value for remote call at that point is HTTP. As one may also notice, a remote call has more tags attributes. It holds request and response properties of the actual network connectivity, the geolocation, the router SSID and the mobile carrier name. In addition to the mentioned properties a remote call span tag also holds the URL which the mobile device requests data from.

Listing 5.4 Example of a JSON object with nested Invocations

```
{ "spans" : [
    { "operationName" : "viewDidLoad()",
      "duration" : 255,
      "tags" : {
        "ext.propagation.type" : "IOS",
        "span.kind" : "client" },
      "spanContext" : {
        "id" : 13174449888346817415,
        "parentId" : 13174449888346817415,
        "traceId" : 13174449888346817415 },
      "startTimeMicros" : 1506260131052448 },
    { "operationName":"loadContent",
      "duration":12000,
      "tags":{
         "http.request.networkConnection":"4G",
         "http.response.longitude":"48.321",
         "http.request.latitude":"13.12345",
         "http.response.networkConnection":"4G",
         "http.response.timeout":"false",
         "span.kind":"client",
         "http.request.ssid":"1234-5678",
         "http.request.networkProvider":"MyProvider",
         "http.response.latitude":"13.52345",
         "http.request.responseCode":"200",
         "http.url":"localhost:8080/callRest",
         "http.response.networkProvider":"MyProvider",
         "http.request.timeout":"false",
         "http.request.longitude":"48.421",
         "http.response.ssid":"1234-5678",
         "ext.propagation.type":"HTTP" },
      "spanContext" : {
        "id" : 13633138442564838767,
        "parentId" : 13174449888346817415,
        "traceId" : 13174449888346817415 },
      "startTimeMicros" : 1506260131052666 }
  ], ...
}
```

Chapter 6

Implementation

In this chapter we will introduce the implementations of the various strategies used for this thesis and explain the selected implementation decisions. The implementation of the agent concepts are mostly written in Swift. In beginning of this chapter we introduce the architecture of the mobile agent module. We will show and describe the relations between the agent classes abstractly. In addition to that, we will comment the used classes and explain their purpose in these projects. In the following subsections, we will walk through the defined agent strategies and explain, with more details, the respective implementations. First of all, we will demonstrate and analyze the written code especially for the tracing approach. Afterwards, we will describe the important implemented parts for method swizzling. Above that, we will expose the instrumentation presets used for instrumenting system library-based methods. The implementation for the use case mapping approach will be described next. In the end of this chapter we will present the implementations of the automated approaches call stack sampling and the automated code injector.

6.1 Agent Architecture

Implementation approaches used for Java enterprise applications can not fully used as a reference for implementing an iOS mobile agent. The operating system iOS does neither allow to manipulate and change an already compiled and built application nor to request or manipulate data from an other local installed application. In consequence, the agent implementations had to follow an other approach. The concept behind our iOS mobile agent implementations is, that the application developer imports the agent modules into the developing application that has to be monitored. By following this approach, we ensure that the agent functionalities are compiled at the same time, the core application

is compiled. Therefore the agent has no need to manipulate an already compiled application. Since the agent configuration has to be imported as a framework to the original iOS application project, each agent bundle files and classes are conventionally named with a prefix. This is useful to avoid intersections in the name-spaces of used classes. Our file and class name prefix is *IITM* which is an acronym for ***I**nspect**IT** **M**obile*. In general all agent bundles contain an agent class. The `IITMAgent` class is the core of every agent configuration. Each application, that has to be instrumented, contains one instance of `IITMAgent`. This means one agent instance per application and not per device, since the operating system iOS does not allow to retrieve external application-based information from an other application. `IITMAgent` provides methods for tracking invocations or use cases. For allowing a global access to the agent, the `IITMAgent` class provides a static shared instance of the agent as a singleton. The singleton is accessible by calling the class method `IITMAgent.getInstace()` or by accessing to the static variable itself `sharedAgent`. All agent configurations also provide an opt out function. The opt out function disables the agent from instrumenting the application source code when toggled. To support this functionality, `Agent` provides an attribute `optOut` as a boolean value. The default value of `optOut` is false. If `optOut` is set to true, the agent will stop measuring source code-based data and will also stop measuring hardware resource-based data. This can be achieved by accessing to `optOut` through the singleton, or by calling the class method `Agent.disable()`. Agent properties are stored in a dictionary named `agentProperties`. In the actual state of the implementations, `agentProperties` is only holding the UUID of the agent running on a specific device. The reason why we decided to choose a dictionary for the agent properties is to improve the maintainability of the agent source code. In the case we need to add new agent attributes or properties, we will be able to add with less work overhead by mapping them on the dictionary. In addition to the mentioned attributes, the `Agent` class relates to all other modules. As mentioned in Section 5.1, an agent configuration has also to manage the collected data. For this task the agent relates to a data organizer named `IITMDataOrganizer`. Dependent on the management strategy, `IITMDataOrganizer` has a dictionary, stack or a tree to buffer the collected spans at runtime. `IITMDataOrganizer` also provides methods for establishing correlations and dependencies between spans. If the developer choses to work with invocation instances, defined in Section 5.3.1, the agent bundle needs have a blueprint for invocation instances. In our implementations supporting this model, we named the span class `IITMInvocation` for tracing and `IITMUseCase` for model used for the use case mapping. Additionally, we implemented `IITMRemoteCall` for holding a span regurding to a remote call. Hardware resource metrics are collected and buffered from an other module mapped in a class named `IITMMetricsController`. `IITMMetricsController` references to class and file methods defined in `IITMNativeRessource`, `IITMBatteryLife` and `IITMDiskMetric`. The c file `IITMNativeRessource` provides global file methods for retrieving the current CPU and RAM workload. `IITMBatteryLife` implements class methods regarding to the actual battery status and `IITMDiskMetric` offers methods

in relation to the current persistent memory status of the iOS mobile device. The metrics controller contains an own ring buffer implementation to hold the measured metrics values. The ring buffer is technically a list where the list objects are linked to each other mono directional. Apart from the already mentioned metrics collectors, our implementations additionally contain `IITMNetworkReachability`, `IITMSSIDReader` and `IITMLocationHandler`. `IITMNetworkReachability` methods return information regarding to the mobile carrier, if existent, and regarding to the network connectivity to the web. The agents are distinguishing between a Wi-Fi connection and a mobile connection, which is specified in 2G, 3G and 4G. If the mobile device is connected to the web via Wi-Fi, `IITMSSIDReader` is able to read the router ID. This metric value is important for analyzing in cases of connectivity failures, depending on the WLAN router. The `IITMLocationHandler` reads the current geolocation of the mobile device user. The geolocation is fetched, when the user performs an action which is requesting data from a back and as consequence. The last part of the agent working pipeline is to send the gathered analytics data to a monitoring back end. As mentioned in Section 5.4.3, the agent bundle needs a REST interface to communicate with the REST interface of the monitoring back end. For this task, which is needed for all agent configurations we implemented `IITMRestManager`. The back end URL can be set by calling `Agent.setBackEndUrl(url: String)`. The developed rest manager is able to perform HTTP post requests in order to send monitoring data. The measured data has to be serialized in a JSON object with the structure defined in Listing 5.2 and Listing 5.4. The data serialization is done by `IITMInvocationSerializer`. For cases, the mobile device is not able to send the data to a monitoring back end, each agent configuration implements a local storage client for saving gathered data persistently. The persistent memory interface is controlled by `IITMDataStorage`. `IITMDataStorage` is able to store agent-based properties and to store measured instrumentation data.

In the following, in Figure 6.1, we demonstrate an abstract class diagram of the an overall agent implementation. The abstract class diagram illustrates all the relations between the agent classes and highlights optional parts of the implementations in dashed rectangles. As described in the section above, classes describing a span model are marked as optional, dependent on the span model strategy. In addition to that, `IITMProactive` is also an optional module. `IITMProactive` contains functionalities such as to inform the end user, if an error occurred or to re-manage collected data in cases the mobile device runs out of memory, RAM as well as persistent memory. In Figure 6.1, one is also able to see the one to one relations of `IITMAgent` with `IITMDataOrganizer`, `IITMMetricsController`, `IITMDataSerializer` and `IITMRestManager`. `IITMMetricsController` is related to `IITMNativeRessource`, `IITMBatteryLife` and `IITMDiskMetric`.

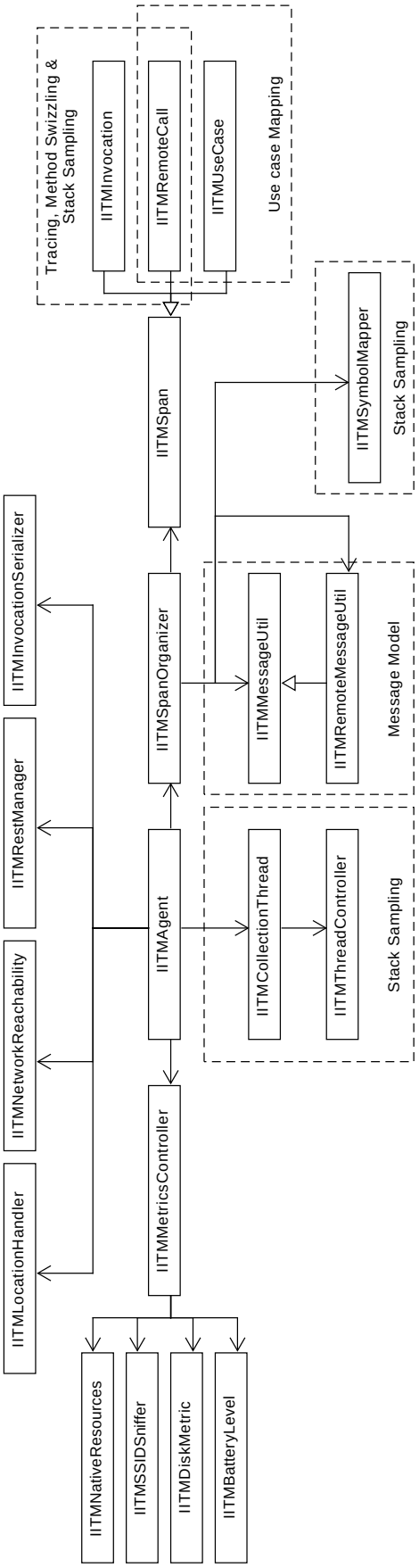


Figure 6.1: Abstract Agent Class Diagram

The following sections will describe the classes, implemented for the agent bundles, in detail. Their respective subsections, if available, will concentrate in the differences between the different approaches. For instance, if an agent class has to implement an other behavior, dependent on a different agent collection, management or dispatch strategy, the subsections will explain the differences. A practical example is the class `IITMAgent`, when using the invocation tracing approach and when using the use case mapping approach. As described above `IITMAgent` provides methods for tracking source code-based data. In both cases, invocation tracing and use case mapping, the tracking invocations has to behave in a different way from each other. In that case, the subsections will describe and explain the logical differences between the implementations.

6.1.1 IITMAgent

The `IITMAgent` is the core of our iOS agent configurations. The agent itself is implemented as a singleton and can be globally accessed by calling the static method `Agent.getInstance()` or by referring to the static attribute `sharedAgent`. Referring to `sharedAgent` is not advised, considering that instance is not checked if it is a nil pointer. Listing 6.1 shows the realization of the agent singleton. `IITMAgent` holds a static variable named `sharedAgent` of the type `IITMAgent`: the singleton. The method `getInstance()` checks whether `sharedAgent` pointer is nil or not. If `sharedAgent` is nil pointer, `getInstance()` returns a new shared instance of `IITMAgent`, otherwise it returns the current singleton. The agent singleton and the method to retrieve the actual instance of the agent has to be public. Otherwise, since the agent will be integrated as an addition framework project, the application developer will not be able to access to the agent methods. The default access modifier for Swift function is *internal*. The *internal* access modifier prohibits to use declared and implemented methods from the outside of the defining module.

Listing 6.1 IITMAgent Singleton

```
public static var sharedAgent: IITMAgent?

public static func getInstance() -> IITMAgent {
   if IITMAgent.sharedAgent == nil {
      return IITMAgent()
   } else {
      return IITMAgent.sharedAgent
   }
}
```

As mentioned above, if `sharedAgent` points to nil, a new agent object will be instantiated. In that case the constructor of `IITMAgent` will be invoked. Listing 6.2 presents all agent attributes and the agent constructor. The application developer is able to pass optionally a dictionary populated with agent setting values. In the case of Listing 6.1, in line five, we are not passing any arguments, which means that the `properties` argument is nil.

The first line of the constructor body initializes an empty dictionary. The key type is String and the values can be any type of objects. In line 15 until line 21, the agent attributes are initialized as default. If the optional argument does not reference to nil, the `agentProperties` dictionary will be populated in line 25 until line 31 of the shown source code. By calling `loadAgentId()`, the id gets assigned to the agent. In Listing 6.3, one is able to see, that the agent checks, whether an agent id is already stored. If it is the case, the agent id is set to the loaded value, otherwise the agent will generate a new id and store the calculated value persistently through the data storage.

Listing 6.3 IITMAgent Singleton

```
func loadAgentId(){
   if let agentid = dataStorage.loadAgentId() {
      self.agentProperties["id"] = agentid
   } else {
      let id = generateAgentId()
      self.agentProperties["id"] = id
      self.dataStorage.storeAgentId(id: id as! UInt64)
   }
}
```

The next instruction, `IITMAgent.sharedAgent = self`, sets the singleton object. Since the constructor should be called only once in the application lifetime, the singleton reference remains always the same. The following location handler instruction, `locationHandler?.requestLocationAuthorization()`, requests the authorization for fetching the geolocation of the device from the user. Otherwise, the agent is not allowed to retrieve the current location of the user, when required.

IITMAgent - Tracing & Method Swizzling

In the agent bundles, which are supporting tracing or method swizzling as a source code data collection strategy, the agent class implements methods for instrumenting the written source code. Since the functionality of both approaches is to intersect instrumentation calls in the beginning and the and of methods, for method swizzling in hooked methods, the agent class needs to provide two functions for instrumenting methods. In our

Listing 6.2 IITMAgent Constructor

```
var agentProperties: [String: Any]
static var sharedAgent: IITMAgent?
var invocationOrganizer: IITMInvocationOrganizer
var locationHandler: IITMLocationHandler?
var networkReachability: IITMNetworkReachability?
var dataStorage: IITMDataStorage
var invocationSerializer: IITMInvocationSerializer
var metricsConrtoller: IITMMetricsController
var restManager: IITMRestManager
var optOut: Bool = false

// IITMAgent Constructor
// Optional Argument: properties
init(properties: [(String, Any)]? = nil) {
   agentProperties = [String: Any]()
   invocationOrganizer = IITMInvocationOrganizer()
   dataStorage = IITMDataStorage()
   metricsConrtoller = IITMMetricsController()
   invocationSerializer = IITMInvocationSerializer(invocationMapper:
        invocationMapper, metricsConroller: metricsConrtoller)
   restManager = IITMRestManager()
   locationHandler = IITMLocationHandler()
   networkReachability = IITMNetworkReachability()

   super.init()

   if let properties = properties {
      for (property, value) in properties {
         if Agent.allowedProperty(property: property) {
            agentProperties[property] = value
         }
      }
   }

   loadAgentId()
   IITMAgent.sharedAgent = self
   locationHandler?.requestLocationAuthorization()
}
```

implementations we implemented `trackInvocation(function: String = #function, file: String = #file) -> IITMInvocation?` and `closeInvocation(invocation: IITMInvocation)`. Since both methods should be accessible by the application developer, the access modifier is set to public. The following listing demonstrate the implementation of both methods.

Listing 6.4 IITMAgent Instrumentation Methods

```
public func trackInvocation(function: String = #function, file: String =
    #file) -> IITMInvocation? {
    if self.optOut == false {
        let invocation = IITMInvocation(name: function, holder: file)
        invocationOrganizer.addInvocation(invocation: invocation)
        return invocation.id
    } else {
        return nil
    }
}

public func closeInvocation(invocation: IITMInvocation) {
    if self.optOut == false {
        invocationOrganizer.removeInvocation(invocation: invocation)
    }
}
```

First of all, the functions check whether the agent is opted out. If it is the case, these invocations has no effects. In the other case, assuming that `trackInvocation()` was called, the agent creates an `IITMInvocation` object. The instantiated object requires the automatically assigned arguments of the method. The argument function, holds the name of the method that is currently instrumented. The variable file retrieves the file name, where the function is located in. The invocation constructor sets span-based attributes to the instance. The details of the `IITMInvocation` class are described in Section 6.1.6. The next instruction, passes the invocation object to the invocation organizer. Dependent on the chosen management strategy, the invocation organizer distributes the invocation object to the data structures which are implemented and the correlations between already collected spans are established. In the case, the developer chooses to save only invocation logs instead of the whole object, the invocation organizer converts the instance to a String. At the end, the invocation object is returned, which is needed to finish the instrumentation when calling `closeInvocation(invocation: IITMInvocation)`. By closing an invocation, the agent notifies the invocation organizer

to remove the invocation from the data structure. Furthermore, the removed invocation passes other stages, if required, such as data serialization and eventual data dispatch.

IITMAgent - Use Case Mapping

In cases the application desires to instrument whole use cases, the agent class needs to implement an other functionality for the code-based instrumentation. The reason for a different approach, is that use cases can be defined globally and could contain more than one method invocation. An other reason, is that several uses cases may be defined and instrumented sequentially, nested and parallel, and the agent is not able to recognize the dependencies between different use cases, due to the fact that use cases can be defined in many different ways from the application developer. As a consequence the agent is not able, to start a use case for instance from `method A()` and to close it from `methodB()`, with the tracing approach, by using the local instantiation values of `methodA()` without passing them to `methodB()`. The `IITMAgent` class for the use case mapping strategy provides two solutions for these problems.

The first solution is to create and identify use cases through their names. For this solution the developer needs firstly to create a root use case, by calling `trackRootUseCase(name: String) -> IITMUseCase` and passing a name. Nested use cases can be created and monitored by calling `trackUseCase(name: String, root: String)`. Since a nested use case could contain an other nested use case, the agent has to provide an other method for realizing the deeper span correlation. In that case the developer calls, `trackUseCase(name: String, parent: String, root: String) -> IITMUseCase`. In all three cases, the agent creates an `IITMUseCase` instance and passes it to the organizer which holds the spans and establishes correlations, based on the passed arguments, between other use case spans. The following source code part, Listing 6.6 demonstrates the implementation of the mentioned methods. Since we are tracking uses cases in this case, we renamed the `IITMInvocationOrganizer` instance to `spanOrganizer`. For stop tracking a use case, the developer also has to pass the defined names for the use cases. Dependent on the organization of the buffered use cases, passing more information for closing a use case would improve the performance of the agent. The reason is, that the agent may be able to fetch the use case in a more targeted way instead of propagating through all buffered spans. The negative aspect of this approach is that root use cases are not allowed to have the same name. This restriction emerges, when closing use cases. If two or more use cases are named the same, the agent would probably close the wrong use case, because use cases are only recognizable from their names with this approach. The other solution bypasses this constraint, but requires from the user to store the use case ids in global variables.

Listing 6.5 IITMAgent Starting Use Case

```
func trackRootUseCase(name: String) -> IITMUseCase? {
   retrun trackUseCase(name: name, parent: name, root: name)
}

func trackUseCase(name: String, root: String) -> IITMUseCase? {
   retrun trackUseCase(name: name, parent: root, root: root)
}

func trackUseCase(name: String, parent: String, root: String) ->
   IITMUseCase? {
   if self.optOut == false {
      let useCase = IITMUseCase(name: name, parent: String, root: root)
      spanOrganizer.addUseCase(useCase: useCase)
      return useCase
   }
   return nil
}
```

Assuming that `trackRootUseCase(name: String)`, `trackUseCase(name: String, root: String)` and `trackUseCase(name: String, parent: String, root: String)` are returning the id of the created use case instance, the developer could have a different opportunity to close the opened spans. In this case, the developer saves the returned value, which is the UUID of the use case, in a global variable to reuse the id afterwards. Instead of passing strings related to the use case names, the application developer could pass the stored use case ids for stop tracking a use case. As a result, the agent or the span organizer has the ability to select a use case to close, without reading the name of it. Consequently, with this solution, the developer is allowed to create different use cases named the same.

Listing 6.6 IITMAgent Closing Use Cases

```
func closeUseCase(useCase: IITMUseCase) {
   if self.optOut == false {
      spanOrganizer.removeUseCase(useCase: useCase)
   }
}

func closeUseCase(useCase: IITMUseCase) {
   if self.optOut == false {
      spanOrganizer.removeUseCase(useCase: useCase)
   }
}

func closeUseCase(name: String, parent: String = "", root: String) {
   if self.optOut == false {
      if parentName == "" {
         spanOrganizer.removeUseCase(name: name, root: root)
      } else {
         spanOrganizer.removeUseCase(name: name, parent: parent, root:
    root)
      }
   }
}
```

IITMAgent - Call Stack Sampling

While tracing, method swizzling and the use case mapping approaches require instrumentation methods, call stack sampling follows an other approach as defined in Section 5.2.4. In order to realize the functionality of this approach, we had to extend the `IITMAgent` class. For this approach we need to collect repeatedly the call stack of all threads in a specific time interval. For this reason we added a data collector, named `IITMDataCollector`, in our agent configuration. The data collector has the task to make sure that the call stacks are retrieved periodically. Listing 6.7 demonstrates the extension of `IITMAgent` in order to perform call stack sampling. The agent starts collecting instrumentation data, when `start(period: Double)` is invoked.

Listing 6.7 IITMAgent Call Stack Sampling Extension

```
var dataCollector: IITMDataCollector

func start(period: Double = IITMDataCollector.DEFAULT_PERIOD) {
   self.dataCollector.startCollection(period: period)
}
```

The following sections describe the `IITMDataCollector` and the `IITMThreadSignalHandler` classes in detail, which are only used, but essential, when the agent performs call stack sampling for gathering source code-based data.

6.1.2 IITMDataCollector

The data collector instance has the task to perform a call stack call repeatedly. As mentioned in Section 5.2.4 iOS does not provide a native method for printing the stack traces of all threads. We implemented an approach that has nearly the same result as the original call stack sampling approach invented from Jyoti Bansal and Bhaskar Sunkara [BS14]. The data collector starts a new thread, which executes a while loop. The boolean value of the condition of the while loop is dependent from the attribute `loopCondition`. By default, `loopCondition` is set to true, which means that the new thread starts an endless loop. Since the sampling of the call stack should be performed in a certain frequency, the while loop implements `Thread.sleep(forTimeInterval: TimeInterval)`. The type `TimeInterval` is a type alias of `Double`, and defines as argument, the time in seconds the current thread has to sleep. The developer is able change the sleep duration, by changing the value of the class variable `PERIOD`. Within the while loop, the data collector, collects all current running threads as `pthreads` and informs the signal handler, to set a specific signal for each of the collected threads. The collection of the call stacks can be stopped by calling `stopCollection()`. Stopping the data collection means, to set `loopCondition` to false. In consequence, when `loopCondition` is set to false, the thread hold by `IITMDataCollector` will be set to nil. The source code containing the explained methods and required attributes is shown in Listing 6.8.

Listing 6.8 IITMAgent Call Stack Sampling Extension

```
var thread: Thread?
static var LOOP_CONDITION = true
static var PERIOD = 0.03
static let DEFAULT_PERIOD = 0.03

func startCollection(period: Double) {
   IITMDataCollector.PERIOD = period
   thread = Thread(block: {
      let threadController = IITMSCThreadController()
      Calling_Thread = pthread_self()
      while IITMDataCollector.LOOP_CONTIDION {
         let threads = threadController.fetchThreads()
         for var t in threads {
            threadController.setSignal(pthread: &t)
         }
         Thread.sleep(forTimeInterval: IITMSCCollectionThread.PERIOD)
       }
       self.thread = nil
   })
   thread?.start()
}
```

6.1.3 IITMThreadSignalHandler

The signal handler has the ability to fetch all currently active threads with `fetchThreads() -> [pthread_t]`. The threads can be fetched through the kernel related port of the Mach operating system. The function `task_threads(_ target_task:task_inspect_t,_act_list:UnsafeMutablePointer<thread_act_array_t?>!,_act_listCnt:UnsafeMutablePointer<mach_msg_type_number_t>!)` retrieves the running threads as mach threads, and buffers them in the reference passed for the argument `act_list` as an array of mach threads. Afterwards, the signal handler, traverses all threads an converts them to `pthread_t` thread references by calling `pthread_from_mach_thread_np(_:mach_port_t)->pthread_t?`. The converted pthreads are added to the local variable `threadList`, which is returned at the end of the method. The signal handler class has also the ability to set a signal action. First of all, for making sure a thread reacts to a certain signal, we have to adapt the signal info of the thread and define on which signal the thread has to react. In our implementation we chose the signal `SIGALRM`. Afterwards the handler sets a signal function, which should be invoked, in cases

of a raised signal, to the thread. At the end `setSignal(pthread: inout pthread_t)` sends `SIGALRM` signal to the passed thread.

Listing 6.9 IITMAgent Call Stack Sampling Extension

```
func setSignal(pthread: inout pthread_t) {
   Target_Thread = pthread
   var sinfo = siginfo_t()
   sinfo.si_signo = SIGALRM
   var sAction = sigaction()
   sAction.__sigaction_u.__sa_sigaction = signalFunction(sig:siginfo:p:)
   sAction.sa_flags = SA_SIGINFO
   let _ = withUnsafeMutablePointer(to: &sAction) {
      sigaction(SIGALRM, $0, nil)
   }
   if pthread != pthread_self() && pthread_kill(pthread, SIGALRM) == 0 {
      // SIGALRM signal sent, otherwise print pthread reference
      print(pthread)
   }
}
```

As one can see, in line six of Listing 6.9, `setSignal()` passes the call back function as signal action named `signalFunction(sig:siginfo:p:)`. This function is defined in the same file, but as a global public function. The reason for this is that the pthread signal function needs to conform to a C function.

6.1.4 IITMSpanOrganizer

Managing tasks of collected measurements are mostly performed by the invocation organizer. The instance of this class contains the desired data structure for buffering the collected spans and the functionalities to establish intelligent relations between different spans. This subsection will also argue with implementation strategies used for specific data collection approaches, since the data holding and correlation logic has to differ for certain circumstances. We decided to implement the complete logic for all possible management strategies, since a data collection strategy can refer to one or more organization concepts, for each bundle. At the end of the evaluation, the default organization strategy will be the most suitable one dependent on the evaluation results. Other possible strategies should then be usable by calling the extended constructor of this class and by passing the respective enumeration. To accomplish the mentioned functionality we extended `IITMSpanOrganizer` with two enumerations. The first one

describes the possible data structures. The other one the data type, which has to be buffered. Since the span organizer has to populate the given buffer, this class needs to contain methods such as `addSpan(span: IITMSpan)` and `removeSpan(span: IITMSpan)`. When the organizer adds a span, it has to set relations between spans if possible. For this reason, the organizer also has to contain a method to set relations such as `setRelation(child: IITMSpan, parent: IITMSpan)`. In the case a span is recognized as a root invocation, the organizer has to set the given span as a root span. For this case the organizer also has to implement `setRoot(span: IITMSpan)`. To summarize the mentioned functionalities, Listing 6.10 demonstrates an abstract implementation of `IITMSpanOrganizer`.

Listing 6.10 Abstract Definition of IITMSpanOrganizer

```
class IITMSpanOrganizer: NSObject {
   var dataModel: IITMSpanOrganizer.IITMDataModel
   var dataStructure: IITMSpanOrganizer.IITMDataStructure

   enum IITMDataModel {}
   enum IITMDataStructure {}

   func addSpan(span: IITMSpan) {}
   func removeSpan(span: IITMSpan) {}

   func setRelation(child: IITMSpan, parent: IITMSpan) {}
   func setRoot(span: IITMSpan) {}
}
```

Each agent bundle organization class extends from `IITMSpanOrganizer`. In the following subsections, we present in detail the extensions related to each data collection strategy. In addition to that, we will argue with the possible and implementable management concept dependent on the collection strategy.

IITMInvocationOrganizer - Tracing & Method Swizzling

Agent bundles supporting tracing for collection source code-based data are implementing `IITMInvocationOrganizer` as extension of `IITMSpanOrganizer`. Tracing is able to support a data buffers as a map or stack. The tree data structure was excluded, due to performance deficits based on the operational time. The supported data types to be buffered are both invocation messages as strings or invocation instances. Since the

invocation container supports two different data structure approaches and two different data model types, it has to contain by default four different buffer with different types.

Listing 6.11 IITMInvocationOrganizer Attributes

```
class IITMInvocationOrganizer: NSObject {
   var dataModel: IITMInvocationOrganizer.IITMDataModel
   var dataStructure: IITMInvocationOrganizer.IITMDataStructure
   enum IITMDataModel {
      case message, instance
   }
   enum IITMDataStructure {
      case map, stack, tree
   }
   // Map data structure
   map: [UInt64: IITMSpan]?
   mapMessage: [UInt64: String]?
   childMap: [UInt: IITMSpan]?
   childMapMessage: [UInt: String]?

   // Stack data structure
   stacks: [UInt: [IITMSpan]]?
   stacksMessage: [UInt: [String]]?

}
```

Listing 6.11 demonstrates the implementation of the needed attributes for buffering spans. In the following we describe the implementations of the abstract methods defined in Section 6.1.4. Each method has to recognize the used organization option and perform different functionalities than other ones. For instance, adding a new span has to perform a lookup for determining which strategy is used to buffer the new span.

Listing 6.12 IITMInvocationOrganizer Adding Spans

```
func addSpan(span: IITMInvocation) {
   switch dataStructure {
      case .map:
         addSpanToMap(span: IITMInvocation)
         break
      case .stack:
         addSpanToStack(span: IITMInvocation)
         break
   }
}
```

Each of the nested methods called from `addSpan(span: IITMInvocation)` defined in Listing 6.12 are checking which data model is used for buffering. Listing 6.13 shows the implementation of organizing spans to a map. This strategy is only performable by using a separate child object. The agent needs this object in order to correlate spans and to recognize which invocation is the one deepest on the active execution trace. The child objects are hold in a map to differentiate the child spans for each active thread. If the child reference is nil, the agent sets the span to a root span. Otherwise it correlates both spans, and changes the child object with the new span.

Listing 6.13 IITMInvocationOrganizer Adding Spans to a Map

```
func addSpanToMap(span: IITMInvocation) {
   switch dataModel {
      case .instance {
         if var parent = childMap?[span.threadId] {
            setRelation(child: &span, parent: &parent)
         } else {
            setRoot(span: &span)
         }
         childMap?[span.threadId] = span
         map?[span.id] = span
         break
      }
      case .message {
         var spanMessage = ""
         if var parent = childMapMessage?[span.threadId] {
            spanMessage = getSpanMessage(span: span)
            setMessageRelation(child: &spanMessage, parent: &parent)
         } else {
            setRootMessage(span: &spanMessage)
         }
         childMapMessage?[span.threadId] = spanMessage
         mapMessage?[span.id] = spanMessage
         break
      }
   }
}
```

The following listing, Listing 6.14, demonstrates the approach when the agent buffers the measured spans in stacks. This approach is similar to the one described before. Each stacks are hold in a map to be differentiated between their respective threads. As done when buffering on a map, the agent has to look, whether it has to store messages of spans or the span instance. Afterwards, the agent tries to access to the last object of the stack. If there is no stack to be retrieved, the agent will create a new stack for the current thread. In case the retrieved stack is empty, the new span will be set as root. Otherwise, the accessed span will be set as the parent of the new one.

Listing 6.14 IITMInvocationOrganizer Adding Spans to a Stack

```
func addSpanToStack(span: IITMInvocation) {
   switch dataModel {
      case .instance:
         if var stackTrace = stacks?[span.threadId] {
            if var parent = stackTrace.last {
               setRelation(child: &span, parent: &parent)
            } else {
               setRoot(span: &span)
            }
         } else {
            stacks?[span.threadId] = [IITMInvocation]()
            setRoot(span: &span)
         }
         stacks?[span.threadId].append(span)
         break
      case .message:
         var spanMessage = ""
         if var stackTrace = stacksMessage?[span.threadId] {
            var parentMessage = stackTrace.last
            spanMessage = getSpanMessage(span: span)
            setMessageRelation(child: &spanMessage, parent: &parentMessage)
         } else {
            stacksMessage?[span.threadId] = [String]()
            setRootMessage(span: &spanMessage)
         }
         stacksMessage?[span.threadId].append(spanMessage)
         break
   }
}
```

6.1.5 IITMSpan

This class represents a simplified span object based on opentracing. A span object holds a unique id, a parent id which refers to the span caller and a trace id which refers to the execution trace. In addition to that each span object possess the creation time and the duration of the span execution.

Listing 6.15 IITMSpan Class

```
public class IITMSpan: NSObject {

    var id : UInt64
    var parentId : UInt64
    var traceId : UInt64
    var startTime : UInt64
    var duration : UInt64

}
```

6.1.6 IITMInvocation

An invocation object inherits from a span object. Beside span attributes, an invocation objects hold method-based attributes such as the method name, the thread name where the method was invoked and the related thread id. Listing 6.17 shows a snippet of the `IITMInvocation` class.

Listing 6.16 IITMInvocation Class

```
public class IITMInvocation: IITMSpan {

    var name : String
    var threadName : String
    var threadId : UInt

}
```

6.1.7 IITMUseCase

A use case object inherits from a span. Since use cases can be traced globally, the thread id is not an important attribute in order to trace use cases. The only additional attribute a use case holds is the use case name.

6.1.8 IITMRemoteCall

A remote call object inherits from an invocation or a use case, dependent on the agent configuration. Remote call objects are defined in order to trace back end requests. In

addition to that, we have to take in account, that a remote call instance has to collect important data for further analysis. Important attributes are the geo location of the end-user, the used mobile provider, the connectivity and the SSID. The mentioned metrics are collected in request and response time. Additional attributes are the response code, the request URL, if a timeout was fired and the HTTP method.

Listing 6.17 IITMRemoteCall Class

```
public class IITMRemoteCall: IITMInvocation {

    var url : String
    var timeout: Bool
    var responseCode: Int
    var startPosition: CLLocationCoordinate2D
    var endPosition: CLLocationCoordinate2D
    var startProvider: String
    var endProvider: String
    var startConnectivity: String
    var endConnectivity: String
    var startSSID: String
    var endSSID: String
    var httpMethod : String

}
```

6.1.9 Automated Remote call Instrumentation

This section describes how the agent configurations are able to monitor remote calls out of the box. In Section 3.2.2 we explained which processes are performed in the background for a remote call. In order to instrument remote calls, the agent hooks `URLSession` methods, which are responsible for remote calls. The new implementation of `URLSession.dataTask(with:completionHandler:)` is shown in Listing 6.18. In the first part of this function, a the `URLRequest` object is converted to a `NSMutableURLRequest` object in order to add header attributes. The new header attributes are needed for the back end correlation of the measured spans. Afterwards, a remote call object is created, which starts tracing the remote call. The next step is to call the real implementation of `URLSession.dataTask(with:completionHandler:)` by calling `URLSession.iitmDataTask(request:completionHandler:)`. This invocation starts the real remote call. When the server responds, the created remote call will be closed.

If the application developer passed a callback function, this function will be invoked afterwards and the nested invocations will be traced.

Listing 6.18 Automated Remote call Instrumentation

```
func iitmDataTask(request: URLRequest, completionHandler: ((Data?,
    URLResponse?, Error?) -> Void)? = nil) -> URLSessionDataTask {
    let agent = IITMAgent.getInstance()
    var req: NSMutableURLRequest = (request as NSURLRequest).mutableCopy()
    as! NSMutableURLRequest
    var remotecall: IITMRemoteCall? = nil
    if req.url?.absoluteString != IITMAgentConstants.HOST {
        remotecall = agent.trackRemoteCall(url: (req.url?.absoluteString)!)
        agent.injectHeaderAttributes(remotecall: remotecall!, request: &req)
    }

    let dataTask = iitmDataTask(request: req as URLRequest,
    completionHandler: {data, response, error -> Void in
        var invocation: IITMInvocation? = nil
        if remotecall != nil || req.url?.absoluteString !=
    IITMAgentConstants.HOST {
            agent.closeRemoteCall(remotecall: remotecall!, response:
    response, error: error)
        }
        if completionHandler != nil {
            if req.url?.absoluteString != IITMAgentConstants.HOST {
                invocation = IITMAgent.getInstance().trackInvocation()
            }
            completionHandler!(data, response, error)
            if req.url?.absoluteString != IITMAgentConstants.HOST {
                agent.closeInvocation(invocation: invocation!)
            }
         }
    })
    return dataTask
}
```

6.2 Agent Configurations

This chapter summarizes all the developed agent strategies and defines the combination of various approaches, in terms of data collection, management and dispatch, as an agent configuration. The following section lists the implemented agent strategies. Section 6.2.2 illustrates all developed mobile agent modules and integrates them into a feature model. The feature model describes which agent concept is combinable and suitable with other ones. The last section of this chapter defines the agent configuration set AC and lists all resulting agent configuration instances $ac_i \in AC$ with a short explanation of the used modules.

6.2.1 Agent Strategies

This section lists all the implemented mobile agent concepts. The strategies are divided in the various working phases of the agent software. The first sub section argues with data collection strategies. The second sub section with data management approaches and the last one with data dispatch concepts. Additionally, the following subsections will mention the related class which implements the named strategy.

Data Collection Strategies

1. Tracing
   (related classes: IITMAgent)
2. Method Swizzling
   (related classes: IITMAgent and files performing method swizzling)
3. Use Case Mapping
   (related classes: IITMAgent)
4. Call Stack Sampling
   (related classes: IITMAgent, IITMCollectionThread and IITMThreadController)

Data Management Strategies

1. Data Organization
   a) Map & Last Child Object
      (related classes: IITMInvocationOrganizer or IITMSpanOrganizer)

   b) Stacks
   (related classes: IITMInvocationOrganizer or IITMSpanOrganizer)

2. Data Model

   a) Span Instance
   (related classes: IITMInvocationOrganizer or IITMSpanOrganizer, IITMInvocation, IITMRemoteCall)

   b) Message & Remote Call Instance
   (related classes: IITMInvocationOrganizer or IITMSpanOrganizer, IITMMessageUtil, IITMRemoteCall)

   c) All Message
   (related classes: IITMInvocationOrganizer or IITMSpanOrganizer, IITMMessageUtil, IITMRemoteMessageUtil)

Data Dispatch Strategies

1. Single Span
   (related classes: IITMAgent, IITMInvocationOrganizer or IITMSpanOrganizer, IITMDataSerializer and IITMRestManager)

2. Closed Trace
   (related classes: IITMAgent, IITMInvocationOrganizer or IITMSpanOrganizer, IITMDataSerializer and IITMRestManager)

3. Periodical Check
   (related classes: IITMAgent, IITMMetricsController, IITMInvocationOrganizer or IITMSpanOrganizer, IITMDataSerializer and IITMRestManager)

6.2.2 Agent Feature Model

As shown in the feature model [FEA90], illustrated in Figure 6.2, collecting, managing and dispatching monitoring data are mandatory features. Analyzing the collected and locally buffered monitoring data is an optional. In order to collect monitoring data, an agent developer can choose between tracing, hooking methods or sampling the call stack. The data manager has to implement a data creator and a buffer in order to hold the collected measurements locally and establish correlations between monitored invocations. The options to dispatch monitoring data are sending single spans, sending complete traces or sending a various amount of spans periodically.

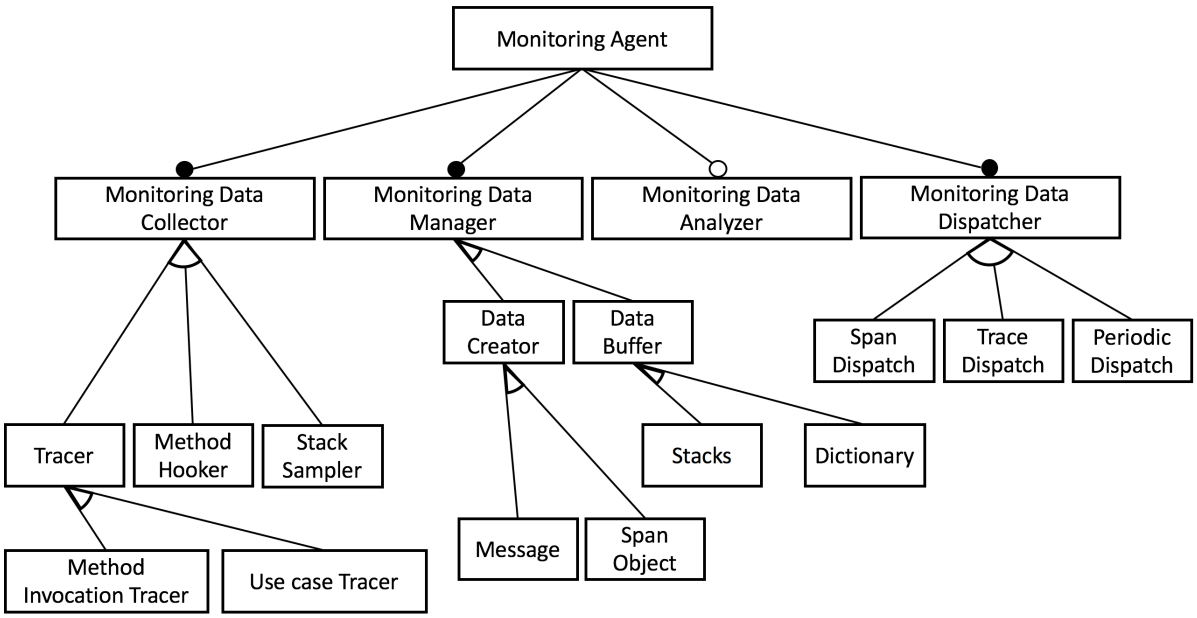


Figure 6.2: Feature Model for a Monitoring Agent

6.2.3 Combination of Strategies

In the following table, Table 6.1, we present the combination possibilities of our agent strategy implementations. Two strategies are allowed to be combined for an agent configuration if the cell of both strategies contains the value X.

	Trace Data Collection	Data Management				Data Dispatch		
	Method Swizzling (System Libraries)	Instance	Message	Stacks	Map	Single Measurement	Closed Trace	Periodic
Tracing	X	X	X	X	X	X	X	X
Method Swizzling	-	X	X	X	X	X	X	X
Call Stack Sampling	X	X	-	X	-	X	X	X
Use Case Tracing	-	X	X	X	-	X	X	X

Table 6.1: Strategy Combination Matrix

Chapter 7

Evaluation

This chapter will firstly introduce the evaluation approaches for the various iOS mobile monitoring strategies. The evaluation chapter is mainly made of five parts. The first part of this chapter, Section 7.1, describes the evaluation goals. The second part of this chapter, Section 7.2, argues with the theoretical evaluation. For the theoretical evaluation we will compare the different monitoring concepts defined in Chapter 5 in terms of runtime complexity and memory usage theoretically. For this task we will rely on the used algorithms for the various mobile agent tasks and on the calculated memory allocation of the utilized agent and monitoring data objects. The third part of this chapter, Section 7.3, presents the experimental evaluation. The first part of the experimental evaluation section describes the experimental settings of the various practical evaluation experiments. Afterwards, the results of the different experiments are presented. The fourth part of the evaluation chapter, Section 7.4, will discuss about the evaluation results and the Section 7.5, Threats to Validity, completes this chapter.

7.1 Evaluation Methodology & Goals

For instrumenting the source code of a mobile application, the mobile agent has to be imported to the application that has to be monitored. Source code-based data and resource usage data of the mobile application can be fetched by the mobile agent. Through agent invocations, the mobile agent is able to collect method invocations and the respective execution times, to correlate the invocations in an execution tree, to collect the hardware workload and to serialize the measured probes for the later data dispatch. Since the iOS mobile agent runs within the mobile application that is monitored, the performance of the application itself will decrease. One has to take in consideration that by adding the agent framework and by adding instrumentation points, the execution

times of methods will probably increase. Therefore the resource workload of the agent itself has to be as low as possible.

In the phase of the theoretical evaluation we will argue with the theoretical operational time of the used algorithms and with the memory usage of the theoretically calculated object sizes that has to be buffered locally. The theoretical evaluation is needed to reduce the number of experiments for the practical evaluation.

For the phase of the experimental evaluation, we have to define the experimental setup. The practical evaluation requires a set of open source applications which belong to the application classes defined in Chapter 3. In addition to that, the practical evaluation also requires an implemented set of various agent configurations, which are defined in Section 6.2. In the phase of the practical evaluation, we will profile the selected open-source applications by performing specific predefined use cases firstly without and afterward with the integrated iOS mobile agent configurations. In this phase, we will document the source-code based information retrieved from the mobile agent configuration and compare the outcome with the expected one. In addition to that we will document the execution times of the instrumented methods and compare them with the execution times of the same methods but without instrumentation. We will also document the amount of memory overhead produced by the various agent configurations.

A more detailed experimental setup definition is presented in Section 7.3.

With the evaluation we want to investigate whether certain agent properties are more suitable for specific application types. Therefore the research question is:

RQ1: Does an agent function with a certain strategy perform better on different application types?

Since the produced overhead of the agent configuration is measurable it is also important to question:

RQ2: How much does each strategy affect the application performance?

Both questions will be answered based on the evaluation results. We will measure the overhead produced by the agent configurations based on the listed criteria:

- Agent integration time
- Agent initialization time
- Hooking overhead per invocation
- Tracking time of methods or use cases
- Tracking time of the complete stack trace

- Closing time of the tracked invocation
- Span serialization time
- Increased application size
- Additional memory usage

The agent integration time measures the time spent by the developer in order to integrate the agent configuration. Integrating an agent includes to add the agent framework to the application and to re-factor the source code to ensure that the agent is working. Spending more time for integrating the agent means that it is more difficult to monitor the application with this agent strategy.

Criterion two, agent initialization time, argues with the time spent in order to build the agent within the application. Since it done one once in the application life time, the importance of this criterion is lower than other ones.

The response time of the monitored application is affected by the following criteria: hooking overhead per invocation, tracking time of methods or use cases, tracking time of the complete stack trace, closing time of the tracked invocation and span serialization time. Tracing based monitoring strategies are increasing the application response time when start tracing and when end tracing methods or use cases. In some cases, the agent has also to serialize a number of measurements when some methods are traced. As shown in Figure 7.1, the mentioned agent activities are producing the overhead in response times.

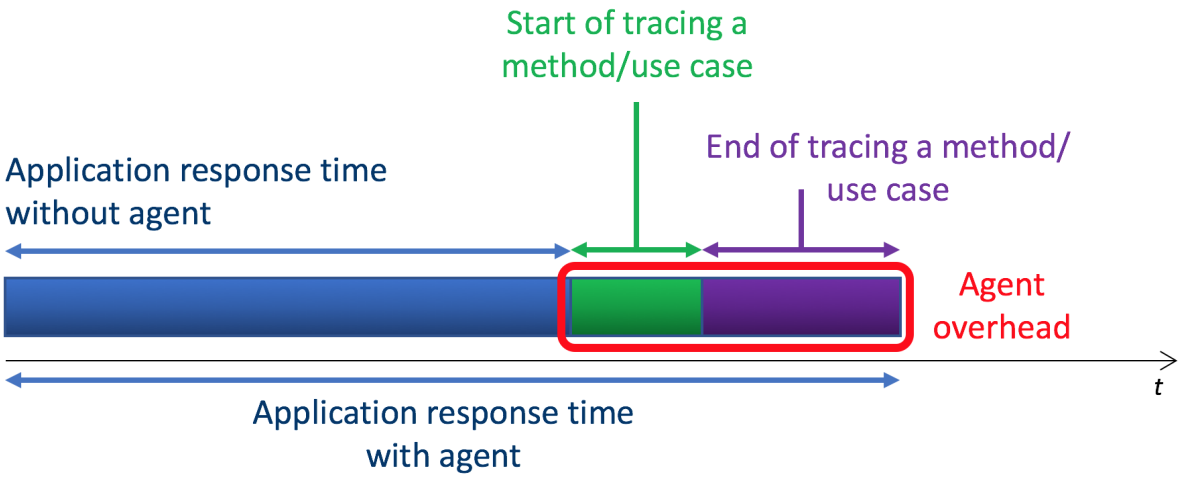


Figure 7.1: Response times change with tracing-based agent configurations

Stack sampling based approaches affects the application response time periodically. In the case of the stack sampling strategy we implemented, the performed user interaction with the application get enqueued until the sample iteration is finished.

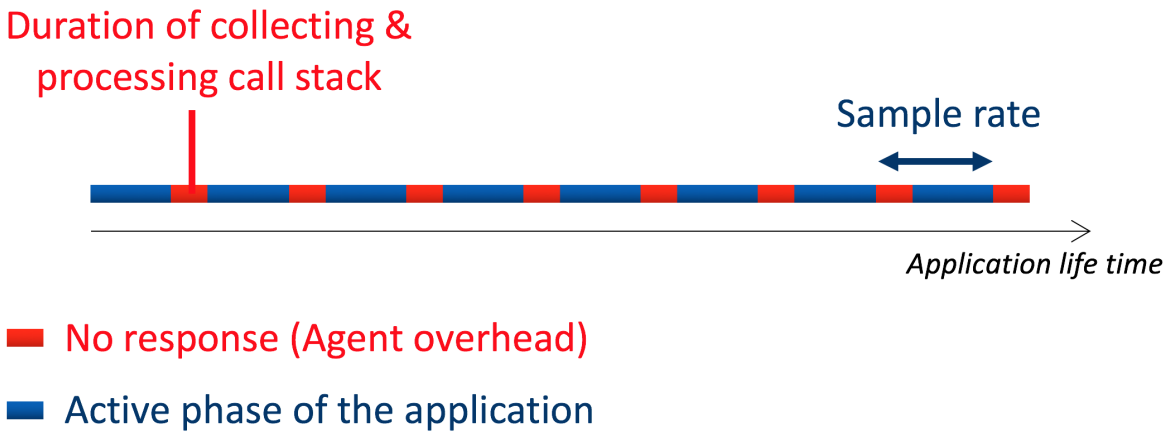


Figure 7.2: Performance overhead of the Stack Sampling-based agent

In the case the user interaction gets enqueued, the application does not respond for a certain time as illustrated in Figure 7.2. If the response time for a certain action exceeds one time period of the active phase, the response time is increased by the duration of collecting and processing the call stack. Therefore, the time of collecting the stack trace and of processing the data should be as minimal as possible.

The last two criteria, increased application size and additional memory usage argue with the memory overhead produced agent configurations. Since we are adding a framework to our application it is trivial that the application size increases. More important is to look how much memory the agent allocates in runtime. We have also to mention that the amount of used memory is related to the amount of measurements done by the agent.

7.2 Theoretical Evaluation

This part of the evaluation argues with different agent strategies of the same working phase. The results of this evaluation will influence the practical part of the evaluation. Since the amount of the configurable iOS agents is too high, the results of the theoretical evaluation will minimize the number of agent configurations to be considered in the practical evaluation. The theoretical evaluation can and will only confront strategies derivating from the same working phase of the agent. For instance, considering the phase of which object type has to be buffered, the strategy which are compared will be span instances, span messages, and span messages with remote call objects. The main evaluation criteria for this part are the usability, the memory usage and the execution times of the utilization of the compared structures or objects. The first part of this evaluation compares the data model options. The second part confronts the buffering structures. The following section argues with the data dispatch strategies and the end of this section summarizes the results.

7.2.1 Evaluation of Data Model Options

In total three different approaches are defined. The first option is to store span instances. The second one is to map the span instances to string messages. The last option is to buffer string messages of method invocation spans and to buffer span objects for remote calls.

Considering all monitoring approaches there are three different span instances to be examined. The agents differ between method invocation spans, remote call spans and use case spans. The respective classes of those three data models are inheriting from `IITMSpan`. We defined the four classes in Section 6.1.5. The absolute instance size of a class can be measured by calling the Objective-C runtime function `class_getInstanceSize(_ cls: AnyClass?) -> Int`, which returns the allocated size in bytes. In order to perform this measurements, we created a new project which implements the mentioned classes and which retrieves the object sizes.

When running the test project, as shown in Listing 7.1, the console prints that an `IITMInvocation` instance requires **80 bytes**, a `IITMRemoteCall` object **296 bytes** and a `IITMUsecase` instance **72 bytes**.

A complete invocation message buffered as a string with the format shown in below, will allocate at least **102 bytes** in the case we convert the numeric values in hexadecimals.

	16	(id)
+	16	(start time)
+	16	(parent id)
+	16	(trace id)
+	16	(thread id)
+	16	(duration)
+	6	(spaces)
+	n	(span name)
>	102	bytes

The amount of allocated bytes for a remote call message would be at least **192 bytes**.

	16	(id)
+	16	(start time)
+	16	(parent id)
+	16	(trace id)
+	16	(thread id)
+	16	(request longitude)
+	16	(request latitude)
+	4	(request network connection)
+	1	(request timeout)
+	s1	(request SSID)
+	p1	(request provider)
+	16	(response longitude)
+	16	(response latitude)
+	4	(response network connection)
+	1	(response timeout)
+	s2	(response SSID)
+	p2	(response provider)
+	3	(response code)
+	16	(duration)
+	19	(spaces)
+	n	(span name or URL)
>	192	bytes

For a use case message at least **85 bytes** are required.

	16	(id)
+	16	(start time)
+	16	(parent id)
+	16	(trace id)
+	16	(duration)
+	5	(spaces)
+	n	(span name)
>	85	bytes

Listing 7.1 Test Program for retrieving the Instance Sizes

```
override func viewDidLoad() {
   super.viewDidLoad()
   print(class_getInstanceSize(IITMInvocation.self))
   print(class_getInstanceSize(IITMRemoteCall.self))
   print(class_getInstanceSize(IITMUsecase.self))
}
```

Invocation Message format:
`id␣startTime␣parentId␣traceId␣threadId␣name␣duration`

Remote call Message format:
`id␣startTime␣parentId␣traceId␣threadId␣url␣requestLongitude␣`
`requestLatitude␣requestNetworkConnection␣requestTimeout␣requestSsid␣`
`requestNetworkProvider␣responseLongitude␣responseLatitude␣`
`responseNetworkConnection␣responseTimeout␣responseSsid␣`
`responseNetworkProvider␣responseCode␣duration`

Use case Message format:
`id␣startTime␣parentId␣traceId␣name␣duration`

In addition to the allocated space we have to consider the execution times of further processes for data conversion and data parsing. Spans are utilized by the agent when starting a span, when correlating with other spans and when closing spans.

To sum up, since buffering spans as messages will not bring up benefits, on the contrary it will increase the execution times of the agent, it is not recommended to utilize invocation string messages.

7.2.2 Evaluation of Data Organization Strategies

Section 5.3.7 explained three different organization strategies. The first concept is using stacks, the second one is using a map with child objects and the third one a tree. Since the span organization is important for span correlation, we need a fast and performant look up, at least for the last child span. In addition to that, the expected amount of memory allocation dependent of the amount of buffered spans is also important. By considering the first criterion, we can exclude the organization with trees for the practical evaluation. The look up for the last child when using stacks is expected $O(1)$, when using a map $O(1)$ and when using a tree in worst case $O(n)$. The next criterion to be considered is the memory usage of the remaining data structures. In order to analyze the amount of used memory and the execution performance of both remaining organization structures, we implemented an other test project. The test project implements the functionalities to add spans to stacks or directly to a map like the span organizer does. On top of buffering the span objects, the tests also implement the functionality of correlating the mapped spans. For our test cases, we initialize in total 100,000 invocation objects split in 10 active execution traces. The first test buffers the objects into stacks, the second one into a map.

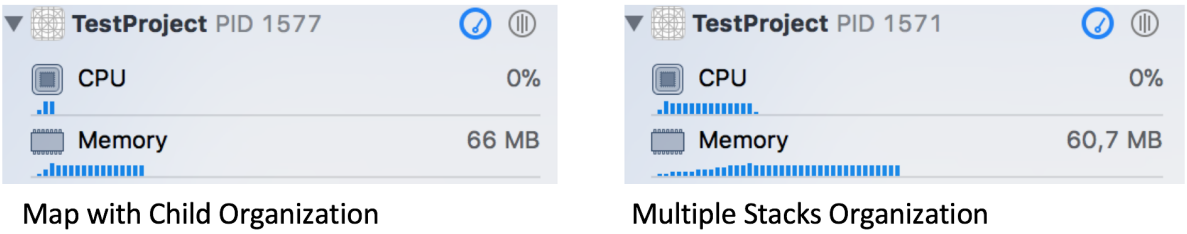
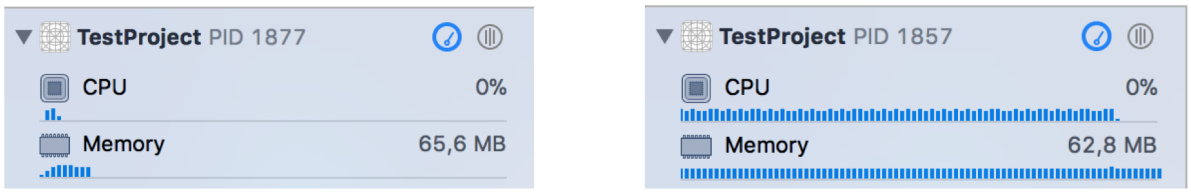


Figure 7.3: Test Results of the Organization Strategies

Figure 7.3 is demonstrating the profiling results of the tests. The amount of memory usage utilizing a map for storing active spans is 5.3MB higher than using the multiple stacks approach. On the other side, the population of the single map executed four times faster, which is noticeable by looking at the CPU workloads in Figure 7.3. Both tests were executed on an iPhone 8 Plus. We performed an other test, which simulates monitoring a single threaded application. Monitoring a single threaded application means that the span organization holds a single active execution trace at a time. For this second test we initialized again 100,000 spans all with the same trace and thread ID.



Map with Child Organization

Multiple Stacks Organization

Figure 7.4: Second Results of the Organization Strategies Test

Figure 7.4 presents the profiling results for the second tests. The memory usage of the organization with a map is 2.8MB higher than using the multiple stacks strategy. On the other hand, mapping data to map took **3 seconds** and populating the stacks **3 minutes**, which means that utilizing the dictionary data structure is 60 times faster than using multiple stacks.

Considering that the amount of memory overhead utilizing a dictionary over multiple dynamic stacks is ignorable small for the current technological time, it is recommended to provide an iOS agent which organizes the spans in a dictionary. It is also recommended due to its execution performance over the long propagation time required with multiple stacks. For the mentioned reasons, we will focus on the **span organization with a map** when performing the experiments for practical evaluation.

7.2.3 Evaluation of Data Dispatch Strategies

The dispatch options declared in Section 5.4.1 are performing all the same type of conversion process. The conversion process is explained in Section 5.4.3. The only difference between them is the dispatch signal. The dispatch signal determines indirectly the amount of spans that have to be serialized and sent to a monitoring back end. There are pro and contra arguments for all options. For instance, if the agent processes each span singularly the performance overhead for serializing and dispatching is split in small parts. If the agent waits for the signal for dispatching a complete trace, the agent has more spans to process but the size of the span document will be smaller than when creating multiple ones. The periodical span dispatch is a middle way of the first two options. It is expected that this option, will not process each single span but also will not wait until a span is completely executed. Performing the span dispatch periodically requires a parallel process in order to firstly determine whether there exists spans to be processed and sent. As a result, the execution times of the core application would not be affected directly from the dispatch process. In addition to that, the mobile agent could use the metrics sampling process in order to perform this task. Since a root span is always the last span of an execution trace that is closed, we have to consider that

created documents from dispatching single spans and from the periodical dispatch often will miss a root span. In that case the monitoring back end has to recognize this deficit and buffer the data until the root span arrives. In other cases the monitoring back end will probably pass wrong data sets to the client, and the monitoring client would render wrong execution traces. Since the dispatch options are dependent on the functionality of the monitoring back end, it is more important to focus on the document size, rather than on the dispatch options in future. If we consider the example JSON-object of Listing 5.4 and append five metrics measurement probes, the size of the document that has to be sent to the monitoring back end will reach around than **2.5kB**. The size of a **serialized invocation or use case span** is around **350 bytes**. **Serialized remote calls**, with the defined format of Section 5.4.3, are reaching **one kilobyte**. Even tough it is not strongly required to activate the dispatch constraint. If we scale the amount of spans up to 1,000, 900 invocations and 100 remote calls, which is a huge scale for an application session, the size of the document will reach less than 0.5MB. Due to time reasons and by taking account of that the differences between the span options are not enormous, we will focus on the dispatch option of **complete traces without constraint** for the practical evaluation.

7.3 Practical Evaluation

This section describes, in Section 7.3.1, the experimental setup of the practical evaluation in detail and presents the experiments and the corresponding results. The evaluation experiments are divided in main experiments and sub experiments. The main experiments are arguing with different applications of different classes. Each experiment presents the results based on the criteria defined in Section 7.1. One has to consider that due to time reasons and due to the huge scale of different agent configuration, it is not possible to practical evaluate each agent configuration. In Section 7.2, we theoretically excluded some agent strategies for some agent working phases in order to reduce the amount of tests and to focus only on the most performant implementations. The remaining main agent configurations (MAC), we focus on in the experimental part of the evaluation, are listed in the following:

1. MAC I: Tracing
   https://github.com/sassanmo/InstrumentITMobileTracer
2. MAC II: Method Swizzling (AOP)
   https://github.com/sassanmo/InstrumentITMobileAspects
3. MAC III: Call Stack Sampling
   https://github.com/sassanmo/InstrumentITMobileStackSampling

4. MAC IV: Use Case Mapping
   https://github.com/sassanmo/InstrumentITMobileUseCase

The main agent configurations will map spans as span instances. The *Tracing* and *Method Swizzling* approach will use a dictionary for organizing the spans. Main agent configuration III and IV will use special stack based constructs. For serializing and dispatching the monitoring data all configurations will use the complete trace dispatch without constraints. On top of these configurations, the most performant one will be combined with the instrumentation source code injector instrumentIT, which should improve the integration of the mobile agent by decreasing the integration time. This agent configuration is declared as:

1. MAC V: MAC X with Source Code Data Processor (instrumentIT)
   https://github.com/sassanmo/instrumentIT

7.3.1 Experimental Setup

This section describes the used approach for the practical evaluation. Overall, the practical evaluation is made of main experiments and sub experiments. Each main experiment is related to in sub experiments. The difference between each other is that main experiments rely on the application that has to be monitored and the sub experiments on the different mobile agent configuration. A main experiment presents the used application and defines the use case that is performed while profiling the application. Therefore the amount of main experiments is dependent on the number of the different application classes. In the case of this thesis, three. The sub experiments are their main experiment. The sub experiments will document the evaluation results based on the application class of the profiled application, on the predefined use case and based on the integrated agent configurations.

For instance we define an experiment named Experiment I: Application Class A as a main experiment. Experiment I holds a short description of the application, presents the use case that will later be performed by the application profiler and holds the expected execution traces by performing the use case. The sub experiments of Experiment I would be for instance, Experiment I.I MAC I, Experiment I.II MAC II, Experiment I.III MAC III and so on. Each sub experiment will present the documentation of the profiling results. In addition to that, sub experiments relying on agent configurations, will argue with data output of the agent, with the execution times of the monitored and agent methods and argue with the CPU, memory workload overhead produced by the agent configuration and also with the time spent to integrate the agent bundle into the application and the additional work that has to be performed by the developer.

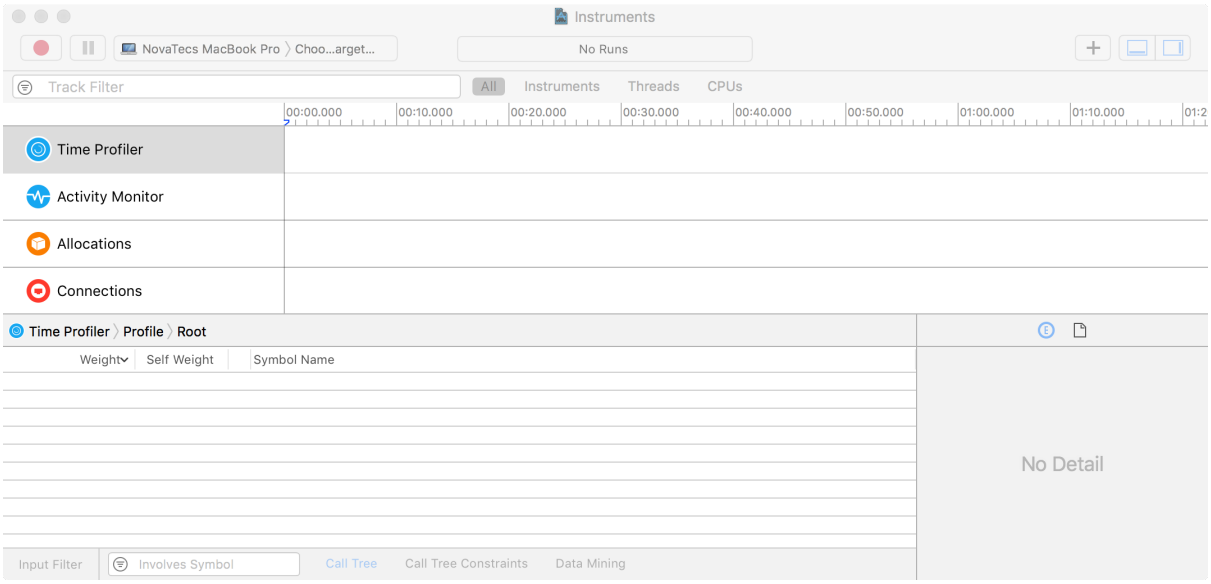


Figure 7.5: Instruments Profiling Plug-in

The required steps for the mentioned approach are the followings:

In the first stage we will define use-cases for each of the selected applications. In the next stage we will profile these application with the Xcode [XCO17] IDE and the Instruments [INS17] plug-in. These programs will help us monitoring performance data of each application without an agent integrated. Figure 7.5 presents the user interface of the profiling plug-in Instruments. As also shown in Figure 7.5, we will use the *Time Profiler*, the *Activity Monitor* and we will measure the *Allocations*. The *Time Profiler* samples the call stack of the profiling application, measures the execution times of the caught methods and calculates the percentage of the spent CPU workload. The *Activity Monitor* monitors the total CPU workload of the application in percentage and the absolute used physical memory. *Allocations* measures the amount and the size of all allocated objects on the heap while running the application and performing the use cases. In the next phase we will integrate each agent configuration to each application. In addition we will measure the spent time for this task. The fourth phase of the practical evaluation is made of profiling the applications with the same use-cases again. The difference lies in the fact that there is an agent configuration integrated in the application. Therefore we should detect a notable performance overhead while profiling.

The following sections will represent the various experiments and present the evaluation results.

7.3.2 Experiment I: Stand-alone mobile application

Used application: swift-2048 (https://github.com/austinzheng/swift-2048/)

Application description: The open-source iOS application, swift-2048, is a game playable by performing swipe gestures. The application is made of two view controllers. The first one introduces the user to the game. By clicking on the *Start game* button, the application loads and displays the second view controller. The view holds a squared field with 16 equal sized fields. In fields there are tiles with numbers. Swiping in one direction, let the tiles move to the swipe direction. If two neighbor tiles are containing the same number, they will be combined in one and the number will be doubled. When combining tiles, the value of the new tile will be added to the total score of the player.

Use case: The profiling use case is to start the application, to press the *Start game* button and to swipe three times in three different directions.

Expected execution trace: As shown in Figure 7.6 the overall absolute operational time to perform the experimental use case was 140.50ms. In line three of Figure 7.6 one can see that is the caller for initializing the new view controller. The view controller, *NumberTileGameViewController*, initializes the game model and loads the view. The function `viewDidLoad()` calls `setUpGame()`. In complete, the duration of the mentioned function was 10.6ms. Afterwards, after the first swipe, a timer handles the animation for the tiles. The profiling tool recognized a closure for the up, left and down swipe.

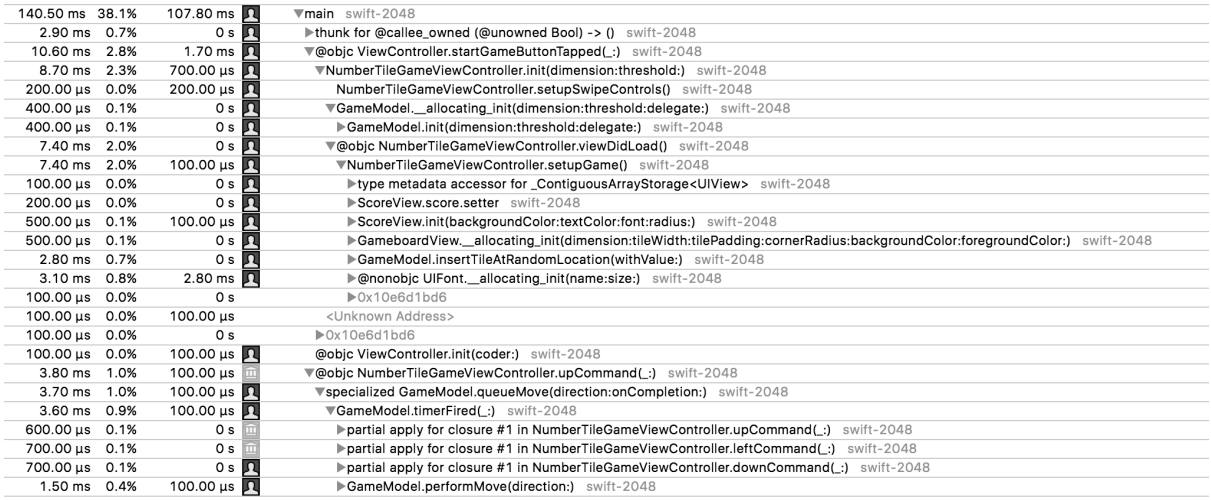


Figure 7.6: Execution trace after performing the Use Case

Runtime memory usage: 1.78MB

7.3.2.1 Experiment I.I: Main Agent Configuration I

Agent Integration Time: 16min 30sec

Agent overhead for invoked methods:
`ViewController.init(coder:)`: The constructor of `ViewController` initializes the view controller object and initializes all global variables contained in `ViewController`. As shown in Listing 7.2, we added an attribute holding the agent reference. Since the iOS agent is `nil` at the beginning, the method `getInstance() -> IITMAgent` will create a new agent instance. As shown in Figure 7.6, the duration for creating an agent instance is 16.30ms. Since the agent lives as a singleton, the initialization of the agent is performed only once. In the normal case when starting the application.

Listing 7.2 2048 ViewController Class

```
import UIKit
import InstrumentITMobileTracer

class ViewController: UIViewController {
   let agent = IITMAgent.getInstance()
   ...
}
```

`IITMAgent.trackInvocation(function:file:)`: This method starts tracing a method invocation. It comprehends instantiating a span object, which takes 0.8ms to 0.9ms, mapping the span object to the dictionary and establishing correlation between spans which takes 0.1ms to 0.2ms. In total, the execution times of this method are around **0.9ms to 1.1ms**.

`IITMAgent.closeInvocation(invocation:)`: This method ends tracing a method invocation. The execution time of this method is dependent on the closing invocation. If the closing invocation is a root span, the agent serializes the whole trace and tries to send the monitoring data to a back end. Therefor the execution time of this method is highly dependent on how many spans has to be serialized. Closing a non-root span is very performant. The high frequency profiler estimates the duration under **0.1ms**. In the other cases, for instance in case `NumberTileGameViewController.upCommand(_:)` was called, the duration for closing the span, serializing the trace and trying to send it was **25.20ms**. In that case 46 spans were serialized, because a root span was closed. We

noticed that the duration for serializing and preparing to send the trace is split in nearly equal parts.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is **0.3MB**. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 28.34MB, but only **2.37MB** were allocated persistently afterwards.

7.3.2.2 Experiment I.II: Main Agent Configuration II

Agent Integration Time: 1hour 10min 30sec

Integration notes: In the process of swizzling methods, we recognized that methods using full native Swift constructs such as tuples can not be swizzled with other methods. In addition to that, all hooked methods needed to be manually marked with the `@objc` attribute. For this reason, methods using tuples or Swift full native enumerations will not be recognized by the mobile iOS agent. This is a huge constraint for the developer in terms of implementing a project.

Configuration notes: Since the architecture of this agent configuration is the same as the main agent configuration I, the execution times of the agent invocations are not different. The difference between both configuration is that, MAC II is hooking the instrumented methods. For this reason we will focus on the overhead created in hooking the monitored methods in this experiment. We have also to consider that this agent configuration hooks instance methods each time when an object of a specific class is initialized. If there exists instrumented methods for a certain instance, the method `initialize()` is called. As shown in the execution trace in Figure 7.7, the profiling application recognized the invocation of `initialize()`, which handles the method swizzling, each time an object is created.

Agent overhead for invoked methods:
As described in the configuration notes, the execution times of the invoked agent methods are the same due to the fact that the agent architectures of both configurations are the same.

Overhead for hooking methods:
`ViewController.initialize()`: In this experiment, we swizzled the function

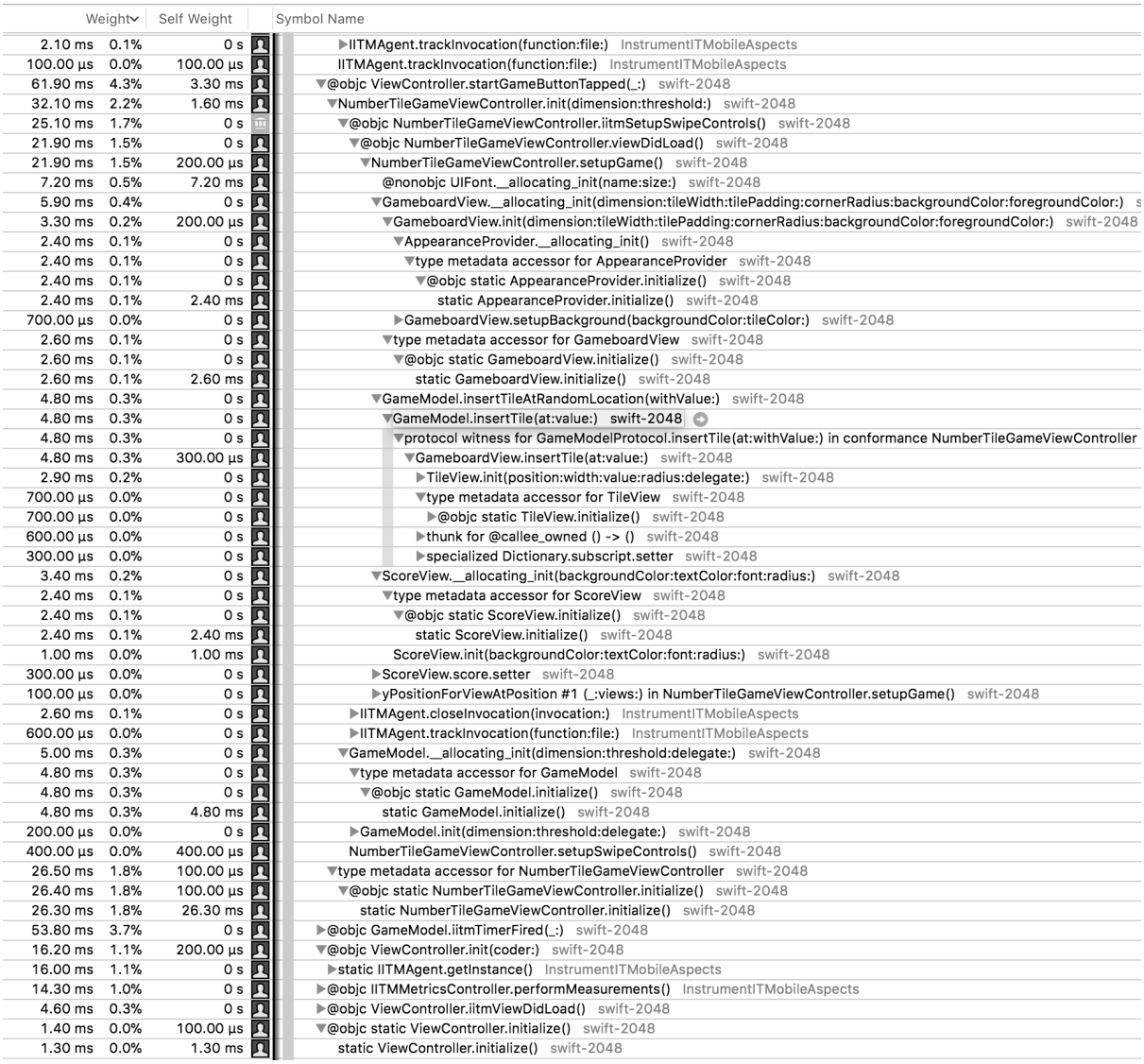


Figure 7.7: Partial Execution trace with Method Swizzling

`viewDidLoad()` which belongs to the `ViewController` class. Swizzling this function took **1.40ms**.

`NumberTileViewController.initialize()`: The next instance created after a `ViewController` instance, is a `NumberTileViewController` instance. For this class the agent configuration was able to hook eleven methods. The duration of the swizzling process was **26.40ms**

`GameBoardView.initialize()`: Afterwards, a `GameBoardView` instance was created. The agent configuration was able to hook three methods for this class. The duration of the swizzling process was **2.60ms**

`GameModel.initialize()`: This function swizzled three methods. The duration of this process was **4.8ms**.

`TileView.initialize()`: A game model objects handles to insert new tiles. Therefore new `TileView` objects are created. The agent swizzled only one function of this object in **0.7ms**.

`ScoreView.initialize()`: The agent configuration was able to hook two methods of a `ScoreView` instance. The duration of the swizzling process was 2.40ms.

`AppearenceProvider.initialize()`: The agent hooked three `AppearenceProvider` methods and took for this **2.40ms**.

As one may notice, there is no exact absolute duration for swizzling a method. In our experiment it took from 700 microseconds to 2.4ms to hook a method. In the following we list the relative duration for hooking one method with the given results:

`ViewController.initialize()` $1.40ms/1 = 1.40ms$
`NumberTileViewController.initialize()`: $26.40ms/11 = 2.40ms$
`GameBoardView.initialize()`: $2.60ms/3 = 0.90ms$
`GameModel.initialize()`: $4.80ms/3 = 1.60ms$
`TileView.initialize()`: $0.70ms/1 = 0.70ms$
`ScoreView.initialize()`: $2.40ms/2 = 1.2ms$
`AppearenceProvider.initialize()`: $2.40ms/3 = 0.8ms$

In the results discussion we will suppose that hooking a method takes **1.2ms** which is the median of the collected relative values. The relative average for hooking a method is 1.28ms, which is not far from the calculated median.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is **1.1MB**. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 14.38MB. **2.48MB** from the total were allocated persistently.

7.3.2.3 Experiment I.III: Main Agent Configuration III

Agent Integration Time: 1min 15sec

Integration notes: This agent configuration does not require from the user to edit much of the source code. The configuration has to be included in the project an the agent has to be started with one line of code.

Configuration notes: The sample rate of this agent configuration was set to 0.03 seconds. The agent configuration has not tried to dispatch any spans, because no span was recognized within the sampling rate.

Experiment notes: We manually measured the duration of the agent method `signalFuntion(sig:siginfo:p:))` in order to retrieve the exact execution time of the mentioned method, since the agent does not count how often the call stack is sampled. In addition to that, we noticed a memory ramp while performing the use case. This is also related to the high execution times of the signal function. Since the function is called often repeatedly, before the last invocation is finished, the call stack is raising. Therefore the memory usage is increasing. We have to also mention that in some cases the application crashed after some seconds for this reason.

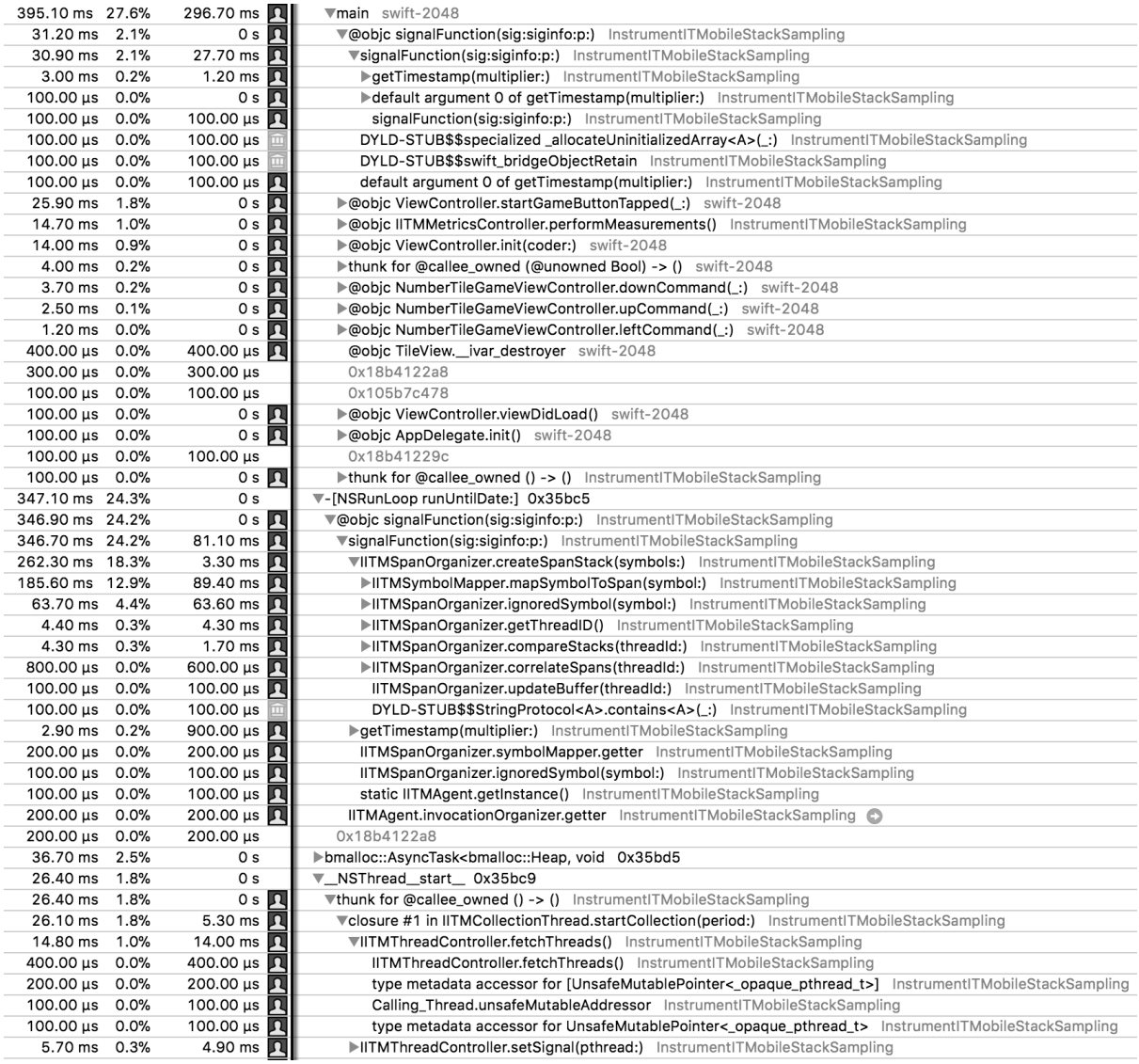


Figure 7.8: Execution trace with Call Stack Sampling

Agent overhead for invoked methods:
`ViewController.init(coder:)`: The constructor of `ViewController` initializes the view controller object and initializes all global variables contained in `ViewController`. We added an attribute holding the agent reference. Since the iOS agent is `nil` at the beginning, the method `getInstance() -> IITMAgent` will create a new agent instance. As shown in Figure 7.8, the duration for creating an agent instance of this configuration is 14.00ms. Since the agent lives as a singleton, the initialization of the agent is performed only once. In the normal case when starting the application.

`signalFuntion(sig:siginfo:p:))`: This function is called for every thread in each sampling iteration. The total duration of this function in the run loop was 346.70ms. This function includes creating the span stack which took 262.30ms in that instance. In addition to that, the symbols has to be filtered. This process took 63.70ms in this instance. Afterwards, the gathered symbols are mapped to spans which took 185.60ms. The stack comparison took 4.30ms and the span correlation 0.7ms. In a manual test we measured the execution time of this methods for one iteration. The results were from 4ms to 135ms

`IITMThreadController.startCollection()`: Fetching the active threads is included by this function. In order to measure each iteration, we added a measurement which calculates the duration of the thread collection iterations. The highest duration of a collection iteration 28.05ms.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is 2.3MB. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 22.04MB. 4.03MB from the total were allocated persistently.

7.3.2.4 Experiment I.IV: Main Agent Configuration IV

Agent Integration Time: 6min 30sec

Integration notes: This agent configuration requires a huge knowledge of the written source code. The framework user has to know which functions are starting a certain use case and which functions are ending a certain use case. For this experiment we spent around 6 minutes to instrument five root use cases needed for our experiment.

Agent overhead for invoked methods:
`ViewController.init(coder:)`: The constructor of `ViewController` initializes the view controller object and initializes all global variables contained in `ViewController`. We added an attribute holding the agent reference. Since the iOS agent is `nil` at the beginning, the method `getInstance() -> IITMAgent` will create a new agent instance. As shown in Figure 7.6, the duration for creating an agent instance is **16.10ms**.

`Agent.startRootUsecase(name:filename:line:funcname:))`: This function starts tracing a root use case. As mentioned in the integration notes, we defined five root use cases within the application. The duration of buffering the first use case was 20ms, as shown in Figure 7.9 in line five. The reason for this, is the instantiation of the special data

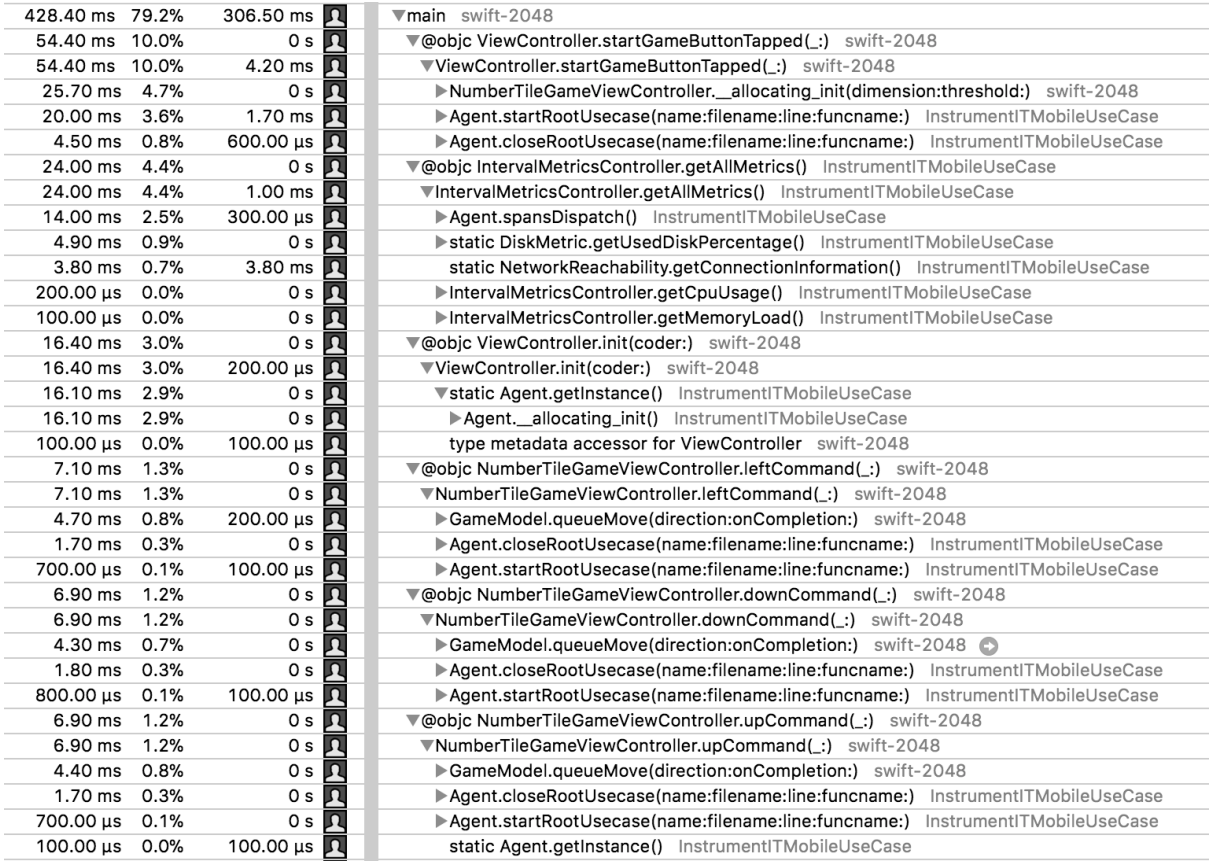


Figure 7.9: Execution trace with Use Case Mapping

structures to buffer the use cases. In addition to that, the timer for retrieving hardware resource workloads is initialized and started. In other cases, when the organization structure is initialized, start tracing a root use case takes 0.7ms to 0.8ms. For the further results discussion, we will add the creation time for the organization structure to the agent initialization time.

`Agent.closeRootUsecase(name:filename:line:funcname:))`: This function ends tracing a root use case. Closing a root use case has as a consequence that all use cases of the same trace are being serialized. Since we only defined root use cases, closing each use case will raise the serialization process. Serializing a use case takes **1.80ms** with this configuration. Only closing a use case takes **1.70ms**.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is **2.3MB**. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the

experimental use case the instrumented application allocated in total 5.03MB. **1.22MB** from the total were allocated persistently.

7.3.3 Experiment II: Hybrid Application

Used application: HybridApplication (https://github.com/sassanmo/HybridApplication-iOS-Example)

Application description: Since we were not able to find any iOS based hybrid application, we decided to implement one ourself. The application represents a small web browser. The end-user is able to type in a link and load the web page by pressing on the load button. In addition to that, we provided two buttons in order to navigate forwards and backwards through the visited pages and one to reload the page. Figure 7.10 shows a screen shot of the application.

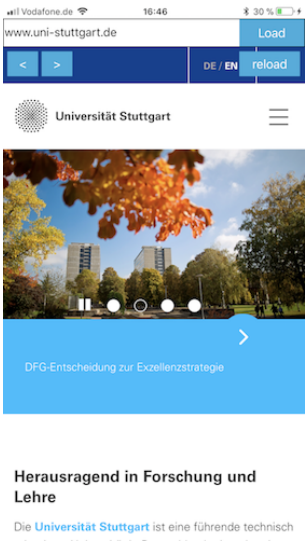


Figure 7.10: Screen shot of the implemented Hybrid Application

Use case: The profiling use case is to start the application, to load a web page by pressing on the *load* button and to reload the web page.

7.3.3.1 Experiment II.I: Main Agent Configuration I

Agent Integration Time: 4min 30sec

Integration notes: In order to trace the remote calls performed from the web view, the web view has to be registered once by the agent.

Configuration notes: As shown in Figure 7.11 all performed web view methods were traced as expected. The performed remote calls were traced.

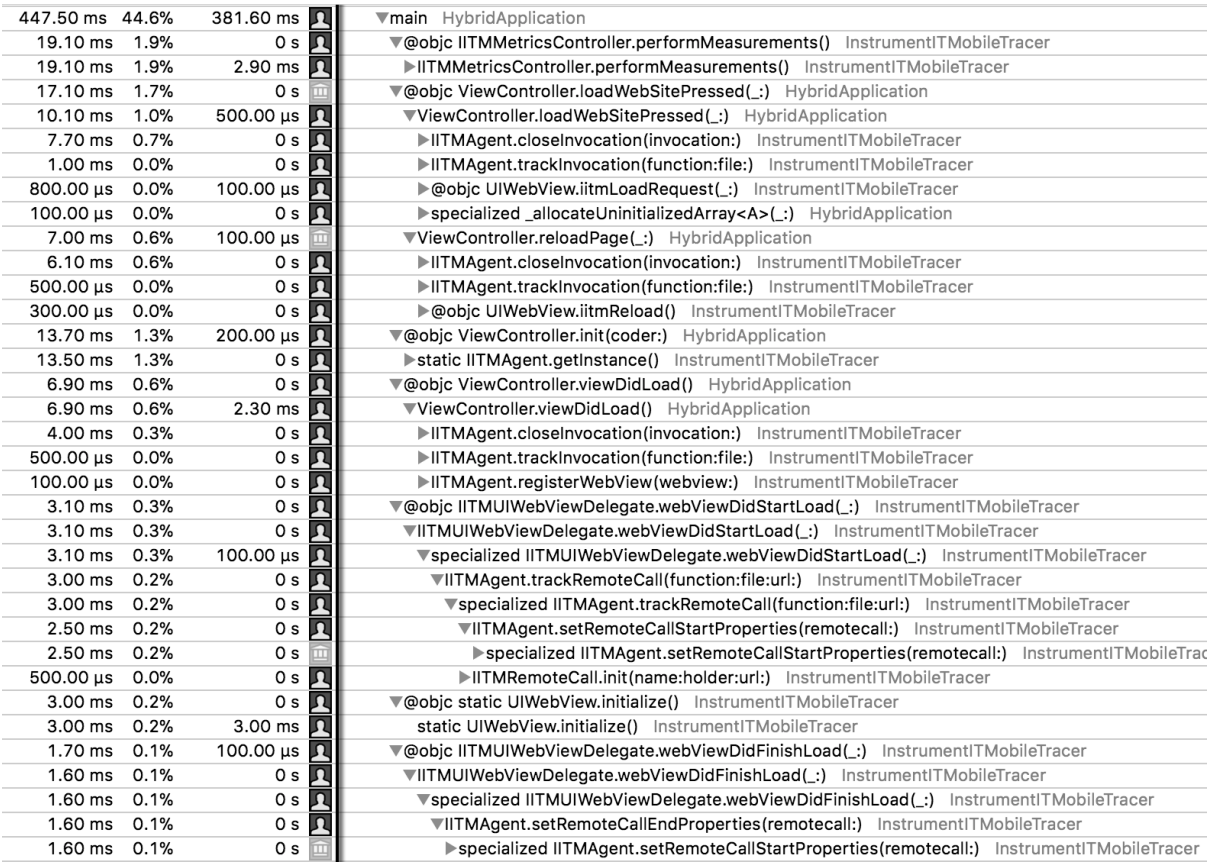


Figure 7.11: Execution trace with Tracing

Agent overhead for invoked methods:
`ViewController.init(coder:)`: The constructor of `ViewController` initializes the view controller object and initializes all global variables contained in `ViewController`.We added an attribute holding the agent reference. Since the iOS agent is `nil` at the beginning, the method `getInstance() -> IITMAgent` will create a new agent instance. As shown in Figure 7.6, the duration for creating an agent instance is **13.50ms**. Since the agent lives as a singleton, the initialization of the agent is performed only once. In the normal case when starting the application.

`IITMAgent.trackInvocation(function:file:)`: This method starts tracing a method invocation. The execution times of this method for this experiment were from around **0.5ms** to **1.0ms**.

`IITMAgent.closeInvocation(invocation:)`: This method ends tracing a method invocation. The execution time of this method is dependent on the closing invocation. For this experiment the execution times were low, due to the reduced amount of traced spans. The execution times of this method for this experiment were from around **4ms** to **7.70ms**.

`UIWebView.initialize()`: Since for this experiment the mobile agent had to trace a system library, `UIWebView`, the agent hooked the web view methods on instantiation. `UIWebView.initialize()` handled the method swizzling and the duration was **3.00ms**.

`IITMAgent.registerWebView(webview:)`: In order to trace the performed remote calls from the web view, the web view has to be registered. In that case, a delegate is created which hooks the performed remote calls. The registration took **0.10ms**.

`IITMAgent.trackRemoteCall(function:file:url:)`: This method starts tracing a remote call. It comprehends instantiating a remote call object, which takes 0.5ms and setting the start remote call properties, which took 2.50ms. The total execution time was **3ms**.

`IITMAgent.closeRemoteCall(remotecall:response:error:)`: This method ends tracing a remote call. The execution time was **1.6ms**.

Overall memory usage: This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 16.75MB, but only **3.61MB** were allocated persistently afterwards.

7.3.3.2 Experiment II.II: Main Agent Configuration II

Agent Integration Time: 9min 10sec

Configuration notes: For this experiment we will follow the same approach as experiment I.II. We will focus on the overhead created in hooking the monitored methods in this experiment. We have also to consider that this agent configuration hooks instance methods each time when an object of a specific class is initialized. As shown in the execution trace in Figure 7.12, the profiling application recognized the invocation of `initialize()`, which handles the method swizzling, each time an object is created.

Figure 7.12: Execution trace with Method Swizzling

Agent overhead for invoked methods:
As described in the configuration notes, the execution times of the invoked agent methods are the same due to the fact that the agent architectures of both configurations are the same.

Overhead for hooking methods:
`ViewController.initialize()`: In this experiment, we swizzled the functions `viewDidLoad()`, `loadWebSitePressed(_:)`, `reloadPage(_:)`, `goForwardPressed(_:)`, `goBackwardsPressed(_:` which belong to the `ViewController` class. Swizzling the mentioned functions took 2.60ms.

`UIWebView.initialize()`: Hooking the web view methods on instantiation took 1.50ms.

As one may notice, there is no exact absolute duration for swizzling a method also in this experiment. In the following we list the relative duration for hooking one method

with the given results:

`ViewController.initialize()` $2.60ms/5 = 0.52ms$
`UIWebView.initialize()`: $1.50ms/6 = 0.25ms$

In the results discussion we will suppose that hooking a method takes **0.39ms** which is the average of the collected relative values. We selected the average this time, due to a small measurement set.

Overall memory usage: This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 16.53MB. **3.60MB** from the total were allocated persistently.

Experiment II.III: Main Agent Configuration III

Agent Integration Time: 1min 15sec

Integration notes: This agent configuration does not require from the user to edit much of the source code as in experiment I.III. The configuration has to be included in the project an the agent has to be started with one line of code.

Configuration notes: The sample rate of this agent configuration was set to 0.06 seconds, due to application crashes with lower sampling rates. The agent configuration unexpectedly did not recognize any performed remote calls.

Experiment notes: We manually measured the duration of the agent method `signalFuntion(sig:siginfo:p:))` in order to retrieve the exact execution time of the mentioned method, since the agent does not count how often the call stack is sampled. In addition to that, we noticed a memory ramp while performing the use case. This is also related to the high execution times of the signal function. Since the function is called often repeatedly, before the last invocation is finished, the call stack is raising. Therefore the memory usage is increasing. We have to also mention that in some cases the application crashed after some seconds for this reason.

Agent overhead for invoked methods:
`ViewController.viewDidLoad()`: The duration for creating an agent instance of this configuration is **12.30ms**. Since the agent lives as a singleton, the initialization of the agent is performed only once. In the normal case when starting the application.

`signalFuntion(sig:siginfo:p:))`: This function is called for every thread in each sampling iteration. This function includes creating the span stacks, correlating the spans, comparing the new span stack with the old ones and to adjust the execution times of the spans. The execution time of this function varied from **4.60ms** to **222.70ms**.

`IITMThreadController.startCollection()`: Fetching the active threads is included by this function. In order to measure each iteration, we added a measurement which calculates the duration of the thread collection iterations. The highest duration of a collection iteration 258.36ms.

Overall memory usage: This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 16.30MB. **3.82MB** from the total were allocated persistently.

Experiment II.IV: Main Agent Configuration IV

Agent Integration Time: 35min 30sec

Integration notes: This agent configuration requires a huge knowledge of the written source code. The framework user has to know which functions are starting a certain use case and which functions are ending a certain use case. For this experiment we spent around over 30 minutes to instrument two root use cases and one remote call needed for our experiment. In addition to that this agent configuration requires a huge refactoring of the source code. In order to fetch the remote call and the real duration for the use case, the developer is forced to implement a web view delegate.

Agent configuration notes: The agent successfully, traced the first use case and the remote call as a sub span of the root use case. The second one could not be traced by the agent.

Agent overhead for invoked methods:
`ViewController.init(coder:)`: The constructor of `ViewController` initializes the view controller object and initializes all global variables contained in `ViewController`. We

added an attribute holding the agent reference. The duration for creating an agent instance is **16.00ms**.

`Agent.startRootUsecase(name:filename:line:funcname:))`: This function starts tracing a root use case. As mentioned in the integration notes, we defined two root use cases within the application. The duration for buffering the first use case was **1.7ms**, as shown in Figure 7.9. The second one **0.5ms**.

`Agent.closeRootUsecase(name:filename:line:funcname:))`: This function ends tracing a root use case. Closing a root use case has as a consequence that all use cases of the same trace are being serialized. Since we only defined root use cases, closing each use case will raise the serialization process. Serializing a use case takes 1.10ms with this configuration. In total closing a use case, which also includes to set the ending properties takes **1.70ms**.

`Agent.startRemoteCall(name:root:parent:url:httpMethod:request:)`: This function starts tracing a remote call. The execution time of this method was **2.0ms**

`Agent.closeRemoteCall(name:root:responseCode:timeout:)`: This function ends tracing a remote call and serializes the remote call. The execution time of this method was **1.2ms**.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is **2.3MB**. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 5.03MB. **1.22MB** from the total were allocated persistently.

7.3.4 Experiment III: Native Client of a Distributed System

Used application: MobileClient (https://github.com/sassanmo/MobileClient-Demo)

Application description: Due to time reasons we decided to implement a mobile client, which requests data from a foreign back end, for ourself. The application implements the standard functionality of requesting content from a back end. The end-user is able to load an image by pressing on the load content button. Figure 7.13 shows a screen shot of the application.

Figure 7.13: Screen shot of the implemented Mobile Client

Use case: The profiling use case is to start the application and to load an image by pressing on the *Load Content* button.

Expected execution trace: As shown in Figure 7.14 the overall absolute operational time to perform the experimental use case was 271.10ms on the main thread and 4.10ms in a sub thread. In the main thread The sub thread handled the performed remote call.

271.10 ms	69.0%	0 s	▼Main Thread 0xbaa2d
271.10 ms	69.0%	262.90 ms	▼main MobileClient
7.60 ms	1.9%	0 s	▼@objc ViewController.loadContentButtonPressed(_:) MobileClient
7.20 ms	1.8%	0 s	▼ViewController.loadContentButtonPressed(_:) MobileClient
7.20 ms	1.8%	7.20 ms	ViewController.loadContent() MobileClient
400.00 µs	0.1%	400.00 µs	_bridgeAnyObjectToAny(_:) MobileClient
200.00 µs	0.0%	200.00 µs	0x102cd4468
100.00 µs	0.0%	0 s	▶thunk for @callee_owned () -> () MobileClient
100.00 µs	0.0%	100.00 µs	0x192020aa8
100.00 µs	0.0%	100.00 µs	@objc ViewController.init(coder:) MobileClient
100.00 µs	0.0%	100.00 µs	0x102cd4004
4.10 ms	1.0%	0 s	▼_dispatch_workloop_worker_thread$VARIANT$armv81 0xbaa84
4.00 ms	1.0%	200.00 µs	▼thunk for @callee_owned (@owned Data?, @owned URLResponse?, @owned Error?) -> () MobileClient
3.80 ms	0.9%	0 s	▼partial apply for closure #1 in ViewController.loadContent() MobileClient
3.80 ms	0.9%	0 s	▼closure #1 in ViewController.loadContent() MobileClient
3.80 ms	0.9%	300.00 µs	▼specialized closure #1 in ViewController.loadContent() MobileClient
3.50 ms	0.8%	3.50 ms	@nonobjc UIImage.init(data:) MobileClient

Figure 7.14: Execution trace after performing the Use Case

Runtime memory usage: 1.73MB

7.3.4.1 Experiment III.I: Main Agent Configuration I

Configuration notes: The performed remote call was traced.

Figure 7.15: Execution trace with Tracing

Agent overhead for invoked methods:
`ViewController.init(coder:)`: The constructor of `ViewController` initializes the view controller object and initializes all global variables contained in `ViewController`.We added an attribute holding the agent reference. Since the iOS agent is `nil` at the beginning, the method `getInstance() -> IITMAgent` will create a new agent instance. As shown in Figure 7.15, the duration for creating an agent instance is **13.10ms**. Since the agent lives as a singleton, the initialization of the agent is performed only once. In the normal case when starting the application.

`IITMAgent.trackInvocation(function:file:)`: This method starts tracing a method invocation. The execution times of this method for this experiment were from around **0.5ms** to **1.0ms**.

`IITMAgent.closeInvocation(invocation:)`: This method ends tracing a method invocation. The execution time of this method is dependent on the closing invocation. For this experiment the execution times were low, due to the reduced amount of traced spans. The execution times of this method for this experiment were from around **0.1ms** to **6.6ms**.

`IITMAgent.trackRemoteCall(function:file:url:)`: The duration for tracking this specific remote call was **6.8ms**. The most time was spent retrieving the SSID (4.2ms) and the network connection type (1.5ms). The duration for crating a remote call instance was 0.5ms.

`IITMAgent.closeRemoteCall(remotecall:response:error:)`: This method ends tracing a remote call. The execution time was **6.1ms**. Closing a remote call also includes the serialization of the remote call and the measurement dispatch.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is **0.2MB**. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 6MB, but only **1.47MB** were allocated persistently afterwards.

7.3.4.2 Experiment III.II: Main Agent Configuration II

Agent Integration Time: 5min 15sec

Configuration notes: For this experiment we will follow the same approach as experiment I.II and II.II. We will focus on the overhead created in hooking the monitored methods in this experiment. We have also to consider that this agent configuration hooks instance methods each time when an object of a specific class is initialized. As shown in the execution trace in Figure 7.12, the profiling application recognized the invocation of `initialize()`, which handles the method swizzling, each time an object is created.

Figure 7.16: Execution trace with Method Swizzling

Agent overhead for invoked methods:
As described in the configuration notes, the execution times of the invoked agent methods are the same due to the fact that the agent architectures of both configurations are the same.

Overhead for hooking methods:
`ViewController.initialize()`: In this experiment, we swizzled the functions `viewDidLoad()`, `loadContent()` and `loadContentButtonPressed(_:)` which belong to the `ViewController` class. Swizzling the mentioned functions took 2.0ms. We also swizzled `URLSession` methods.

`ViewController.initialize()` $1.8ms/3 = 0.6ms$
`URLSession.initialize()` $1.1ms/2 = 0.55ms$

In the results discussion we will suppose that hooking a method takes **0.58ms** which is the calculated average value of swizzling a method in this experiment.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is 0.5MB. This probe was measured after integrating, building and

running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 5.97MB. **1.45MB** from the total were allocated persistently.

7.3.4.3 Experiment III.III: Main Agent Configuration III

Agent Integration Time: 1min 15sec

Integration notes: This agent configuration does not require from the user to edit much of the source code as in experiment I.III and II.III. The configuration has to be included in the project an the agent has to be started with one line of code.

Configuration notes: The sample rate of this agent configuration was set to 0.06 seconds, due to application crashes with lower sampling rates. The agent configuration has only tried to dispatch the measured remote call span as shown in Listing 7.3. Other invocation measurements were not caught by the monitoring agent.

Listing 7.3 Traced Remote call Span

```
...
{
   "operationName" : "iitmDataTask(request:completionHandler:)",
   "duration" : 99998,
   "tags" : {
      "http.request.ssid" : "eduroam",
      "http.request.networkConnection" : "WLAN",
      "http.response.networkConnection" : "WLAN",
      "http.response.ssid" : "eduroam",
      "span.kind" : "client",
      "http.url" : "https:\/\/www.the...\/content.jpg",
      "http.response.networkProvider" : "",
      "ext.propagation.type" : "HTTP",
      "http.request.networkProvider" : ""
   },
   "spanContext" : { ...
   }
}
...
```

Experiment notes: We manually measured the duration of the agent method `signalFuntion(sig:siginfo:p:))` in order to retrieve the exact execution time of the mentioned method, since the agent does not count how often the call stack is sampled. In addition to that, we noticed a memory ramp while performing the use case. This is also related to the high execution times of the signal function. Since the function is called often repeatedly, before the last invocation is finished, the call stack is raising. Therefore the memory usage is increasing. We have to also mention that in some cases the application crashed after some seconds for this reason.

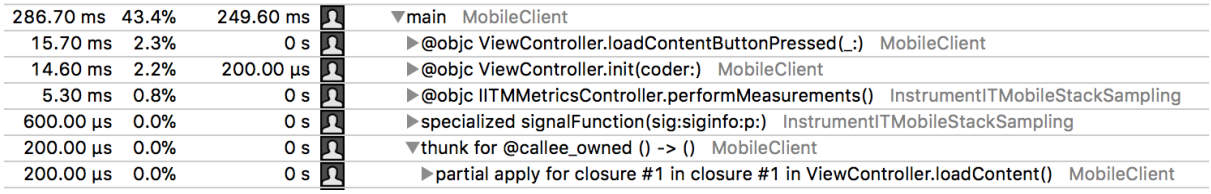


Figure 7.17: Execution trace with Call Stack Sampling (Main Thread)

Agent overhead for invoked methods:
`IITMAgent.getInstance()`: The duration for creating an agent instance of this configuration is 13.60ms. Since the agent lives as a singleton, the initialization of the agent is performed only once. In the normal case when starting the application.

`signalFuntion(sig:siginfo:p:))`: This function is called for every thread in each sampling iteration. This function includes creating the span stacks, correlating the spans, comparing the new span stack with the old ones and to adjust the execution times of the spans. The execution time of this function varied from 2.93ms to 21.26ms.

`IITMThreadController.startCollection()`: Fetching the active threads is included by this function. In order to measure each iteration, we added a measurement which calculates the duration of the thread collection iterations. The highest duration of a collection iteration 33.94ms.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is 2.3MB. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total 5.03MB. 1.22MB from the total were allocated persistently.

7.3.4.4 Experiment III.IV: Main Agent Configuration IV

Agent Integration Time: 15min 15sec

Integration notes: As mentioned in the related experiments before, this configuration requires a huge knowledge of the written source code. The framework user has to know which functions are starting a certain use case and which functions are ending a certain use case. In addition to that this agent configuration requires a huge refactoring of the source code.. Since the selected application does not provide a lot of functions, the number of traceable use cases is also low. Therefore integrating this agent configuration for this experiment required less time. We defined two root use cases and added a nested remote call to a use case.

Agent configuration notes: The agent successfully, traced the root use case when the view appeared, the root use case when pressing the *Load Content* button and the nested remote call as a sub span of the root use case. The performed agent invocation are included in the execution trace shown in Figure 7.18.

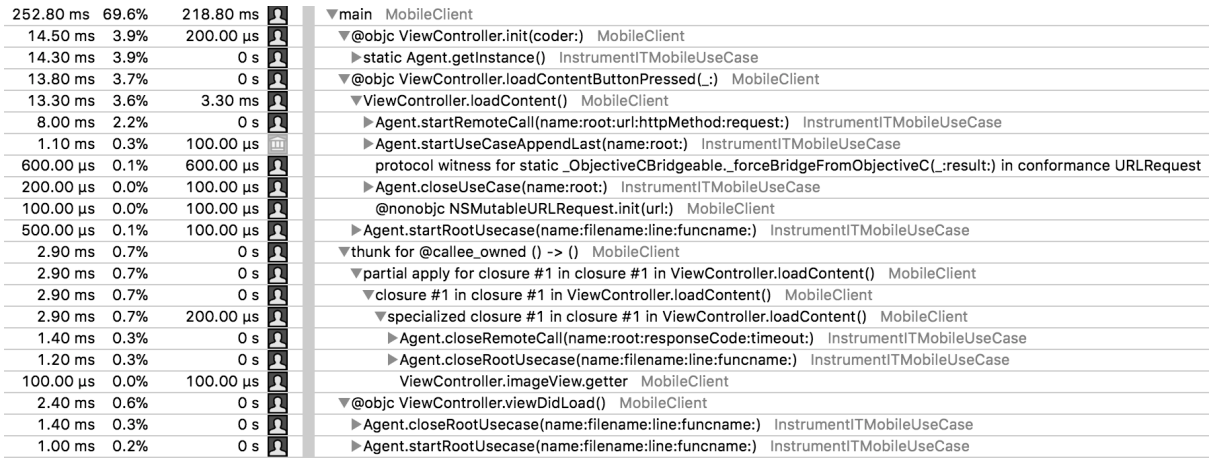


Figure 7.18: Execution trace with Use Case Mapping

Agent overhead for invoked methods:
`ViewController.init(coder:)`: The constructor of `ViewController` initializes the view controller object and initializes all global variables contained in `ViewController`. We added an attribute holding the agent reference. The duration for creating an agent instance is **14.30ms**.

`Agent.startRootUsecase(name:filename:line:funcname:))`: This function starts tracing a root use case. As mentioned in the integration notes, we defined two root use cases within the application. The duration for buffering the first use case was **1.4ms**, as shown in Figure 7.18. The second one **0.5ms**.

`Agent.closeRootUsecase(name:filename:line:funcname:))`: This function ends tracing a root use case. Closing a root use case has as a consequence that all use cases of

the same trace are being serialized. Since we only defined root use cases, closing each use case will raise the serialization process. The first use case was closed in **1.40ms**. The duration for closing the second one was **1.20ms** even though more spans were serialized.

`Agent.startRemoteCall(name:root:parent:url:httpMethod:request:)`: This function starts tracing a remote call. The execution time of this method was **8.0ms**

`Agent.closeRemoteCall(name:root:responseCode:timeout:)`: This function ends tracing a remote call and serializes the remote call. The execution time of this method was **1.4ms**.

Overall memory usage: The measured memory overhead of the integrated and compiled agent module is **0.2MB**. This probe was measured after integrating, building and running the agent within the application without allocating spans. While performing the experimental use case the instrumented application allocated in total **5.9MB**. **1.3MB** from the total were allocated persistently.

7.4 Discussion of the Results

We summarized the results of all experiments in Table 7.1. After the summary of the results, we will discuss the results for each criterion.

Agent Configuration	Integration time	Agent Initialization time	Hooking Overhead / Invocation	Tracking Method/Use case (Remote call)	Tracking Stack Trace	Close Invocation/Use case (Remote call)	Span Serialization / Span
MAC I	16min	16ms	-	1ms -	-	1ms -	0.6ms
MAC II	1h 10min	16ms	1.2ms	1ms -	-	1ms -	0.6ms
MAC III	1min	14ms	- -	-	135ms	- -	-
MAC IV	6min	16.1ms	-	0.7ms -	-	1.4ms -	0.4ms
MAC I	4min	13.5ms	0.5ms	1ms 3ms	-	1ms 1.6ms	0.6ms
MAC II	9min	13.5ms	0.39ms	1ms 3ms	-	1ms 1.6ms	0.6ms
MAC III	1min	12.3ms	-	- -	222.7ms	- -	-
MAC IV	35min	16ms	-	1.7ms 2ms	-	1.4ms 1.2ms	1.4ms
MAC I	2.5min	13.1ms	0.5ms	1ms 6ms	-	1ms 2.4ms	2ms
MAC II	5min	13.1ms	0.58ms	1ms 6ms	-	1ms 2.4ms	2ms
MAC III	1min	13.6ms	6.1ms	- 6ms	55.2ms	- 3.8ms	-
MAC IV	15min	14.3ms	-	1.4ms 8ms	-	1.4ms 1.4ms	1.4ms

Table 7.1: Execution times results summary of all Experiments

The first criterion of the practical evaluation was the agent integration time. Integrating an agent consists of including the agent framework to the application project, linking the binaries and activating the agent by adding instrumentation source code. Some strategies require more code refactoring other ones less. A diagram of the integration times is illustrated in Figure 7.19.

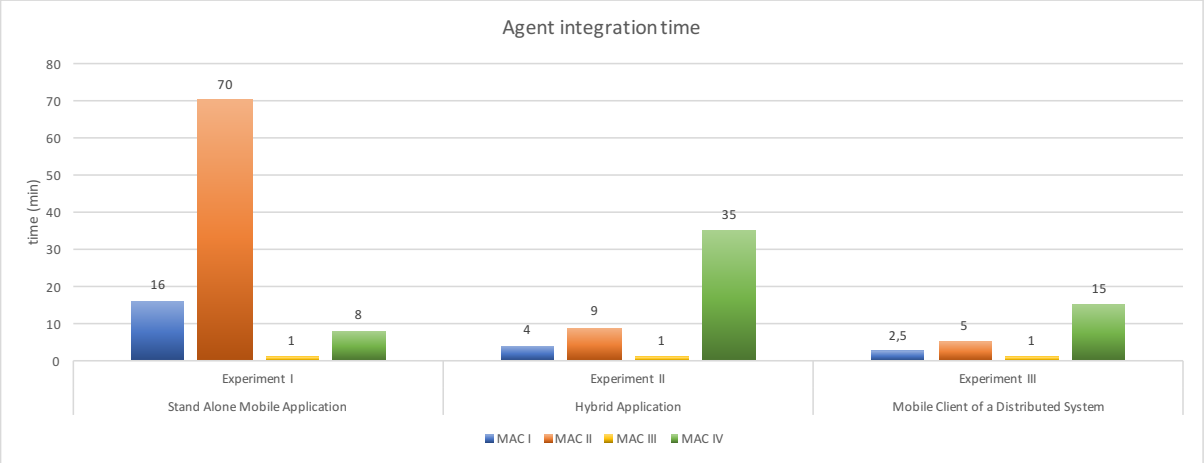


Figure 7.19: Integration Times

The given bar chart is divided in three parts. Each part is related to an executed experiment. At first sight it seems that integrating MAC II in stand alone mobile application takes long time. For this case we have to mention the sizes of the application projects. The application selected for experiment I had the most lines of code, more precisely the selected application had more methods to instrument than other applications. The application used for the second experiment the second most, and the third one the less. Considering the mentioned facts, one can observe a pattern for method invocation tracing-based agent configurations (MAC I & MAC II). The more methods are implemented in the given application, the more instrumentation code has to be added and therefore more time is required. Integrating MAC II took more time over MAC I because more lines of code has to be added. An other point explaining the long integration of MAC II is that the developer firstly has to check whether the method can be set as an `@objc` method. In cases the developer is not allowed to, even a refactoring of the method has to be performed. In that context, refactoring means replacing native *Swift* constructs with bridgeable *Objective-C* objects. MAC III was integrated for all experiments in constant time. This result was expected since only one line of code has to be added for each environment. Tracing remote call with MAC IV was more problematic than with other ones. The reason is that `URLRequest` objects has to be manually converted to `NSMutableURLRequest` and the flow of the remote call has to be caught by the developer

manually. As a result, the developer spends a lot of time in recognizing the entry end exit points of the remote calls.

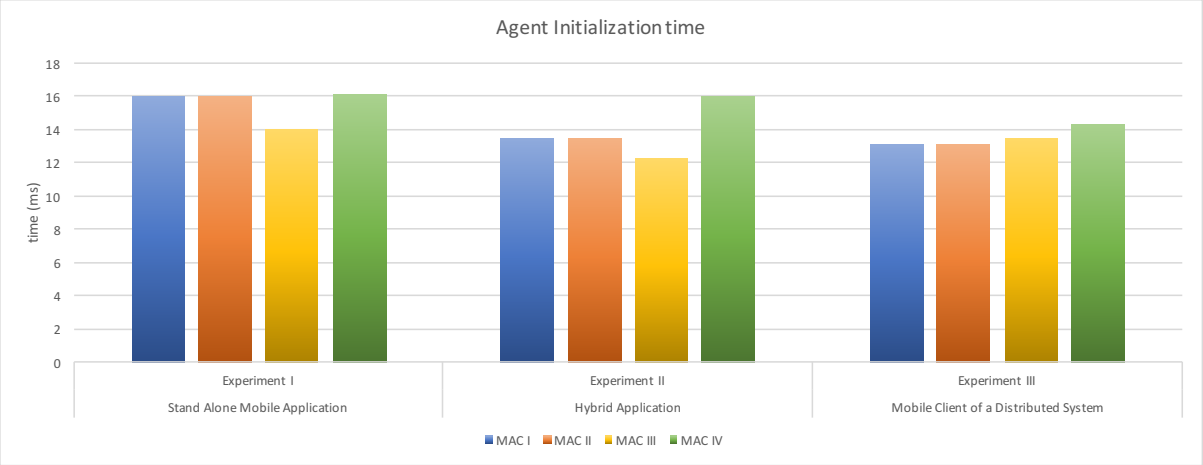


Figure 7.20: Agent Initialization Times

In the normal case, the monitoring agent is initialized only once within the application life cycle. Therefore, this criterion is not weighted as much as the other ones. The bar chart displayed in Figure 7.20 shows the initialization times in milliseconds of each agent configuration dependent on the monitored application type. As one can notice the execution times of creating an agent instance do not fluctuate much. This result was expected due to the similarity of the agent architectures. Since initializing the agent does not require application-based data we did not expect to notice a pattern dependent on the experiments in the diagram.

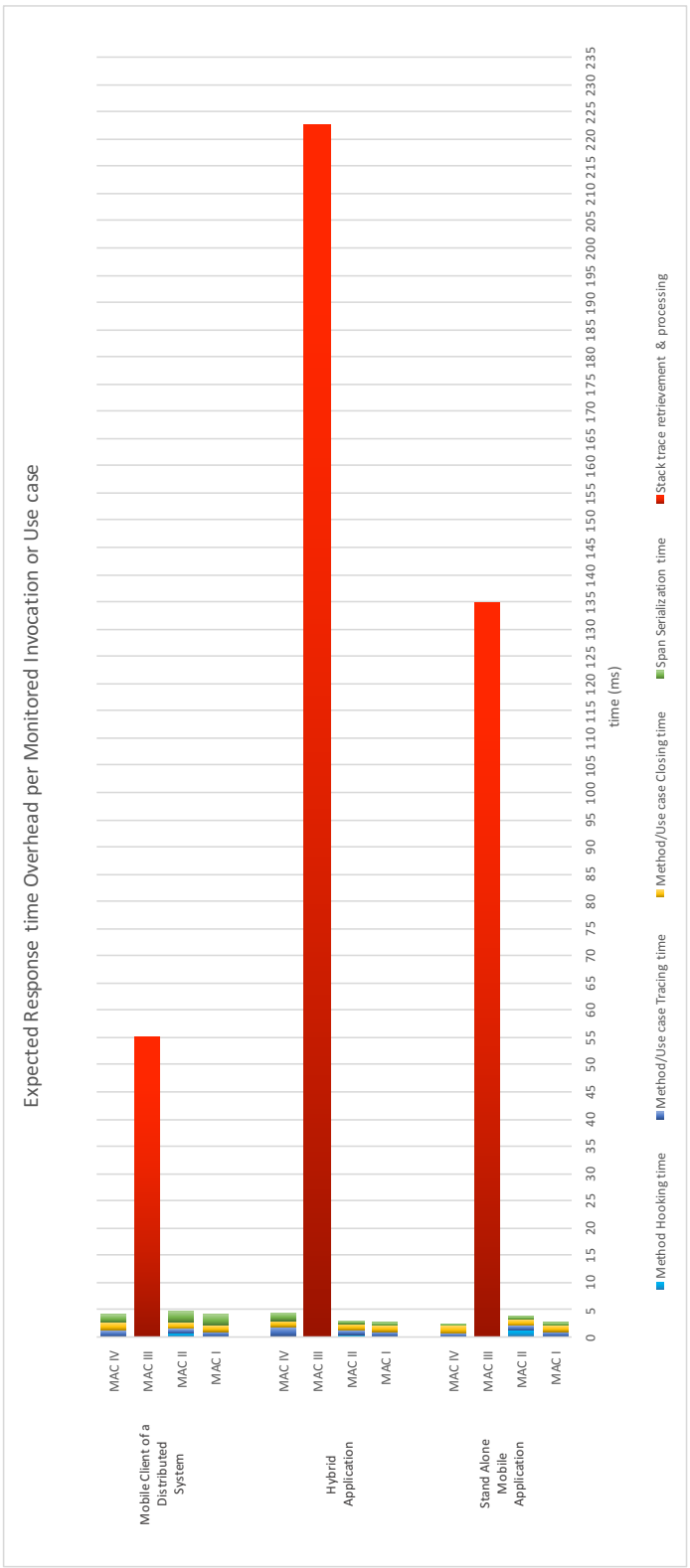


Figure 7.21: Response time Overhead

The stacked bar chart of Figure 7.21 shows the measured overhead in response times when monitoring method invocations or use cases. As mentioned in Figure 7.1 the overhead is made of occasionally hooking a method, tracking an invocation, correlating and processing the measured span and serializing it. In Figure 7.2 we explained how the overhead for stack sampling-based approaches differ from tracing-based ones. Since our sampling-based agent injects instrumentation code on top of the call stack periodically, the response time increases at each iteration by the execution time of the instrumentation code. As one may notice the overhead produced by MAC III is the biggest in relation to the other ones. The peek monitoring overhead (222,7ms) was measured when monitoring an hybrid application. The other monitoring configurations were more performant. In two out of the three experiments MAC I increased the response times less. MAC IV was slightly more performant (0.1ms faster) in the first experiment. This can be explained that only root use cases were monitored. The subsequent experiments show that as soon as the developer wants to monitor nested use cases, the processing increases and becomes higher than the processing time measured for MAC I or MAC II. The second agent configuration is less performant than MAC I due to the method hooking process. The average time of hooking a method varied from 0.39ms to 1.20ms. We did not notice any dependencies between agent configuration and application type. Overall MAC I performed better than other configurations.

In order to conclude the discussion of the results we sum up the gained results. As one may notice, the results are merely the same for each application class. As a result we are able to say that an agent configuration is not more suitable than an other one dependent on the application type. The different application types do not influence the performance of the agent. This can be explained with the base architecture of the different application types and the functionality of the agents. The difference between the different application types lays in the usage of different system libraries. Since all the agent configurations, apart from MAC III (only for remote calls), are hooking system library-based functions in order to monitor them, there is no difference in terms of performance. We can only suppose that while monitoring applications using system libraries, the number of measured spans is higher. In that case the performance overhead increases proportionally for all agent configurations.

The main research question (**RQ1**) for this evaluation was, whether there exist an agent configuration with certain strategies perform better on certain application types than other ones. The answer to this question, based on the gathered, presented and explained results, is that in general, and with the actual state of the implementations, the **tracing-based approach** with an implemented **dictionary organization** performs better than other agent configurations, **independent on which application they are running on**. In order to respond to the second research question (**RQ2**) we have to calculate the additional spent time to monitor an invocation. Based on the gained results, tracing, closing and serializing a span in total takes from **2.6ms** to **4.0ms**.

The integration time of the method invocation tracing-based agent can be decreased by using the implemented tool *InstrumentIT*. The application developer has to add and link the binaries of the agent, start InstrumentIT, choose the project that has to be monitored and to click on the *instrument all* button. After the mentioned steps the application is ready to be monitored. We added an experiment for this pipeline. The original application project, which was used for the first experiment, has been copied and instrumented again by utilizing InstrumentIT. In this experiment the agent was integrated in **45 seconds** instead of 15 minutes.

7.5 Threats to Validity

Threats to Conclusion Validity: When repeating the experiments a certain amount of time, we recognized that the profiling application returns different measurements. In order to bridge this problem we performed the tests several times until the results became stable. Even though we can not guarantee that the results are fully correct.

Threats to Internal Validity: The reliability of the agent configuration was not tested, therefore we cannot guarantee that the measurements and processes performed by the agent configurations are true. We can only assume that the processes of the agent configurations are well performed. Apart from that, we assumed in the theoretical evaluation that some strategies are in general less performant than other ones and excluded them for the experimental evaluation.

Threats to Construct Validity: For the mobile hybrid application type experiment, we implemented an own application in order to test this application class. We did not test the agent configurations with other open-source hybrid applications, therefore we cannot guarantee for the results retrieved of this experiment. We also implemented a mobile client in order to test this application class.

Threats to External Validity: The evaluation results might not be reliable because the tested agent configurations were implemented by ourselves and might be optimized. For two main experiments we also used self implemented applications.

Chapter 8

Conclusion

This chapter recapitulates what we did in the course of this thesis. Section 8.1 summarizes the work of the thesis. Section 8.2 argues with the defined goals of Section 1.2. In the last section of this chapter, Section 8.2, we list recommended work for the future that could not be done in the course of this thesis.

8.1 Summary

In the context of this thesis we researched and documented various mobile application classes. Considering the presented classes, we listed the software requirements for implementing a monitoring agent for mobile devices. In order to implement various agent configurations we researched the state of the art and other possibilities to monitor monitoring applications. For the three important working phases of the agent such as collecting, managing and dispatching monitoring data we discussed and presented various applicable strategies. As a result we implemented four main agent configurations. The first one is tracing-based on method level. The second configuration traces methods by exchanging the original method implementation with the instrumented one. With this approach the application developer is able to separate the cross cutting concerns from the core concerns of the application. The third agent configuration we implemented is call stack sampling-based. The last one is able to trace predefined use cases of the application. Some attributes of each of the implementations can be modified at runtime. The first and second agent configuration are able to organize trace data in a dictionary as well as in multiple stacks. Apart from that, the dispatch option and the dispatch constraint can be changed in each implementation. At the end of this thesis we evaluated the different monitoring strategies in two phases. The first phase consisted of comparing various strategies of the same working phase theoretically. In the second phase of the evaluation we performed several practical experiments where we profiled applications of different

classes with integrated monitoring agents in order to compare the performance in terms of execution times. The results indicate, with the actual state of the implementations, that the tracing-based approach with an implemented dictionary organization holding span objects performs better than other agent configurations, independent on which application they are running on.

8.2 Retrospective

The goals of this thesis were defined in Section 1.2. This section argues whether we reached the goals and how.

The fist goal of this thesis was to define a set of application classes. We researched topic related literature without appropriate results. Additionally, we searched for similar topics on the web. The only application classification we found were the categories defined by Apple. In Section 3.1 we explained that in the context of this thesis it is not proper to distinguish applications dependent on the application category. Rather than that we had to distinguish them through the used libraries and the application architecture. As a result we defined three main mobile application classes.

The second goal was to research important system libraries for the context of APM. This goal was reached simultaneously with the first one. In order to define a set of mobile application classes we had to deep dive in the used system libraries for this kind of application types.

The third goal was to design and implement the defined agent configurations. We successfully reached this goal. Chapter 5 describes all the agent strategies we focused on. Furthermore the functionality and the core concerns of the agent concepts were explained. The implementations of the agent strategies, based on the designs, are listed in Chapter 6. A huge challenge was to design and implement the agent strategy *Call Stack Sampling*, but we also managed to achieve this goal.

The last goal was to evaluate the mobile monitoring strategies based on different application types. In order to achieve this goal, we firstly evaluated general monitoring strategies theoretically in order to filter unimportant and less performant concepts for the practical evaluation. For the practical evaluation, we selected three different mobile applications with different application classes. Afterward we defined profiling use cases and profiled the application with the different monitoring configurations, in order to read out the performance overhead produced by the agents.

Future Work

This sections presents possible future work topics, which we were not able to focus on in the scope of this thesis.

- Architecture refactoring: In order to reduce the execution times of the agent configurations, especially when organizing spans, a refactoring of the implemented classes is needed. Instead of many switch-blocks within a method, a more modular framework can be implemented. This task would also have positive impacts in terms of runtime configuration.
- Refactoring of the call stack sampling implementation: Implementing the Call Stack Sampling Strategy was a challenge in this thesis. Even though, this strategy was less performant than expected. In order to accelerate the data collection and management process, it might be useful to analyze and re-factor unperformant methods in future.
- Compression of the serialized data: In order to save memory and to reduce the network connectivity usage, it might be useful to focus on minimizing the size of serialized data in future.
- Local measurement analysis: Filtering application monitoring data would reduce the overall memory usage of the monitoring agent. On the other hand, it may increase the response times of the application. It is essential to figure out how to deploy the filtering process in order to hold the application response time on the same level.
- Class versus structure: The programming language *Swift* also allows the application developer to create structures for custom data types. It would be important to analyze the performance of both options and to adapt the agent configurations in order to increase the performance.

Chapter 8

Bibliography

[APC] A. Inc. *Application Categories*. URL: https://developer.apple.com/app-store/categories/ (cit. on pp. 5, 25).

[APF] A. Inc. *iOS Frameworks*. URL: https://developer.apple.com/app-store/categories/ (cit. on p. 5).

[APL115] A. Inc. *Guides and Sample Code - Selector*. 2015. URL: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Selector.html (cit. on p. 11).

[APL217] A. Inc. *Understanding Low Memory Reports*. 2017. URL: https://developer.apple.com/library/content/technotes/tn2151/_index.html#//apple_ref/doc/uid/DTS40008184-CH1-UNDERSTANDING_LOW_MEMORY_REPORTS (cit. on p. 14).

[APL317] A. Inc. *Memory Warning*. 2017. URL: https://developer.apple.com/documentation/uikit/uiviewcontroller/1621409-didreceivememorywarning?preferredLanguage=occ (cit. on p. 14).

[APP17] A. Inc. *AppDynamics iOS SDK*. 2017. URL: https://docs.appdynamics.com/display/PRO43/Instrument+iOS+Applications (cit. on p. 16).

[APR17] S. Development. 2017. URL: http://appreviewtimes.com (cit. on p. 26).

[ATH14] A. Inc. *Threads*. 2014. URL: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Multithreading/AboutThreads/AboutThreads.html#//apple_ref/doc/uid/10000057i-CH6-SW2 (cit. on p. 51).

[AUDA17] A. Inc. *Audible audio-books file sizes*. 2017. URL: http://www.audible.com/audioformats (cit. on p. 26).

[AUDB17] A. Inc. *Audible Harry Potter and the Goblet of Fire, Book 4*. 2017. URL: https://www.audible.com/pd/Kids/Harry-Potter-and-the-Goblet-of-Fire-Book-4-Audiobook/B017V4NUPO/ref=a_search_c4_1_4_srTtl?qid=1501240285&sr=1-4 (cit. on p. 26).

[BS14] J. Bansal, B. Sunkara. *Performing call stack sampling*. US Patent App. 14/071,523. 2014. URL: http://www.google.com/patents/US20140068068 (cit. on pp. 13, 78).

[CFD98] A. C. Inc. *CFDictionary Header*. 1998. URL: https://opensource.apple.com/source/headerdoc/headerdoc-8.9.5/ExampleHeaders/CFDictionary.h.auto.html (cit. on p. 58).

[CMF+14] M. Chow, D. Meisner, J. Flinn, D. Peek, T. F. Wenisch. "The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services." In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014, pp. 217–231 (cit. on p. 11).

[COC17] B. Asher, D. Koutsogiorgas, D. Tomlinson, O. Therox, T. C. D. Team. *CocoaPods*. 2017. URL: url (cit. on p. 11).

[DEVP17] K. Angerbauer, T. Angerstein, A. Hidiroglu, S. Lehmann, M. Palenga, O. Röhrdanz, M. Sassano, C. Völker. *Mobile-aware Diagnosis of Performance Problems in Enterprise Applications*. 2017 (cit. on pp. 16, 18, 63).

[DYN17] D. LLC. *Dynatrace iOS app monitoring*. 2017. URL: https://www.dynatrace.com/technologies/mobile/ios-monitoring/ (cit. on pp. 16, 17).

[FEA90] K. K. C., C. S. G., H. J. A., N. W. E., P. A. S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990 (cit. on p. 90).

[GAR17] G. Inc. *Application Performance Monitoring (APM)*. 2017. URL: http://www.gartner.com/it-glossary/application-performance-monitoring-apm (cit. on p. 9).

[HHMO17] C. Heger, A. van Hoorn, M. Mann, D. Okanović. "Application Performance Management: State of the Art and Challenges for the Future." In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. L'Aquila, Italy: ACM, 2017, pp. 429–432 (cit. on p. 9).

[HKGH11] A. van Hoorn, H. Knoche, W. Goerigk, W. Hasselbring. "Model-Driven Instrumentation for Dynamic Analysis of Legacy Software Systems." In: *Proceedings of the 13. Workshop Software-Reengineering (WSR 2011)*. Also appeared in Softwaretechnik-Trends 31(2) (2011) 18?19. 2011, pp. 26–27 (cit. on pp. 14, 43).

[HRH+09] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, D. Kieselhorst. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Research Report. Kiel University, 2009 (cit. on p. 10).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. ACM, 2012, pp. 247–248 (cit. on p. 10).

[IAD17] A. Inc. *Apple iAd App Network*. 2017. URL: https://developer.apple.com/support/iad/ (cit. on p. 32).

[IHE15] O. Ibidunmoye, F. Hernández-Rodriguez, E. Elmroth. "Performance Anomaly Detection and Bottleneck Identification." In: *ACM Comput. Surv.* 48.1 (2015), 4:1–4:35 (cit. on p. 10).

[INS17] A. Inc. *Instruments*. 2017. URL: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/ (cit. on p. 104).

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. "Aspect-oriented programming." In: *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, 1997 Proceedings*. Ed. by M. Akşit, S. Matsuoka. Springer Berlin Heidelberg, 1997, pp. 220–242 (cit. on p. 14).

[LIG17] L. Inc. *LightStep iOS app monitoring*. 2017. URL: http://lightstep.com (cit. on p. 18).

[MUR17] A. Inc. *NSMutableURLRequest*. 2017. URL: https://developer.apple.com/documentation/foundation/nsmutableurlrequest (cit. on p. 29).

[NEW17] N. R. Inc. *New Relic iOS SDK*. 2017. URL: https://docs.newrelic.com/docs/mobile-monitoring/new-relic-mobile-ios/install-configure/work-ios-sdk-api (cit. on pp. 16, 17).

[NK14] R. Napier, M. Kumar. *IOS 7 Programming Pushing the Limits: Develop Advance Applications for Apple IPhone, IPad, and IPod Touch*. 2014 (cit. on pp. 11, 44).

[NSUI17] A. Inc. *NSUInteger*. 2017. URL: https://developer.apple.com/documentation/objectivec/nsuinteger (cit. on p. 50).

[OPT17] B. Cronin, B. Sigelman, B. Gonzalez, D. Kuebrich, M. Sembwever, P. Sharma, S. Gutekanst, W. Schottdorf Tobias Sheng, Y. Shkurom. *Opentracing framework*. 2017. URL: http://opentracing.io (cit. on pp. 18, 51, 52).

[SAR17] A. Inc. 2017. URL: https://developer.apple.com/documentation/swift/array (cit. on p. 22).

[SFU17] A. Inc. 2017. URL: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-ID158 (cit. on p. 21).

[SOP17] A. Inc. 2017. URL: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/OptionalChaining.html#//apple_ref/doc/uid/TP40014097-CH21-ID245 (cit. on p. 21).

[UIN17] A. Inc. *UInt*. 2017. URL: https://developer.apple.com/documentation/swift/uint (cit. on p. 50).

[URR17] A. Inc. *NSURLRequest*. 2017. URL: https://developer.apple.com/documentation/foundation/nsurlrequest (cit. on p. 29).

[USD17] A. Inc. *URLSessionDataTask*. 2017. URL: https://developer.apple.com/documentation/foundation/urlsessiondatatask (cit. on p. 31).

[USE17] A. Inc. *URLSession*. 2017. URL: https://developer.apple.com/documentation/foundation/urlsession (cit. on p. 30).

[USL17] A. Inc. *URLSessionDownloadTask*. 2017. URL: https://developer.apple.com/documentation/foundation/urlsessiondownloadtask (cit. on p. 31).

[USU17] A. Inc. *URLSessionUploadTask*. 2017. URL: https://developer.apple.com/documentation/foundation/urlsessionuploadtask (cit. on p. 31).

[WKN17] A. Inc. 2017. URL: https://developer.apple.com/documentation/webkit/wknavigationdelegate (cit. on p. 29).

[WKU17] A. Inc. 2017. URL: https://developer.apple.com/documentation/webkit/wkuidelegate (cit. on p. 28).

[WKW17] A. Inc. 2017. URL: https://developer.apple.com/documentation/webkit/wkwebview (cit. on p. 28).

[XCO17] A. Inc. *Xcode*. 2017. URL: https://developer.apple.com/xcode/ (cit. on p. 104).

All links were last followed on January 02, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature