Estudo comparativo de *frameworks* multiplataforma de desenvolvimento de aplicações móveis

Mestrado em Engenharia Informática – Computação Móvel

Dany Lopes da Mota

Leiria, Setembro de 2020

(Esta página foi deixada em branco propositadamente)

Estudo comparativo de *frameworks* multiplataforma de desenvolvimento de aplicações móveis

Mestrado em Engenharia Informática – Computação Móvel

Dany Lopes da Mota

Dissertação realizada sob a orientação do Professor Doutor Ricardo Martinho, Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria

Leiria, Setembro de 2020

(Esta página foi deixada em branco propositadamente)

Originalidade e Direitos de Autor

A presente dissertação é original, elaborada unicamente para este fim, tendo sido devidamente citados todos os autores cujos estudos e publicações contribuíram para a elaborar.

(Esta página foi deixada em branco propositadamente)

Dedicatória

Dedico este trabalho a todas as pessoas que me apoiaram incansavelmente das mais diversas formas, durante todo o percurso que fiz até à conclusão deste.

(Esta página foi deixada em branco propositadamente)

Agradecimentos

Aqui agradeço todo o apoio que recebi sob as mais diversas formas, no desenvolvimento deste trabalho:

Ao Professor Doutor Ricardo Martinho, como meu orientador mostrou-se sempre disponível para me prestar apoio.

À minha família e amigos que sempre estiveram disponíveis para me ouvir as diversas frustrações que foram surgindo e não me deixarem perder o foco que tinha para a conclusão de uma etapa tão importante como esta.

Ao Instituto Politécnico de Leiria pelas condições prestadas.

(Esta página foi deixada em branco propositadamente)

Resumo

Atualmente existem tecnologias na área do desenvolvimento de aplicações de software multiplataforma. Estas tecnologias permitem criar uma aplicação com uma única base de código em linguagens de programação mais básicas, que posteriormente irá ser transformada para aplicações executáveis em múltiplos sistemas operativos como o Android OS ou o iOS, proporcionando diversas vantagens tanto a nível de desenvolvimento como na manutenção das aplicações.

Neste trabalho, inicialmente era pretendendido realizar um estudo de comparação entre as duas das *frameworks* multiplataforma mais dominantes da atualidade. No entanto, uma comparação entre duas *frameworks* pode facilmente tornar-se obsoleta. Isto pode acontecer pelo facto de existir uma grande probabilidade de que, após a disponibilização de uma nova versão de uma das *frameworks*, a conclusão obtida no estudo já não corresponder à realidade. Deste modo, foram analisados vários estudos que envolviam a comparação de múltiplas tecnologias e abordagens para a criação de aplicações multiplataforma, com o intuito de encontrar possíveis lacunas. No decorrer do estudo foram encontrados diversos lapsos que eram praticados e que colocavam em causa todo o trabalho de análise. Tendo em conta a existência destas lacunas, e ainda a inexistência de um procedimento de testes e métricas padronizado, foi definido como objetivo principal o estudo e a criação de um procedimento completo de como fazer uma comparação da *performance* de aplicações móveis, desenvolvidas recorrendo a diferentes tecnologias ou abordagens, com a ambição de se tornar num procedimento reconhecido e amplamente usado noutros estudos comparativos. Assim, não só este trabalho envolve a comparação de duas tecnologias de desenvolvimento multiplataforma, como ainda tem um objetivo maior: criar um procedimento credível e exequível para a comparação de *performance* entre aplicações móveis desenvolvidas quaisquer tecnologias. Contudo, a sua concretização revelou-se mais complexa do que o esperado, devido a determinados aspetos que faltavam ser estudados e definidos e à inexistência de utensílios práticos.

Um ponto que é extremamente importante de se encontrar em qualquer estudo comparativo, cujo objetivo seja perceber qual a melhor alternativa no desenvolvimento de uma aplicação móvel, é o modo de *release*. Este termo é algo que é praticamente inexistente nos diversos estudos que foram analisados, no entanto foi descoberto através de estudos paralelos,

nomeadamente na observação de uma palestra realizada pela Apple, relacionada com uma ferramenta de auxílio ao desenvolvimento, o Instruments. Além do modo de *release*, havia ainda outros pontos importantes a definir, como o caso das funcionalidades a testar e quais as métricas a utilizar na avaliação.

Visto que a comparação da *performance* entre aplicações deve ser feita em modo *release*, também as ferramentas devem suportar este modo. A ferramenta de medição de *performance* disponibilizada pela Apple suporta este modo de compilação, mas as ferramentas mais atuais disponibilizadas pela Google para o auxílio no desenvolvimento de aplicações Android não suportam. Surge assim um dos maiores desafios encontrados no decorrer deste trabalho. Foi necessário recorrer a ferramentas mais antigas e mais trabalhosas, por forma a criar vários *scripts* e mecanismos de execução de testes para dispositivos Android que pudessem ser compatíveis com este modo de *release*.

Depois de todos os pontos anteriormente referenciados estarem resolvidos, foram definidas nove funcionalidades de software mais comuns em aplicações móveis a serem testadas, de onde resultou o desenvolvimento de cinquenta e duas (mini) aplicações móveis e ainda todo o procedimento de testes a realizar às mesmas. Posteriormente, foi feita a análise dos resultados de teste obtidos e tiradas conclusões. Das conclusões obtidas da comparação realizada entre as *frameworks* React Native e Flutter (aquelas previamente selecionadas tendo em conta vários critérios), pode concluir-se que, maioritariamente, a Flutter é a melhor solução para as funcionalidades selecionadas, independentemente do sistema operativo alvo do dispositivo de *smartphone* a ser utilizado, entre os sistemas iOS e Android OS.

Palavras-chave: Aplicações móveis multiplataforma, *Performance*, Flutter, React, iOS, Android

Abstract

The use of multiplatform software development frameworks has become quite common nowadays. These technologies allow you to create an application in a single code base, and later compile/deploy it in several platforms, such as Android or iOS, providing several advantages both in terms of application development and maintenance.

In this work, we intended to carry out a comparison study between two of the most dominant multiplatform frameworks currently used. However, a comparison between two technologies can easily become obsolete. This can happen because there is a high probability that, after the release of a new version of one of the technologies, the conclusion obtained in the study becomes outdated. In this way, several studies were analyzed that involved the comparison of multiple technologies and approaches for the creation of multiplatform applications, in order to find possible gaps. During literature review, several limitations were found, as well as the inexistence of a standardized testing and metrics procedure. This way, it was defined, as an objective, to create a complete procedure on how to make a comparison between mobile applications developed using different technologies/frameworks, with the ambition of becoming a procedure recognized and widely used in other similar comparative studies. Thus, not only does this work involves the objective that motivated it initially, but it also has a larger scope: to create a credible and feasible procedure for comparing performance between mobile applications developed in different technologies. However, its implementation proved to be more complex than expected, due to certain aspects that needed to be studied and defined and the lack of practical tools.

A point that is extremely important to find in any comparative study, whose objective is to understand which is the best alternative in the development of an application, is the *release mode*. This term is something that is practically nonexistent in the various studies that were analyzed. However, it was discovered through parallel studies, namely in the observation of a lecture given by Apple, related to their performance measurement tool: Instruments. In addition to *release mode*, there were still other important points to define, such as the features to be tested and which metrics to use in the evaluation.

Since performance comparison between mobile applications must be done in release mode, the tools used to run the applications and measure their performance must also support this

mode. It turns out that the tool made available by Apple supports this compilation mode, but the most currently used tools made available by Google to assist in the development of Android applications do not. Thus, one of the biggest challenges encountered during this work arises: it was necessary to resort to older and more basic tools in order to create various *scripts* and test execution mechanisms for measuring performance of mobile apps for Android devices.

After solving all the previously mentioned points, nine software features for mobile apps were elected for testing, which resulted in the development of fifty-two (mini) applications and the entire testing procedure to be carried out on them. Subsequently, the analysis of the obtained testing results was made and conclusions were drawn. From the conclusions obtained from the comparison made between the React Native and Flutter frameworks (the ones selected for this study through several criteria), it can be concluded that, for the most part, Flutter is the best solution for the selected features, regardless of the target operating system of the smartphone device, when considering iOS and Android OS.

Keywords: Multiplatform mobile applications, Performance, Flutter, React, iOS, Android

Índice

(Esta página foi deixada em branco propositadamente)

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Acrónimos

ADB	*Android Debug Bridge*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
CSS	*Cascading Style Sheet*
CSV	*Comma-separated values*
FPS	*Frames per second*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
PCs	*Personal Computers*
RAM	*Random Access Memory*
SDK	*Software Development Kit*
UI	*User Interface*

(Esta página foi deixada em branco propositadamente)

1. Introdução

Sendo este o primeiro capítulo deste documento, pretende-se dar a conhecer o contexto do trabalho realizado, assim como os objetivos que almejámos atingir no decorrer da sua realização. Por fim, será apresentada a estrutura que rege este relatório.

Aplicações multiplataforma são aplicações desenvolvidas com recurso a tecnologias que permitem criar uma aplicação com suporte em mais do que um único sistema operativo. Esta abordagem de desenvolvimento pode oferecer diversas vantagens a diversas entidades, tais como o desenvolvimento da aplicação, com a redução do número de equipas envolvidas, até à complexidade exigida aos *developers*. No entanto, as vantagens não se ficam por aqui e por este mesmo motivo este é um tema de grandes atenções no que toca ao desenvolvimento de aplicações móveis.

1.1. Contexto

Na indústria dos *smartphones* existe uma diversidade de sistemas operativos. No entanto, o mercado tem-se concentrado mais em apenas dois deles. No segundo trimestre de 2018, cerca de 99,9% do mercado era dominado pelos sistemas operativos Android e iOS, como pode ser visto na Figura 1. O Android é o sistema operativo com a maior prevalência no mercado, com cerca de 88%. O iOS, sendo o segundo a dominar o mercado, apenas contém 11,9% [1]. Para que uma empresa de desenvolvimento de *software* alcance esta percentagem de mercado dividido entre Android e iOS, terá, normalmente, de desenvolver aplicações separadas para os diferentes sistemas operativos, utilizando, para cada uma delas, o respetivo Software Development Kit (SDK). O desenvolvimento de duas aplicações que apenas se distinguem pelo facto de se destinarem a sistemas operativos diferentes, pode significar um investimento acrescido, tanto no seu desenvolvimento como na sua manutenção corretiva e evolutiva. Isto porque o desenvolvimento será feito em, pelo menos, duas linguagens de programação diferentes (Java e/ou Kotlin para Android e Swift e/ou Objective-C para iOS), implicando normalmente duas equipas de programadores separadas, e duas aplicações com meios de *deployment*, correção, manutenção e evolução distintos.

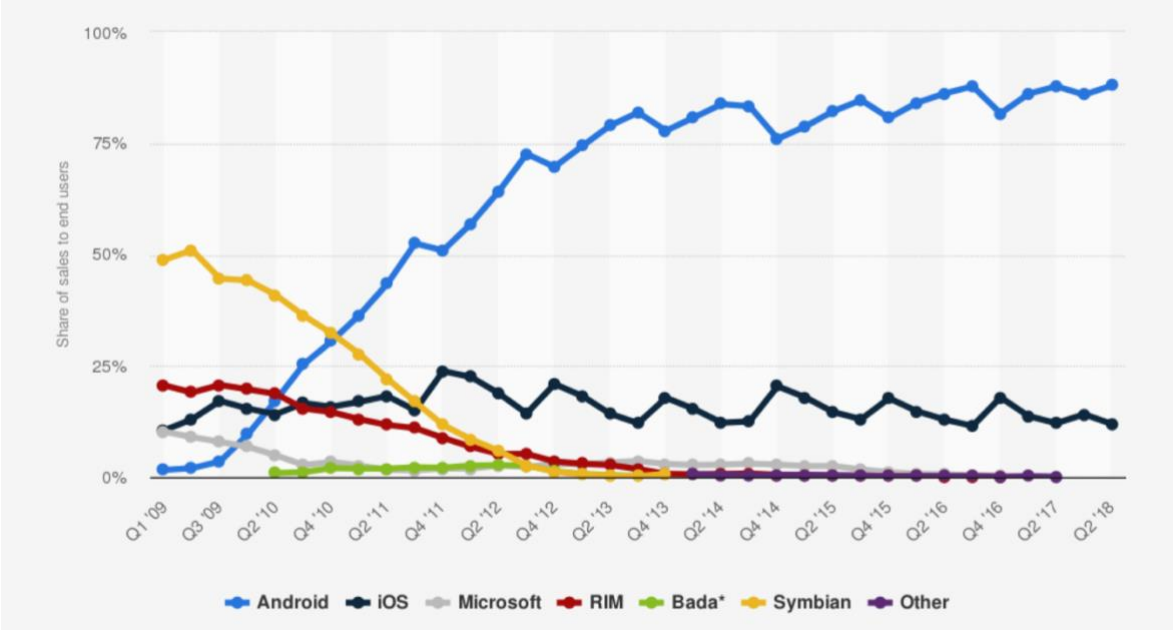


Figura 1 - Gráfico demonstrativo da evolução da partilha de mercado pelos distintos sistemas operativos (Fonte: [1]**)**

Foi para preencher esta necessidade que começaram a surgir tecnologias com opções de desenvolvimento para múltiplas plataformas em simultâneo. O desenvolvimento multiplataforma é atualmente suportado por diversas *frameworks* distintas. Existem *frameworks* que fazem uso de tecnologias muito semelhantes às utilizadas no desenvolvimento de *websites*, como é o caso da Ionic que faz uso de HTML (HyperText Markup Language), CSS (Cascading Style Sheet) e JavaScript [2], e outras *frameworks* que utilizam outras linguagens menos conhecidas na área de desenvolvimento web como é o caso da Flutter que utiliza Dart [3].

1.2. Problema

Existem vários estudos que comparam estas diferentes tecnologias e *frameworks*, com ferramentas e métricas para diferentes propósitos. No entanto, alguns destes estudos evidenciam lacunas na forma como testam estas tecnologias, ou na forma como selecionam e utilizam ferramentas e métodos que poderão não ser os mais corretos (e.g., [4, 5]). Isto, possivelmente, deve-se à falta da existência de um processo devidamente definido de como proceder para a realização correta de comparações de *performance* entre aplicações.

Também são encontrados estudos, como é o caso do estudo [6], que fazem uma comparação de *frameworks* apenas através do tempo de execução de determinadas funcionalidades. Um exemplo é a medição do tempo para ativar a vibração dos dispositivos. A *performance* de

aplicações móveis desenvolvidas em determinada *framework* inclui outros aspetos para além da medição de tempos, pois pode suceder que, por exemplo, a *framework* mais rápida a ativar determinada funcionalidade, leve o dispositivo a um colapso de *hardware*, sendo uma opção duvidosa para desenvolver uma aplicação por completo.

1.3. Motivação e objetivos

A existência de diversas abordagens e tecnologias que possibilitam o desenvolvimento de aplicações para dispositivos móveis, acaba por deixar a comunidade de desenvolvimento de software em dúvida sobre qual destas é que devem utilizar nos seus projetos. Um dos objetivos deste trabalho é precisamente ajudar a perceber qual a que melhor se adequa aos objetivos pretendidos em cada aplicação a ser desenvolvida.

Contudo, o maior objetivo deste trabalho é a definição e criação de todo um processo de comparação de aplicações móveis, que envolve um largo conjunto de procedimentos e testes que realmente possam ser implementados para a comparação de *performance* de aplicações móveis desenvolvidas através de quaisquer abordagens ou tecnologias.

Anteriormente, não existia a mesma necessidade atual de comparar duas aplicações iguais, desenvolvidas por tecnologias distintas pelo simples facto de cada sistema operativo ter a sua linguagem oficial. Possivelmente devido à falta desta necessidade, as entidades responsáveis pelos sistemas operativos concentraram as ferramentas disponibilizam mais para o auxílio no desenvolvimento de aplicações e não para uma avaliação do que é e/ou foi desenvolvido.

Assim, a falta de ferramentas devidamente prontas para uma correta avaliação e comparação de performance entre *frameworks* multiplataforma, e o facto de não existir qualquer procedimento predefinido, nem orientações para a realização de uma avaliação e análise corretas da *performance* de aplicações, constituíram os maiores desafios no decorrer de todo o percurso deste trabalho.

1.4. Estrutura do documento

Abordado o âmbito do trabalho em questão e quais os seus objetivos, passo a apresentar a estrutura deste relatório, oferecendo uma visão global de todo o trabalho desenvolvido para a execução deste estudo. A estrutura deste relatório está fracionada em sete capítulos

distintos, sendo estes: Introdução, Trabalho Relacionado, Metodologias, Aplicações e Casos de Teste, Configurações de Testes, Resultados e Discussão e Conclusão.

No que se refere à Introdução, esta é constituída por uma nota introdutória, a apresentação da motivação, problemas e objetivos deste trabalho, e, por fim, a apresentação da estrutura do documento, na qual se procura enquadrar o leitor relativamente ao estudo em questão.

O segundo capítulo, Trabalho Relacionado, incide na abordagem de todo o meio ambiente já existente da área em que este trabalho assenta. Comentar o estudo de trabalhos já existentes, o que é o desenvolvimento multiplataforma e quais as tecnologias mais influentes da atualidade são alguns dos pontos abordados no capítulo.

Na Metodologia é pretendido apresentar a forma de organização e estruturação do processo de estudo e desenvolvimento do trabalho envolvido para este relatório e quais as principais etapas que este seguiu.

Aplicações e Casos de Teste é o capítulo que expõe não só as aplicações desenvolvidas nas *frameworks* a serem comparadas, resultantes das funcionalidades a serem avaliadas, mas também o procedimento de testes realizados a cada uma e as respetivas métricas aplicadas.

No capítulo Configurações e Testes são abordados pontos importantes para a comparação de *performance* de aplicações. Alguns desses pontos incluem o ambiente de testes, procedimento de testes, ferramentas e métricas de avaliação.

O sexto capítulo, Resultados e Discussão é o capítulo de maior dimensão e é onde são apresentados todos os resultados do trabalho relatado nos capítulos anteriores e onde estes são analisados e comparados para tirar as conclusões pretendidas, desde que foi definido o objetivo deste trabalho.

Por último, a Conclusão reflete o enquadramento do conhecimento adquirido ao longo de todo o período deste trabalho, assim como as conclusões obtidas. Neste capítulo também é descrito o trabalho ambicionado para o futuro, por forma a dar continuidade a este projeto e à sua evolução.

2. Trabalho relacionado

Antes de iniciar este trabalho, houve a preocupação de perceber que outros estudos já haviam sido realizados para contextualizar e compreender diversos aspetos relacionados com o tema. Foi feita uma pesquisa no motor de busca Google Scholar com as seguintes palavras-chave: "*cross platform frameworks*" OR "*hybrid mobile frameworks*" OR "*native mobile frameworks*" *performance metrics*. Na pesquisa foi aplicado um filtro, para serem apresentados apenas os resultados posteriores a 2018, inclusive. Com esta pesquisa, foram obtidos 121 resultados, dos quais foram rejeitados 100 por não se enquadrarem corretamente no tema. As 21 publicações analisadas encontram se listadas na Tabela 1.

Destes trabalhos foram recolhidas diversas informações que auxiliaram o desenvolvimento deste trabalho, nomeadamente:

- Que ferramentas eram usadas;
- Quais as métricas utilizadas;
- Quais as funcionalidades que eram testadas.

Atualmente, tem sido observado um crescente interesse na área de desenvolvimento de aplicações multiplataforma, que não só é percetível pelo elevado número de estudos já realizados, como também através de uma simples pesquisa no motor de busca da Google, com as palavras-chave "*multiplatform app development*" onde, à data, foram obtidos mais de quatro milhões de resultados.

Os estudos selecionados para análise tinham como objetivo perceber qual a melhor forma de desenvolver estas aplicações. No entanto, verificou-se que não é possível definir qual a melhor *framework*, mas sim a que melhor se adequa a uma determinada situação, dependendo do propósito e dos requisitos da aplicação de software que se pretende. Contudo, é necessário perceber a melhor forma de avaliar uma aplicação, para posteriormente poder ser feita uma comparação das diversas alternativas que pretendemos avaliar.

Tabela 1 - Lista dos 21 estudos analisados

Animations In Cross-Platform Mobile Applications: An Evaluation Of Tools, Metrics And Performance, [5]
Performance Comparison Between React Native And Flutter, [7]
Evaluating Performance Of A React Native Feature Set, [8]
A Comparison Of Ionic 2 Versus React Native And Android In Terms Of Performance, By Comparing The Performance Of Applications, [9]
Performance Analysis Of Mobile Cross-Platform Development Approaches Based On Typical Ui Interactions, [10]
Android-Platform Based Determination Of Fastest Cross-Platform Framework, [11]
Performance Breakdown Of React Native Framework Compare To Native Applications, [12]
React Native Performance Evaluation, [13]
Impact Of Cross-Platform Development Frameworks On The Performance Of Mobile Communications For Short Distances, [14]
Evaluating The Mobile Development Frameworks Apache Cordova And Flutter And Their Impact On The Development Process And Application Characteristics, [4]
Evaluating Cross-Platform Mobile App Performance With Video-Based Measurements, [6]
Exploring End User's Perception Of Flutter Mobile Apps, [15]
Cross-Platform Development Frameworks For The Development Of Hybrid Mobile Applications: Implementations And Comparative Analysis, [16]
Ott Video-Oriented Mobile Applications Development Using Cross-Platform Ui Frameworks, [17]
Usability And User Experiencein Mobile App Frameworks, [18]
A Study Of Cordova And Its Data Storage Strategies, [19]
React Native Vs Flutter, Cross-Platform Mobile Application Frameworks, [20]
Cross-Platform Smartphone Application Development With Kotlin Multiplatform, [21]
Estudio Comparativo De Frameworks Multiplataforma Para Desarrollo De Aplicaciones Móviles, [22]
Using React Native For Mobile Software Development, [23]
React Native Vs Xamarin – Mobile For Industry, [24]

A avaliação de *performance* de uma aplicação é um processo complexo, que facilmente pode tornar-se inválido por não conter determinados procedimentos, que são cruciais para ser considerada fiável. Um desses passos é a avaliação das aplicações única e exclusivamente

em *Release Mode*; ou, em alternativa, num modo dedicado à avaliação das aplicações (caso disponível), tendo em conta as *frameworks* que estão a ser avaliadas [25, 26].

Este trabalho visa satisfazer uma necessidade comum a uma grande comunidade que engloba desde programadores a grandes empresas: perceber qual o melhor caminho a seguir para o desenvolvimento dos seus projetos. Para isso é necessário chegar a uma conclusão em diversos aspetos:

- Quais as ferramentas mais adequadas para a recolha da *performance* das aplicações;
- Quais as métricas usadas na avaliação;
- Quais as funcionalidades mais usadas para serem implementadas e avaliadas.

O desenvolvimento de aplicações para dispositivos móveis pode considerar-se dividido em duas abordagens: 1) o desenvolvimento através das linguagens, SDKs e *frameworks* nativas dos fornecedores dos sistemas operativos, e 2) a utilização de tecnologias alternativas para um desenvolvimento multiplataforma. Ambas as abordagens têm características próprias e, dependendo da finalidade da aplicação, trazem pontos positivos e pontos menos positivos.

2.1. Desenvolvimento nativo

O desenvolvimento nativo consiste em criar uma aplicação para determinado sistema operativo em particular, utilizando as tecnologias recomendadas pelo fornecedor desse sistema operativo (linguagem de programação, SDKs, bibliotecas, Integrated Development Environment (IDE)). A grande vantagem desta abordagem é que todas as funcionalidades dos dispositivos e sistema operativo, em que a aplicação vai ser usada estão disponíveis de forma natural.

Quando uma aplicação é desenvolvida nativamente, são utilizadas as linguagens oficiais de cada sistema operativo. Java e/ou Kotlin, quando a aplicação se destina ao sistema operativo Android OS e Swift e/ou Objective-C, quando é destinada ao iOS. Como o seu desenvolvimento é nativo, as aplicações resultantes têm acesso ao hardware do dispositivo e suportam todas as interfaces e interações do utilizador disponíveis no respetivo ambiente operativo móvel, de uma forma natural e sem barreiras de incompatibilidade [27].

2.2. Desenvolvimento Multiplataforma

O desenvolvimento multiplataforma consiste no desenvolvimento de uma aplicação que irá ser suportada por mais que uma única plataforma em simultâneo. Esta abordagem pode seguir diversas vertentes, sendo as principais: *Web applications*, *Hybrid*, *Interpreteted* e *Cross-compliled*. Como apresentado no diagrama da Figura 2, existe uma evolução no que toca à aproximação de uma abordagem nativa [5].

Figura 2 - Diagrama do grau de aproximação das diferentes abordagens em relação à nativa (adaptado de [22]**).**

2.2.1. Web Applications

A criação de uma aplicação *web* direcionada aos dispositivos móveis é uma das abordagens de desenvolvimento multiplataforma mais cómoda e abrangente. Isto deve-se ao facto de, atualmente, todos os *smartphones* terem a capacidade de suportar as tecnologias *web* modernas como é o caso das linguagens HTML5, CSS3 e JavaScript. Portanto, através desta abordagem, as equipas de desenvolvimento têm a capacidade de criar uma aplicação que facilmente é acedida em qualquer *smartphone*. Uma das tecnologias mais conhecidas na área das *Web Applications* é a PWA - *Progressive Web Application*, introduzida pela Google em 2015. Com as PWAs, a aplicação que é desenvolvida traz inúmeras vantagens a um *website* comum. Para uma PWA, é possível armazenar dados em cache para que possa haver um acesso *offline* e ainda permite aceder a determinados componentes de *hardware*.

No entanto, uma aplicação destas tem as suas desvantagens devido a limitações naturais de comunicações entre os componentes nativos dos dispositivos. Uma aplicação que seja desenvolvida como PWA não pode ser submetida nos *marketplaces* nativos, para que esta esteja disponível nos sistemas operativos como uma aplicação comum [4, 28].

2.2.2. Hybrid

A abordagem *Hybrid* no desenvolvimento multiplataforma é uma combinação de elementos *web* com elementos nativos. Neste tipo de abordagem, existe um aglomerado de tecnologias, como é o caso das *frameworks* Apache Cordova [29] , Ionic [2] e do Adobe PhoneGap [30]. Todas estas *frameworks* permitem aos programadores o uso de diversas tecnologias *web* para criar aplicações, que são incorporadas num componente *WebView* nativo, que acaba por

fazer um papel idêntico a um *browser* exclusivo e embutido da aplicação. O desenvolvimento *Hybrid* é idêntico a uma PWA, mas com melhorias no que toca ao acesso a funcionalidades dos dispositivos, *performance* e ainda permite que a aplicação seja submetida para o mercado de aplicações, tanto no caso do Android para a PlayStore, como no iOS para a AppStore [4].

Ainda assim, as aplicações podem ainda apresentar problemas de *performance*, especialmente a nível de processamento e renderização de conteúdos gráficos. A aparência das aplicações também dificilmente é a de uma aplicação nativa, por determinados componentes nativos não estarem acessíveis a esta abordagem [17].

2.2.3. Interpreteted

Na abordagem *Interpreteted*, as aplicações originadas aproximam-se mais das aplicações nativas, tanto em termos de *performance* como nas suas interações e comportamentos. As *frameworks* que usam uma abordagem interpretada utilizam um intérprete que, durante o tempo de execução, sempre que necessário, faz chamadas às APIs (*Application Programming Interfaces*) nativas de cada sistema operativo [4, 17]. Isto acontece porque os dispositivos Android e iOS incorporam um interpretador de JavaScript, também conhecido como JavaScript engine, denominado JavaScriptCore (JSCore). O JSCore é responsável por processar todo o código JavaScript que é produzido pelos programadores, para instruções específicas da plataforma, em tempo real. Um exemplo de uma *framework* desta categoria é a React Native [31]. Através desta abordagem, o acesso às APIs nativas dos dispositivos é total, permitindo assim criar aplicações com uma UI (*User Interface)* extremamente próxima de uma nativa. As aplicações criadas podem ser disponibilizadas de forma natural nos *marketplaces*.

Contudo, tem determinadas desvantagens: ainda que as aplicações possam ter uma *performance* superior relativamente às abordagens anteriores, o seu desempenho pode ser negativamente afetado por ter de ser interpretada em tempo real durante a sua execução [17].

2.2.4. Cross-Compiled

No desenvolvimento multiplataforma com *frameworks* que usam uma abordagem *Cross-Compiled*, as aplicações são criadas numa linguagem diferente da nativa e posteriormente compilada, através de um compilador próprio, numa aplicação próxima de nativa. Um exemplo de uma *framework* que faz uso deste método é a Flutter. Assim como na abordagem

anterior, as aplicações têm acesso a todas as APIs dos dispositivos, o que possibilita a criação de aplicações idênticas às nativas. Também têm um desempenho muito superior às aplicações *web* e híbridas; segundo a Google, a Flutter tem mesmo a capacidade de obter desempenhos semelhantes a aplicações nativas [3].

Uma aplicação desenvolvida através da abordagem *Cross-Compiled* pode, ainda assim, contar com determinadas desvantagens, como é o caso do seu tamanho, que pode ser superior ao das aplicações nativas [17].

2.3. Frameworks multiplataforma

Tendo em conta que um dos objetivos deste trabalho é o desenvolvimento de um cenário de testes que permita avaliar e definir qual a melhor escolha relativamente às *frameworks* utilizadas atualmente no desenvolvimento multiplataforma, foi necessário fazer uma seleção de *frameworks* para esse efeito.

Para isso, foi realizada uma pesquisa para determinar quais as duas *frameworks* que mais sucesso têm atualmente na comunidade. A lista inicial é constituída pelas seguintes opções:

- Flutter [3];
- Ionic [2];
- React Native [32];
- Vue Native [33].

A seleção destas *frameworks* teve como base as que foram abordadas mais do que 5 vezes, nos diversos estudos analisados, que sejam *open source*, que não tenham custos associados e que reúnam maior preferência na comunidade (ver Apêndice A). Relativamente ao Vue Native, esta *framework* não foi referenciada em nenhum dos estudos, sendo no entanto uma das mais recentes *frameworks* e de já ter uma presença bastante significativa, com *Watchs* e *Stars* no repositório oficial múltiplas vezes superior ao Cordova ou PhoneGap que, para além disto, não cumprem o primeiro critério de seleção (são abordadas somente 2 vezes nos estudos considerados).

Para se obter as duas *frameworks* a serem estudadas no final desta seleção, procedeu-se a duas recolhas de dados, com oito meses de intervalo entre si, com o objetivo de avaliar as *frameworks* que mais progrediram na sua relação com a comunidade.

No entanto, para uma seleção mais rápida, o primeiro critério de filtragem foi a exclusão das *frameworks* que não tivessem lançado uma *release* no ano de 2020 (*releases* de cada *framework* disponíveis em [34 - 37]). Como se pode verificar na Tabela 2, a *framework* Vue Native é das primeiras a ser excluída.

Tabela 2 – Informações relativamente a *releases* das *frameworks* iniciais

	Framework	1.ª Release	Última Release
1.ª Avaliação	Flutter	2017	2019
	Ionic	2015	2019
	React Native	2015	2019
	Vue Native	2018	2019
2.ª Avaliação	Flutter	2017	2020
	Ionic	2015	2020
	React Native	2015	2020
	Vue Native	2018	2019

Depois desta filtragem ficaram apenas três *frameworks*: Flutter, Ionic e React Native. Para uma seleção final, foram usados dados relativamente aos repositórios GitHub[1] que cada *framework* tem. Nas Tabelas 3 e 4, é possível observar quais as *frameworks* que mais cresceram na comunidade e quais as que se encontram mais ativas no que toca a atualizações de código [38 - 40].

Tabela 3 - Informações relativamente aos dados dos repositórios de cada tecnologia

	Framework	***GitHub Stars***	***GitHub Forks***	***GitHub Watchs***
1.ª Avaliação	Flutter	78800	10000	2500
	Ionic	39500	13100	1800
	React Native	82500	18500	3700
2.ª Avaliação	Flutter	93400	12700	2800
	Ionic	41000	13200	1800
	React Native	87600	19500	3700

[1] https://github.com/

Tabela 4 - Crescimento percentual de cada *framework* entre os dois momentos de recolha de informação.

Framework	GitHub Stars	GitHub Forks	GitHub Watchs
Flutter	19%	27%	12%
Ionic	4%	1%	0%
React Native	6%	5%	0%

Tendo em conta a informação das tabelas e os critérios de seleção que foram definidos anteriormente, as *frameworks* selecionadas para serem usadas neste estudo foram Flutter e React Native. Ainda com a informação recolhida nesta pesquisa, foi possível perceber que a Flutter é uma das *frameworks* mais recentes e atingiu níveis de crescimento muito superiores, o que pode indicar que permanecerá entre as preferidas pela comunidade nos próximos tempos.

2.3.1. Flutter

A Flutter é uma *framework* de desenvolvimento de aplicações multiplataforma, fornecida através de um SDK (*Software Development Kit*), para criar aplicações de alto desempenho e fiabilidade para sistemas operativos Android, iOS e, mais recentemente, veio permitir ainda o desenvolvimento de aplicações *web* e sistemas operativos para Personal Computers (PCs). Tudo isto através de uma única base de código: o Dart [41]. A Google criou esta *framework* e lançou a sua primeira versão em 2017 [42] com o objetivo de permitir aos programadores a criação de aplicações com alto nível de desempenho, mantendo o aspeto de aplicações nativas. Antes da Flutter tornar-se popular, o Dart era uma linguagem desconhecida por grande parte dos programadores. No entanto, como se pode ver no gráfico da Figura 3, as estatísticas apontam para que tenha sido uma das linguagens mais procuradas nos finais de 2018 [17].

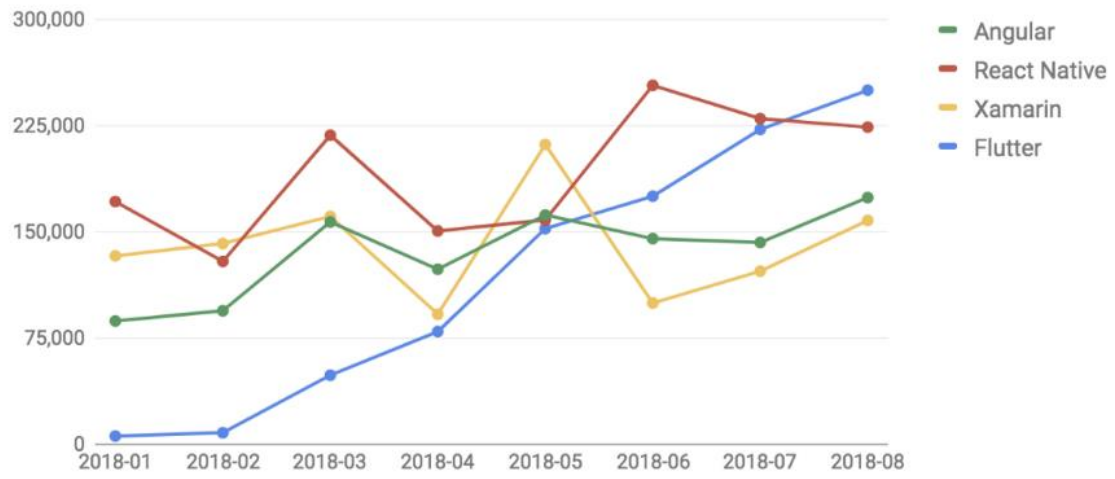


Figura 3 - Evolução da popularidade da Flutter nos primeiros oitos meses de 2018 (Fonte: [17]**)**

Para o desenvolvimento de aplicações, a *framework* Flutter permite utilizar apenas o Dart. Porém, Flutter é constituído não só de Dart, mas também de C, C++ e Skia, um mecanismo de renderização 2D [43]. Ao contrário das outras *frameworks* já nomeadas neste trabalho, a Flutter não depende de linguagens *web* para exibir os seus componentes. Utiliza, em vez disso, a abordagem de *Cross-Compiled*, o que significa que o código Dart é compilado em bibliotecas nativas ARM e x86 [44]. Este método possibilita à Flutter ser mais rápida em determinadas situações que outras tecnologias, como a React Native, porque não necessita de um interpretador [17].

A Flutter funciona muito com base em *widgets*, que são elementos que afetam e controlam a UI da aplicação. Os *widgets* são os blocos de construção mais importantes numa aplicação em Flutter; de facto, em Flutter, tudo é um *widget*. É através do mecanismo gráfico de renderização incorporado na Flutter, o Skia, que os *widgets* são renderizados [16].

2.3.2. React Native

A React Native é uma *framework* que segue a abordagem *Interpreteted*, que foi lançada em 2015 pela Facebook. A React Native foi criada com base numa biblioteca de JavaScript também criada pela Facebook, a ReactJs. Lançada em 2013, a ReactJs é uma biblioteca *open-source* usada para criar interfaces de interação com utilizadores, possibilitando a divisão da interface em componentes de fácil reutilização [8]. A React Native segue os mesmos conceitos da ReactJs, no entanto, em vez de renderizar HTML, usa componentes UI nativos das plataformas.

Na framework React Native existem duas *threads* principais, a Main ou UI *thread* e a JavaScript (JS) *thread*. A Main *thread* é a responsável pela UI e por comunicar com o sistema nativo. A JS *thread* é responsável por interpretar o código criado. Ambas as *threads* podem comunicar através de um componente interno da React Native chamado Bridge, que assume qualquer comunicação entre os componentes nativos e a JS [7, 13].

2.4. Métricas

Para proceder a uma comparação e, posteriormente, a uma avaliação, é necessário estabelecer métricas comparativas. Desta forma, aquando da análise dos 21 estudos selecionados e referidos anteriormente (Ver Tabela 1 da secção 2), foram levantadas as métricas que estes faziam uso. Deste processo de recolha, resultou um vasto conjunto de métricas, sendo as 10 mais encontradas as seguintes:

- Consumo de CPU (*Central Processing Unit*);
- Consumo de memória RAM (*Random Access Memory*);
- Tempo de Execução;
- Número de *frames* por segundo;
- Tamanho da aplicação, quando instalada no dispositivo;
- Consumo de energia;
- Consumo de memória gráfica;
- Tamanho do executável da aplicação;
- Qualidade do código;
- Linhas de Código.

Por forma a ser selecionado um número de métricas exequível para serem utilizadas neste estudo, foi feita uma filtragem pelas métricas com mais de 5 ocorrências nos estudos previamente selecionados. A lista de métricas ficou assim constituída por 4 elementos:

- Utilização de CPU, com 9 ocorrências;
- Consumo de memória RAM, com 9 ocorrências;
- Tempo de execução, com 8 ocorrências;
- Número de FPS (*frames* por segundo), com 7 ocorrências.

2.5. Ferramentas de medição

Para um processo de avaliação são necessárias métricas e, por sua vez, ferramentas que tenham capacidade de recolher informações sobre as mesmas. Sendo assim, é necessário perceber que ferramentas existem ao nosso dispor para o devido efeito.

No caso do sistema operativo iOS, a Apple tem uma ferramenta que vai ao encontro das necessidades deste trabalho, o Instruments [45]. Esta ferramenta foi também referenciada em diversos estudos como solução para o levantamento de consumos por parte das aplicações na plataforma iOS (e.g., [4, 5]). O Instruments é fornecido no pacote do IDE oficial da Apple, o Xcode [46]. No entanto, é totalmente independente, podendo ser usado sem que o próprio Xcode esteja em execução. Apresentada na Figura 4, esta ferramenta inclui um vasto conjunto de perfis de métricas, por padrão, integrados na ferramenta.

Cada perfil pode ser constituído por diversas métricas. Neste caso, os perfis por padrão da ferramenta agregam métricas da mesma área. No entanto, há a possibilidade de criar perfis customizáveis com as métricas pretendidas.

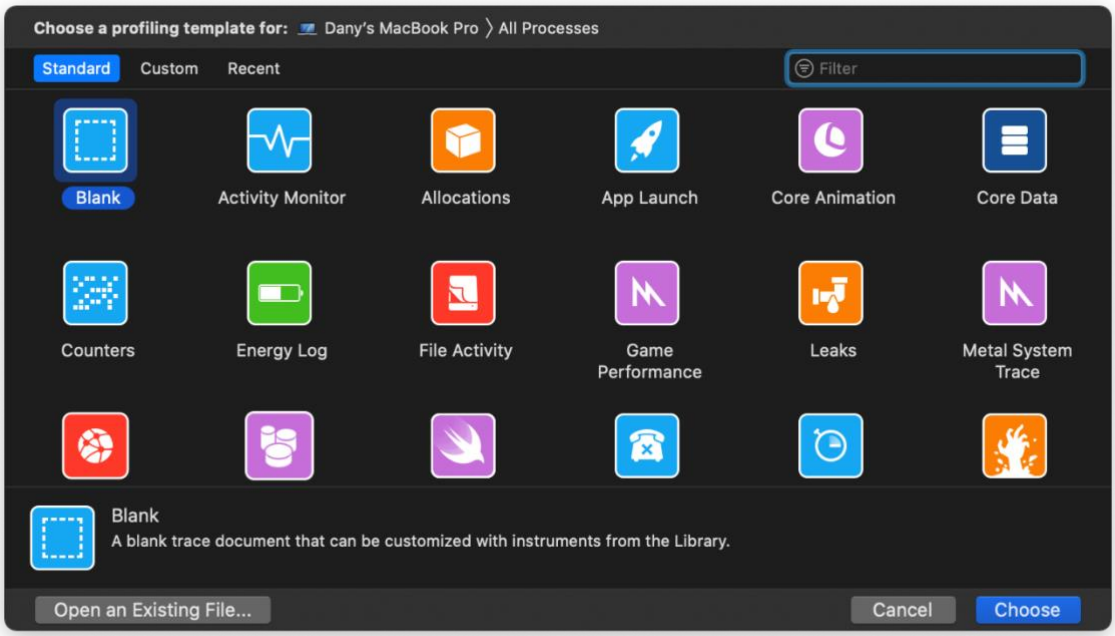


Figura 4 - Perfis padrão disponibilizados pelo Instruments.

Relativamente ao sistema operativo Android, a ferramenta que mais se assemelha ao Instruments e que também vem integrada no IDE oficial da Google para o desenvolvimento Android (o Android Studio) [47], é o Android Profiler. Esta ferramenta, ao contrário do Instruments, é dependente do IDE, e não tem a possibilidade de ser executada sem que o

Android Studio esteja também em execução. Acontece que ambas as ferramentas, Instruments e Android Profiler, não foram estruturadas para a avaliação de aplicações, mas sim, para auxiliar no desenvolvimento das mesmas. Por este motivo, o Android Profiler não tem utilidade no propósito deste trabalho, porque não tem capacidade de analisar aplicações em *release mode*, modo em que uma aplicação deve estar para se obter uma avaliação relativamente próxima da realidade [22, 25, 26].

Para além do Android Profiler, também outras ferramentas foram encontradas nos estudos analisados, como é o caso da Trepn Profiler e da Android Device Monitor. No entanto, estas foram excluídas de imediato de uma possível utilização. No caso da ferramenta Trepn Profiler, por não ser uma ferramenta oficial do sistema operativo Android, e quanto à ferramenta Android Device Monitor, pelo facto de ter sido descontinuada pela própria Google [48].

Com o objetivo de serem utilizadas apenas ferramentas oficiais de cada sistema operativo, de forma a garantir uma maior credibilidade dos resultados, foi necessário recorrer a ferramentas baseadas em linha de comandos, o que aconteceu também em alguns dos estudos analisados [5, 9, 22, 49].

Dos trabalhos anteriores analisados foram ainda levantadas as seguintes ferramentas: Systrace [50] e Dumpsys gtxinfo [51], usados para a recolha de informações relativamente ao processamento gráfico; Dumpsys meminfo [52] utilizado para a recolha relativamente à memória consumida; Logcat [53] para a medição de tempos de execução; ADB (Android Debug Bridge) bugreport [54] para a monotorização dos consumos energéticos; e, por último, o ADB Top [55] usado para consulta dos consumos de CPU.

Das opções existentes, a ferramenta Dumpsys meminfo foi excluída tendo em conta que a sua função é a recolha de informação relativamente às alocações de memória no dispositivo, contudo tem a limitação de fazer apenas um *snapshot*, isto é, um registo momentâneo [52]. À semelhança do Dumpsys meminfo, a ferramenta Dumpsys gfxinfo apenas fornece informações relativamente aos últimos 120 *frames* renderizados pela aplicação. Tendo em conta que através do Systrace temos a possibilidade de obter a mesma informação, mas ao longo de todo o decorrer do teste, também o Dumpsys gfxinfo foi excluído. A ferramenta ADB bugreport foi excluída porque não possibilita a recolha de informações relativamente a apenas uma aplicação em questão. Finalmente, a ferramenta Logcat foi excluída porque o

processo de consulta dos tempos de execução foi feito através de componentes das próprias aplicações. Excluídas estas ferramentas, restaram assim a Systrace e a ADB Top.

A Systrace é uma ferramenta utilizada para a recolha de informações relativamente ao consumo gráfico de cada *framework*, mais propriamente de *Janky frames*. Esta não possibilita de forma viável a recolha da quantidade exata de FPS. Assim sendo, será recolhida a quantidade relativamente aos *Janky frames*, sendo também uma informação bastante relevante para a comparação da *performance* de cada *framework* no que diz respeito à capacidade de renderização de conteúdo.

Uma *frame* é classificada como *Janky frame* quando esta ultrapassa o tempo de 16,666ms (milissegundos) para ser renderizada. Foi delimitada a marca dos 16,666ms devido à importância de apresentar uma aplicação com uma UI fluida e suave e, para isso, a necessidade de se ter uma renderização próxima de, pelo menos, 60 FPS. A marca dos 16,666ms tem como origem a divisão de 1000 ms, o que corresponde a um segundo, por 60. Desta forma são atribuídos até 16,666ms a cada *frame*, para que a aplicação possa ter uma renderização em torno dos 60 FPS [51].

Visto que, através dos comandos das ferramentas ADB e Shell [56], temos acesso à maioria dos comandos dos sistemas Unix mais comuns, isto dá-nos acesso a comandos como é o exemplo do ADB Top, que é muito útil para observar o estado dos processos em execução de um sistema Unix, neste caso aplicado ao Android OS. É através do comando Top e de um *script* criado exclusivamente para este trabalho, abordado mais à frente, que será feita a leitura do estado de consumo da CPU e da memória RAM pela aplicação que esteja a ser avaliada [55].

Depois de perceber as alternativas existentes, as ferramentas selecionadas são as seguintes:

- Apple iOS: Instruments;
- Android OS: Systrace e o comando Top.

Uma nomenclatura que ocorre diversas vezes em que se fala das ferramentas à base de linha de comandos é ADB. É importante falar deste comando porque é este que permite a comunicação de dispositivos Android com outros sistemas, neste caso um computador, e é através deste que as ferramentas que vão ser usadas, conseguem receber informações relativamente às aplicações a testar. Junto com o comando ADB são disponibilizadas

diversas funcionalidades; uma delas é o comando Shell, que permite emitir, para o sistema Android, a maioria dos comandos dos sistemas Unix mais comuns [56].

2.6. Seleção de *funcionalidades*

De modo a fazer uma avaliação mais abrangente e não de um só determinado tipo de aplicação móvel, foi feita uma pesquisa das aplicações móveis mais utilizadas em 2019, para perceber quais as funcionalidades com maior necessidade de serem avaliadas. A pesquisa foi realizada no dia 24/02/2020 no motor de busca Google, com as seguintes palavras-chave "*most used apps* 2019", filtradas pelos resultados do último mês. Como resultado desta pesquisa foram consultados diversos sites, onde foi encontrada uma fonte comum a diversos, a APP ANNIE [57].

De acordo com essa fonte e com a Figura 5, as aplicações mais descarregadas durante o ano de 2019 foram:

1. Facebook Messenger;
2. Facebook;
3. WhatsApp Messenger;
4. TikTok;
5. Instagram.

Figura 5 - Top 10 das aplicações e jogos mais descarregados em 2019 (Fonte: [57]**)**

Com o resultado desta pesquisa foi possível definir cinco aplicações modelo, com o objetivo de perceber quais as funcionalidades de software mais comuns que podem ser alvo para testes. Depois das aplicações modelo estarem selecionadas, foi iniciado um processo de análise das mesmas, para ir ao encontro das funcionalidades mais frequentes que estas continham.

Após a análise das diversas aplicações, foram selecionadas nove funcionalidades para serem reproduzidas e testadas, nomeadamente:

- *Launch* – sendo este o primeiro contacto de um utilizador com a aplicação, foi decidido avaliar esta funcionalidade;
- Listas – esta funcionalidade foi selecionada tendo em conta que a maior parte do conteúdo em qualquer das aplicações modelo é apresentado em forma de listas. Das listas originam-se duas subfuncionalidades distintas: uma direcionada para listas de conteúdo remoto e outra para listas de conteúdo local;
- Câmara – tendo em conta que todas as aplicações modelo são aplicações de variação social e comunicação, todas possibilitam a captura de imagens, tanto fotografias como vídeos. Deste modo, temos duas funcionalidades relacionadas, a captura de imagens e de vídeos;
- Acesso a conteúdo multimédia local – com o mesmo propósito do acesso à câmara, os utilizadores das aplicações modelo podem optar por partilhar conteúdo multimédia que tenham já adquirido anteriormente. Desta forma, estamos perante mais duas funcionalidades: acesso a imagens e vídeos nos dispositivos;
- Acesso a conteúdo multimédia remoto – os utilizadores das aplicações modelo estão constantemente a consumir conteúdos multimédia remotos. Assim, resulta mais uma *funcionalidade*, reprodução de conteúdo multimédia remoto;
- Animações – todas as aplicações modelo apresentam diversas animações, seja na navegação ou na interação de conteúdo. As animações estão presentes com o objetivo de tornar a utilização da aplicação uma melhor experiência. Desta forma, a renderização de animações será também uma funcionalidade a testar.

Na escolha destas funcionalidades foi tido em conta também as funcionalidades testadas nos diversos estudos analisados. No entanto, seguindo a mesma filtragem efetuada na lista de métricas, a única funcionalidade que ocorre mais que 5 vezes são as listas, com um total de 10 ocorrências.

Depois de todo este processo de análise dos estudos prévios e de todas condições em que os mesmos foram realizados, foi percetível que não existe apenas uma necessidade de comparação entre tecnologias, como também a necessidade de chegar a um consenso quanto a um método completo, estável e credível para a realização das comparações entre as tecnologias. O objetivo deste trabalho assenta nessa mesma necessidade, não se limitando a comparar apenas duas tecnologias, onde uma nova *release* de uma destas pode ser o suficiente para colocar o resultado do estudo em causa, mas procurando criar um processo de comparação, que possa servir de orientação a qualquer entidade que pretenda perceber qual a melhor alternativa tecnológica para os seus projetos.

3. Metodologia

No presente capítulo, pretende-se dar a conhecer a metodologia de desenvolvimento adotada e as ferramentas de apoio à gestão do trabalho utilizadas na realização do mesmo.

Tendo em conta que este trabalho consistiu num estudo de múltiplas tecnologias recentes, foram adotados e adaptados alguns dos processos da metodologia ágil Scrum [58, 59].

Durante todo o decorrer deste estudo, existiram inúmeros momentos de reflexão e discussão relativamente às múltiplas decisões que tiveram de ser tomadas. As atividades a exercer foram sendo planeadas e detalhadas com recurso a quadros de atividades, também conhecidos como quadros Kanban, por forma a auxiliar na gestão do tempo disponibilizado e na organização das atividades a executar [60, 61].

Inicialmente foram identificadas múltiplas etapas que este trabalho teria de percorrer para que fosse concluído com sucesso (Figura 6).

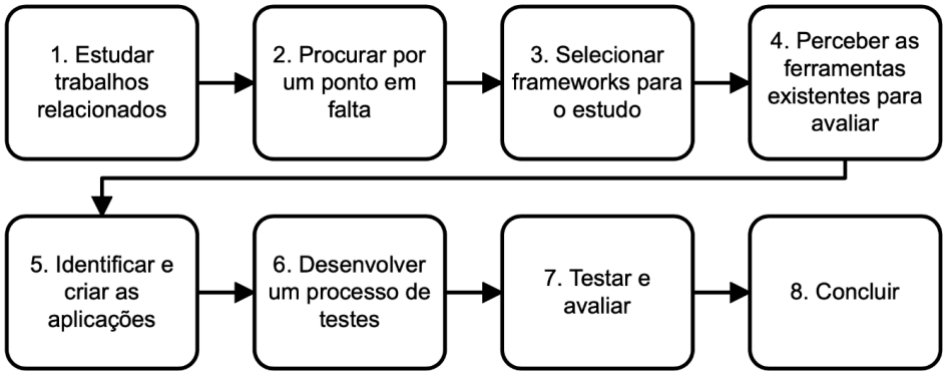


Figura 6 - Etapas iniciais do trabalho

Primeiramente, foi iniciado um estudo para perceber que trabalhos relacionados já existiam e como estes poderiam servir de exemplo para a criação de algo que viesse preencher uma verdadeira necessidade. Identificada a necessidade existente, foram avaliadas múltiplas *frameworks* para serem usadas no decorrer deste estudo. Com as *frameworks* selecionadas, antes de proceder, era importante perceber quais as possibilidades para realizar a avaliação das mesmas. Assim sendo, foram estudadas as ferramentas disponíveis para o devido efeito. Prosseguiu-se então para uma parte mais prática, a identificação e implementação de funcionalidades que pudessem ser uma mais valia na comparação e avaliação das *frameworks*. A projeção e desenvolvimento do processo de testes foi um dos pontos mais importantes, tendo em conta que foi onde a maioria dos estudos analisados apresentaram

falhas. Por fim, prosseguiu-se com a realização dos testes em concreto, a sua avaliação, discussão de resultados e conclusão.

3.1. Trabalho relacionado

De forma a identificar possíveis lacunas nos estudos já realizados que pudessem tornar-se objetivos a alcançar neste trabalho, foram analisados diversos estudos existentes na área da avaliação e comparação de aplicações móveis.

Durante a análise dos estudos, procurou-se perceber os múltiplos pontos que eram abordados ou não por estes, incluindo as métricas usadas, as funcionalidades consideradas para a avaliação, as ferramentas com que o faziam e em que tecnologias eram implementadas as aplicações de exemplo a serem testadas.

3.2. Pontos em falta

Numa pesquisa paralela, referente às ferramentas que vinham sendo referenciadas, mais precisamente ao Instruments, foi mencionado numa palestra oficial da Apple, o termo *Release Mode* [26]. Nos documentos estudados até àquele momento, este termo nunca tinha sido referido e mesmo depois, no decorrer de todo o processo de estudo dos outros trabalhos, as menções a este termo foram muito raras. Este ponto, em junção a outros como o caso da incoerência de ferramentas e métricas que eram usadas, levaram a tornar como objetivo a criação de um processo consistente e fidedigno para a comparação da *performance* de aplicações móveis.

3.3. Seleção de *frameworks*

Tendo em conta que o objetivo definido envolve a comparação de aplicações desenvolvidas por tecnologias de desenvolvimento multiplataforma, procurou-se também comparar as *frameworks* multiplataforma que mais aderência e preferência colhem por entre a comunidade na atualidade. Para isso foram analisados os repositórios oficiais de cada *framework* e comparados os valores de *Watchs*, *Stars* e *Forks*, quanto maior forem estes valores maior é a sua preferência na comunidade.

3.4. Métricas e viabilidade das ferramentas de medição

Devido à existência de uma enorme diversidade de métricas e ferramentas para aferição de desempenho encontradas em estudos anteriores, foi necessário fazer uma filtragem. Primeiramente, relativamente às métricas foram selecionadas as mais encontradas nos estudos, mais propriamente, as que ocorriam mais de 5 vezes. Nas ferramentas, foi definido que apenas seriam utilizadas ferramentas oficiais das plataformas/sistemas operativos (Android OS e iOS). Depois, foi necessário que essas ferramentas fossem compatíveis com o modo de *release*. Estes critérios foram suficientes para reduzir o número de ferramentas disponibilizadas. Contudo, foi ainda necessário a complementação das ferramentas no sistema operativo Android através de *scripts*, devido a limitações que as ferramentas apresentavam.

3.5. Funcionalidades a desenvolver para aplicação de testes

Depois de todos os pontos anteriores definidos, faltava definir as funcionalidades de uma aplicação de testes a desenvolver para cada uma das *frameworks* selecionadas para o estudo comparativo. De forma a definir as funcionalidades a implementar, foram procuradas as aplicações mais populares e estudadas algumas das principais funcionalidades que estas incorporavam. Assim, foram selecionadas seis funcionalidades do Top cinco das aplicações mais populares. Na implementação destas funcionalidades, as aplicações foram produzidas de forma a terem a maior semelhança possível entre as *frameworks*, sem entrar em detalhes pormenorizados e respeitando também alguns dos comportamentos nativos de cada uma.

3.6. Configuração e desenvolvimento dos testes

A configuração e desenvolvimento do processo de testes exigiu muito trabalho e rigor. Tendo em conta que foi deste processo em que foram obtidos todos os valores posteriormente analisados e traduzidos em conclusões, foi necessário garantir que tudo estava bem definido e estável. Neste processo, um dos pontos cruciais é o modo em que as aplicações devem estar para serem devidamente testadas, neste caso o *release mode*. Depois de garantir o modo de compilação das aplicações, foi preparado o ambiente de testes. Neste, foi necessário escolher dispositivos específicos de cada sistema operativo e garantir que estes não seriam obstruídos por aplicações alheias ou outras comunicações, durante os testes.

3.7. Execução de testes e avaliação

No decorrer da execução dos testes, por forma a que toda a avaliação fosse o mais confiável possível, foram definidos dois procedimentos importantes.

O primeiro procedimento diz respeito à forma como o teste é realizado. Cada teste é constituído por um conjunto de passos importantes para a obtenção de resultados credíveis. Cada teste é iniciado pela desinstalação/remoção da aplicação (caso exista ainda no dispositivo móvel/*smartphone*); posteriormente, o dispositivo é reiniciado, para que seja perdida qualquer informação relativamente à aplicação na memória RAM. Com o dispositivo reiniciado, a aplicação é compilada e instalada pela própria *framework* no modo *release*, e finalmente procede-se à execução do teste.

O segundo procedimento é referente ao número de repetições de cada teste para cada aplicação. Esta questão surgiu do facto de que podiam surgir valores que não correspondessem à realidade. Assim, foi definido que cada teste seria executado 10 vezes, exatamente o mesmo número que a Apple utiliza nos testes escritos automatizados.

Após execução dos testes, foram usadas tabelas com colorações, de forma a proporcionar uma análise mais intuitiva e facilitada. A avaliação é feita por cada aplicação que é criada dentro de cada funcionalidade selecionada, através dos consumos que foram registados em cada sistema operativo e cada tecnologia. Neste processo, também são atribuídos pontos a cada tecnologia, conforme as métricas em que estas têm melhores resultados.

3.8. Conclusão

De forma a concluir o estudo comparativo entre as diferentes tecnologias de desenvolvimento de aplicações multiplataforma, foram utilizados os pontos que no decorrer da avaliação foram atribuídos a cada tecnologia. Com o recurso destes pontos e um sistema de alavancagem resultante da divisão entre os valores registados de cada *framework*, foi obtida uma análise menos pormenorizada, mas que reflete igualmente a realidade. Deste modo, foi possível efetuar uma comparação mais evidente e acessível.

Na conclusão dos resultados, estes foram também apresentados em tabelas coloridas, tanto da perspetiva das funcionalidades, como das métricas. Estas duas perspetivas produziram resultados diversificados também no que diz respeito ao sistema operativo e à *framework*.

4. Aplicações e casos de teste

No presente capítulo irão ser abordadas as aplicações criadas (uma por cada frameworks a comparar), como resposta às funcionalidades identificadas anteriormente, a serem alvo de testes e avaliação. Neste capítulo, também é abordado o procedimento em que cada aplicação será testada.

Tratando-se de aplicações que foram criadas com o objetivo de simularem funcionalidades reais, a maior parte destas necessitaria da interação do utilizador para fazerem uso do seu propósito. Acontece que, no processo de testes e avaliação, esta interação poderia colocar em risco a imparcialidade dos resultados. Deste modo, foi procurada uma solução que fosse compatível com ambos os sistemas operativos Android OS e iOS, de forma a tornar o processo de testes isento de interações humanas.

Depois da análise de diversas ferramentas de automatização, concluiu-se que estas apresentavam diversas limitações, como por exemplo, não suportarem a realização de testes em aplicações no modo *release*, ou terem a necessidade de serem instaladas aplicações adicionais nos dispositivos móveis, sendo desconhecida qual a interferência que teriam nos resultados. Para além disto, havia ainda outro fator que impedia a utilização dessas ferramentas de automatização: o facto de necessitarem de dar início às aplicações e este procedimento ir contra o processo de execução da ferramenta Instruments. Isto porque, para este realizar a avaliação de uma aplicação, tem de ser a própria a dar início à aplicação em teste, assim como as ferramentas de automatização. Sem a possibilidade de implementar testes automatizados, optou-se por criar processos de automatização integrados diretamente nas aplicações, modelando alguns dos testes e reduzindo a interação humana sempre que possível.

O processo de testes foi o mesmo para ambas as aplicações desenvolvidas para Android OS e iOS. Não obstante, em determinados casos, por limitações das ferramentas e/ou de algumas tecnologias como mencionado anteriormente, alguns testes foram adaptados a cada plataforma, respeitando a imparcialidade entre as diferentes plataformas e *frameworks* para que, posteriormente, também as avaliações fossem confiáveis.

Na tentativa de percecionar possíveis picos de consumo distintos nas plataformas, determinadas aplicações foram estrategicamente criadas com compassos de espera, para que a identificação destes picos fosse facilitada numa posterior análise.

Tendo em conta que estão a ser estudadas *frameworks* que se dedicam à criação de uma aplicação para dois sistemas operativos distintos, de cada *funcionalidade* identificada e estudada, nascem quatro aplicações distintas, tal como representado na Figura 7.

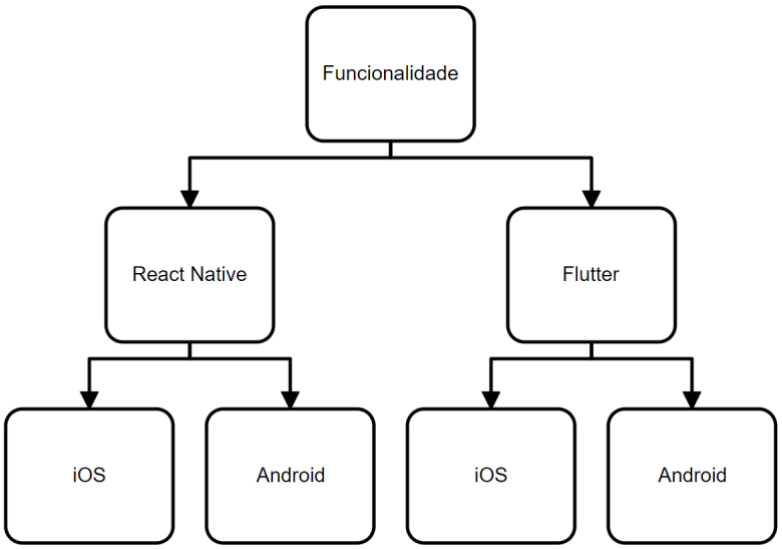


Figura 7 - Diagrama demonstrativo da origem das aplicações

Sendo que estão a ser estudadas nove funcionalidades distintas e que, em algumas destas, são criadas diversas variações das mesmas, obtém-se um total de 52 aplicações, sendo 26 direcionadas para o sistema operativo iOS e as outras tantas para o sistema operativo Android (ver Apêndice B).

4.1. Funcionalidade de *Launch*

O tempo de carregamento de qualquer aplicação afeta a experiência do utilizador, assim como a sua satisfação em utilizar a mesma [62]. Por esta mesma razão, é importante considerar este ponto no que toca também a aplicações móveis. Desta forma, foi criada uma aplicação que simplesmente inicia e apresenta uma curta mensagem, como ilustrado na Figura 8.

Figura 8 - Aplicação *Launch* em iOS para ambas as *frameworks* React Native (esquerda) e Flutter (direita).

Como é percetível pelo diagrama apresentado na Figura 9, o processo de execução desta aplicação é relativamente pequeno, sendo a funcionalidade apenas a apresentação de uma mensagem, uma vez que o desenvolvimento desta aplicação teve como objetivo perceber quais os recursos que são consumidos no arranque de uma aplicação.

Figura 9 - Diagrama do fluxo de funcionamento da aplicação *Launch*

Tendo em conta que não existe uma forma relativamente fácil e funcional de monitorizar o tempo de execução desde que a aplicação é iniciada até apresentar o conteúdo pretendido, a métrica de execução temporal, no caso desta aplicação, foi feita de forma distinta. Neste caso, não foram usados os marcadores de tempo da própria aplicação e calculadas as diferenças. O tempo de execução foi calculado com recurso a componentes visuais que o Instruments (iOS) e o Systrace (Android OS) criam relativamente a consumos da aplicação.

Ainda assim, tal como acontece em outras funcionalidades testadas, a métrica temporal não foi a única a ser avaliada, sendo as outras métricas avaliadas de forma normal; excetuando apenas a métrica de FPS, visto que o tempo de execução é relativamente curto para o registo de valores.

4.2. Listas

Uma das funcionalidades mais presentes na maioria das aplicações móveis são as listas. Existe uma enorme diversidade do que pode ser apresentado em listas, incluindo listas de conteúdo multimédia, mensagens, contactos, entre outros.

De forma a ser possível testar a *performance* de cada *framework* na execução de aplicações que fazem uso de listas, esta foi uma das funcionalidades consideradas e desenvolvidas na aplicação de testes.

Para ser possível uma melhor comparação com o mercado de aplicações, foi estudado o número de elementos que cada aplicação modelo da lista apresentada na secção 2.6 carrega na primeira vez que é iniciada.

O processo de recolha deste número foi o seguinte: instalar as aplicações sem dados nem *cache* no dispositivo e, posteriormente, efetuar *login* nas aplicações. Adicionalmente, tendo em conta que todas utilizam conteúdo remoto, foram interrompidas todas as conexões, de forma a que não fossem carregados mais dados durante o procedimento de contagem. A contagem foi feita nas listas apresentadas na primeira janela que cada aplicação apresenta. No caso das aplicações de *chat* Facebook Messenger e Instagram, também foram contadas as mensagens que estas carregam numa conversação. A aplicação WhatsApp foi excluída, pois acede à *cloud* e restaura todos os dados, não sendo possível fazer a contagem pretendida.

Terminado o processo de contagem dos elementos que são apresentados nas aplicações modelo, concluiu-se que não existe nenhuma quantidade padrão utilizada entre as aplicações. Apenas foi observado que a lista de conversações (*chats*) das aplicações Facebook Messenger e Instagram é inicialmente carregada com as 10 mensagens mais recentes. Não havendo um padrão entre as aplicações, à exceção dos *chats*, foram tomados em conta três valores distintos para a elaboração dos testes de *performance*: 10, 50 e 100.

Os valores 50 e 100 foram selecionados pelo facto de que, desta forma, durante os processos de avaliação também será possível perceber qual a reação das *frameworks* ao lidarem com listas de maior porte.

Na renderização de elementos em lista, por norma, apenas os elementos que estão visíveis são realmente renderizados. Este comportamento é também nativo de ambas as *frameworks* React Native e Flutter em avaliação. Tendo isto em conta, todas as aplicações criadas que envolvem listas foram automaticamente preparadas para que, assim que a lista for apresentada com os primeiros elementos, seja realizado o *scroll* automático na lista até que o último elemento seja apresentado. O objetivo deste procedimento é obrigar a que sejam renderizados todos os elementos das listas em cada aplicação.

Para a funcionalidade de lista, foram criados dois tipos de aplicações: listas locais e listas remotas. Com o desenvolvimento destes dois tipos, é possível testar a *performance* de criar e carregar uma lista com itens locais do próprio dispositivo e carregar uma lista com informação que tem de ser primeiramente acedida remotamente (através, por exemplo, do consumo de *web services*).

Relativamente à avaliação das aplicações, para que esta fosse a mais confiável, a métrica de tempo de execução apenas foi cronometrada desde a ação que dá origem aos dados que serão posteriormente carregados para a lista até ao exato momento em que a lista está para ser renderizada com os mesmos. Isto pelo facto de que, nas diferentes *frameworks*, o processo de automatização do *scroll* nas listas é nativamente realizado de forma distinta.

4.2.1. Listas Locais

Tendo em conta que para a criação de uma lista são necessários itens para a preencher, foi criada uma aplicação que é iniciada sem qualquer registo. Inicialmente, esta apresenta uma caixa de texto para que seja introduzida a quantidade de itens pretendida, possibilitando assim ao utilizador definir o número de elementos que pretende carregar para a lista. Como valor padrão é definido o valor de elementos que esteja para ser testado, evitando assim uma maior interação humana com a aplicação durante o processo de testes. Assim que o utilizador clica para iniciar a lista, são carregados os elementos pretendidos para uma tabela numa base de dados em SQLite [63] localizada no dispositivo, e de seguida carregados para uma vista de lista, como ilustrado na Figura 10.

Figura 10 – Ecrãs das vistas de listas locais das aplicações para iOS, desenvolvidas nas plataformas React Native (esquerda) e Flutter (direita).

Através do diagrama apresentado na Figura 11, é possível observar a linha de desenvolvimento que a aplicação segue e quais as interações feitas pelo utilizador. Nesta aplicação, a avaliação da métrica temporal foi levantada desde que o utilizador dá a instrução para iniciar a lista até que esta é apresentada. As restantes métricas foram recolhidas ao longo de todo o processo ilustrado no diagrama, excluindo apenas o carregamento inicial da aplicação.

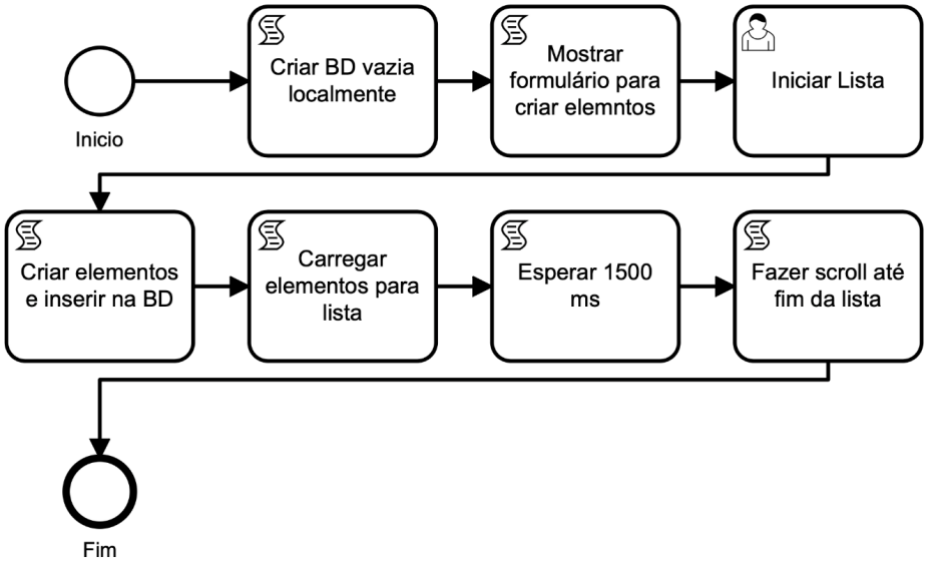


Figura 11 - Diagrama do fluxo de funcionamento das aplicações desenvolvidas para a funcionalidade das Listas Locais.

4.2.2. Listas Remotas

Nas aplicações de listas remotas, todo o seu processo é relativamente idêntico ao das aplicações das listas locais. O que mais as distingue é o aspeto visual (Figura 12), a fonte de informação, o conteúdo e aspeto dos itens. De maneira a criar uma aplicação mais próxima das aplicações modelo, os elementos das listas podem ser de três tipos de conteúdo: texto, imagens ou vídeos, e nas três variações de listas, cada uma tem uma quantidade adaptada. No caso da variação de 10 itens, a lista é constituída por um vídeo, duas imagens e os restantes elementos são de texto. A lista de cinquenta elementos é composta por quatro vídeos, sete imagens e os restantes 39 elementos são de texto. Por fim, a variação de 100 elementos contém 10 vídeos, 13 imagens e os restantes 77 são de texto.

De referenciar que todo o conteúdo foi criado e organizado de forma aleatória e é exatamente o mesmo em todas as aplicações desenvolvidas, tanto para ambas as tecnologias React Native e Flutter, como para ambos os sistemas operativos Android OS e iOS.

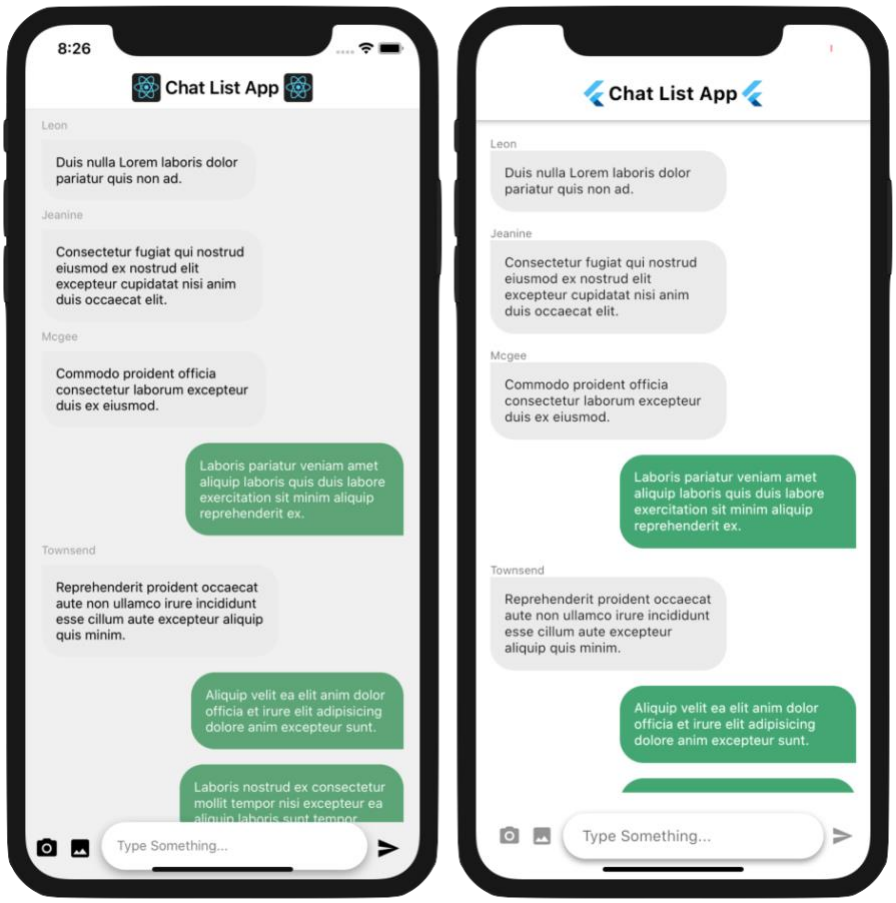


Figura 12 – Ecrãs das vistas das aplicações de Listas Remotas para iOS (esquerda – React Native, e direita – Flutter).

Esta aplicação inicia-se e, após um compasso de espera, faz um pedido de mensagens ao servidor. Posteriormente, essas mensagens são listadas, e é feito o *scroll* automaticamente na lista até que o último elemento seja renderizado (Figura 13). Tal como acontece nas aplicações de listas locais, a medição da métrica temporal é iniciada imediatamente antes do pedido ao servidor ser feito, até que a aplicação receba a resposta, carregue os dados em memória e esteja pronta para os carregar para a lista. Relativamente às restantes métricas, estas são recolhidas depois do processo de inicialização da aplicação ser concluído.

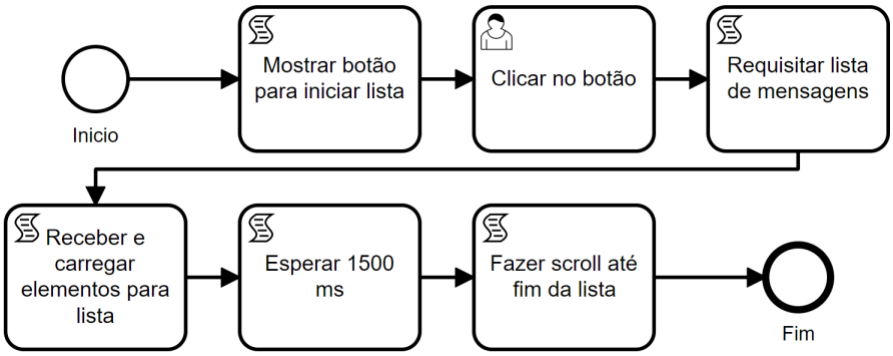


Figura 13 - Diagrama do fluxo de funcionamento da aplicação Listas Remotas

4.2.3. Conteúdos multimédia local

A recolha de conteúdos multimédia é uma funcionalidade muito utilizada nas aplicações modelo apresentadas na secção 2.6. Assim sendo, neste trabalho foram criadas quatro aplicações dedicadas a esta categoria. Duas aplicações são para fazer uso da câmara do dispositivo, uma para captura de fotografias e a outra aplicação para a captura de vídeos. As outras duas aplicações têm o propósito de dar ao utilizador o acesso às fotografias e vídeos já existentes nos dispositivos, sendo que cada aplicação é dedicada a cada tipo de conteúdo (fotografias e vídeos).

Nas aplicações de multimédia, tendo em conta que é necessária uma maior interação do utilizador para a sua execução e não existindo a possibilidade de automatizar o processo, a métrica temporal foi removida por deixar de ser viável. Excluindo a inicialização das aplicações, as restantes métricas foram obtidas normalmente, até que a notificação de que o conteúdo multimédia tenha sido enviado pela aplicação e recebido pelo servidor corretamente fosse apresentada ao utilizador.

4.2.3.1. Câmara

Nesta funcionalidade havia duas formas de criar as aplicações: através do desenvolvimento de aplicações que fazem uso do módulo nativo da câmara de cada dispositivo ou utilizando módulos de cada *framework* para o uso da câmara. Tendo em conta que o objetivo é comparar e avaliar duas *frameworks*, a utilização do módulo nativo dos sistemas operativos foi a opção escolhida e a mais imparcial para efeitos da sua avaliação.

Foram então criados dois grupos de aplicações (duas aplicações em React Native e duas em Flutter – uma por cada sistema operativo): um dos grupos permite que o utilizador faça a captura de imagens (Figura 14), enquanto que o outro permite a captura de vídeos (Figura

15). Tudo isto com recurso ao módulo nativo da câmara de cada dispositivo, possibilitando avaliar quais os recursos que cada *framework* necessita para esta funcionalidade.

Figura 14 – Aplicações de captura de imagens para iOS (esquerda – React Native, e direita – Flutter).

Tal como no caso da aplicação de captura de imagens, na aplicação de captura de vídeos, em metade das iterações dos testes, foi utilizada a câmara traseira dos dispositivos e na outra metade a câmara frontal. Com o mesmo intuito, procurou-se definir múltiplas categorias de conteúdo para ser capturado pelas aplicações. Foram então selecionadas cinco categorias nas quais os consumidores gastaram mais dinheiro através de *e-commerce* no ano de 2019 [64].

- Viagens;
- Moda e beleza;
- Eletrónica e *media* física;
- Brinquedos;
- Mobília e eletrodomésticos.

Na categoria de viagens, foram capturadas e gravadas imagens de paisagens; na categoria de moda e beleza, imagens de uma palete de maquilhagem; um *smartphone* para a categoria de

eletrónica e *media* física; um peluche para a categoria de brinquedos; e, por fim, uma cadeira para mobília e eletrodomésticos.

Figura 15 - Aplicação de Captura de Vídeos em iOS (esquerda – React Native, e direita – Flutter).

Como é percetível pelo diagrama da Figura 16, depois do utilizador fazer a recolha do respetivo conteúdo de multimédia, este é enviado para um servidor que apenas se dedica a receber os conteúdos. Depois da aplicação móvel enviar o conteúdo, o utilizador recebe um alerta de que a operação foi bem sucedida.

Um aspeto dos testes à aplicação de captura de vídeos que é importante referir é que todos os vídeos capturados tiveram uma duração próxima de 10 segundos, para tornar os testes o mais imparciais possível.

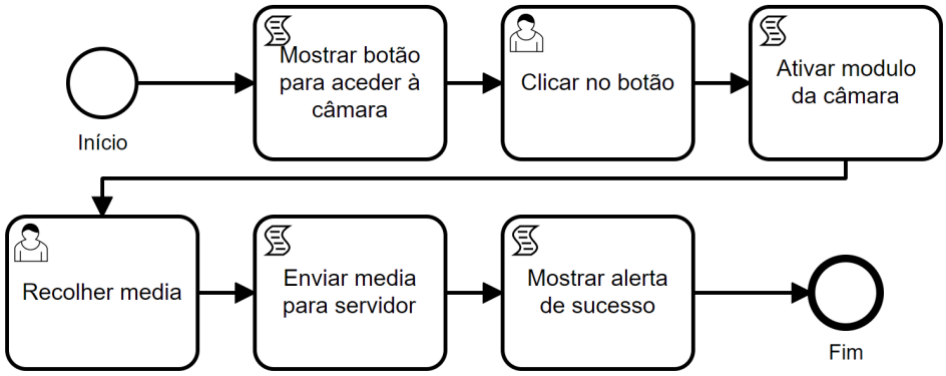


Figura 16 - Diagrama do fluxo de funcionamento das aplicações de captura de conteúdos multimédia

4.2.3.2. Acesso a conteúdo multimédia local

Na partilha de conteúdos multimédia é comum haver a necessidade de aceder a conteúdo já armazenado nos dispositivos dos utilizadores. Deste modo, foram também produzidos dois conjuntos de aplicações para testar o acesso a imagens (Figura 17) e vídeos (Figura 18) localizados nos dispositivos.

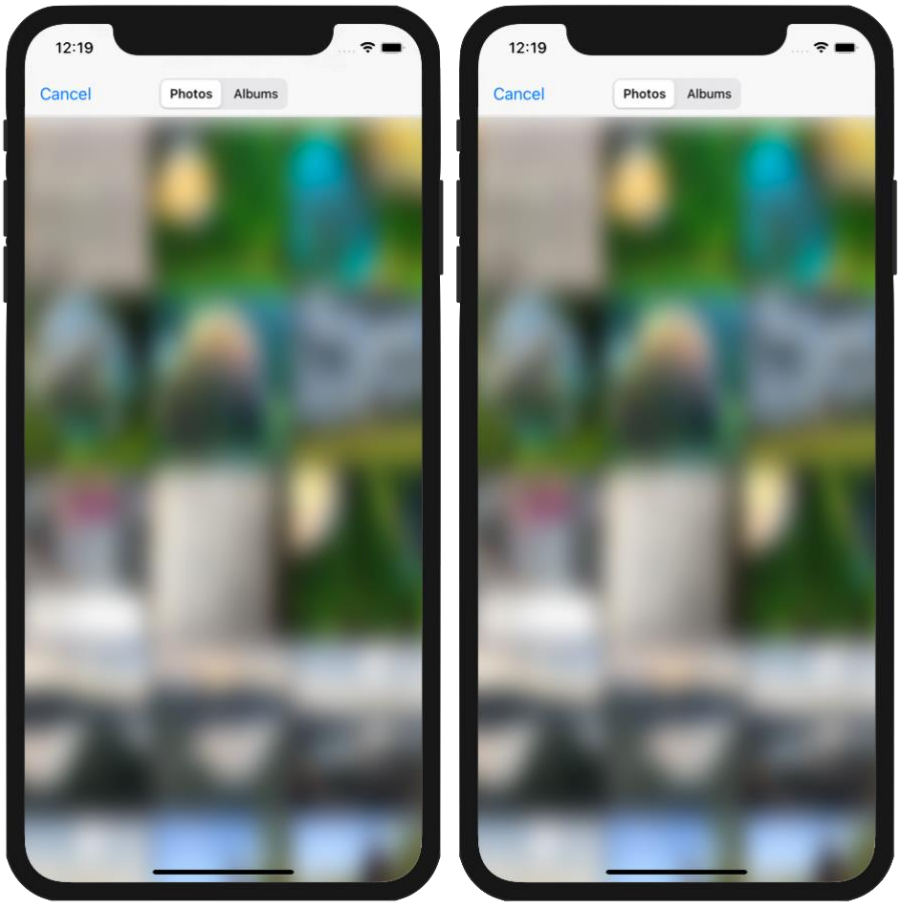


Figura 17 - Aplicação de acesso a imagens em iOS (esquerda – React Native, e direita – Flutter).

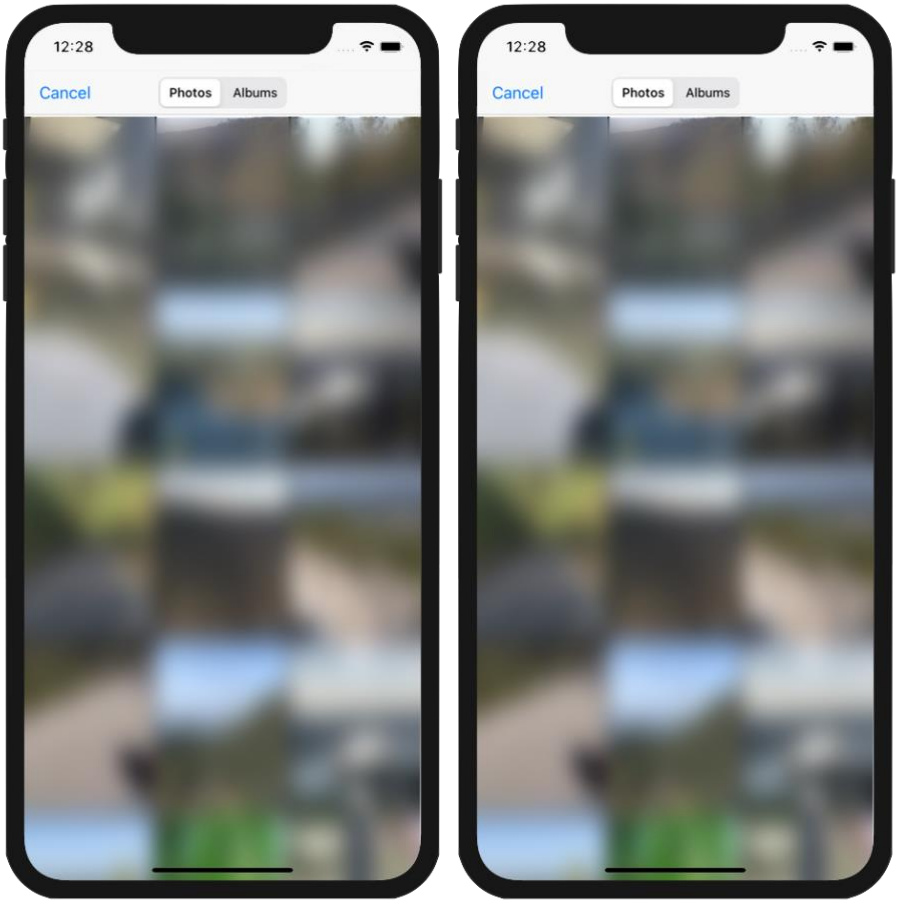


Figura 18 - Aplicação de acesso a vídeos em iOS (esquerda – React Native, e direita – Flutter).

O processo das aplicações desenvolvidas para avaliar o acesso a conteúdos multimédia locais é bastante idêntico às aplicações abordadas no ponto anterior (captura de multimédia), assim como é percetível pelo diagrama da Figura 19. Após o utilizador selecionar o conteúdo pretendido, este também é enviado para o servidor. A avaliação nestas aplicações tem como objetivo avaliar a forma como cada *framework* faz uso do módulo nativo para aceder à galeria em ambos os sistemas operativos e segue exatamente o mesmo procedimento que as aplicações do ponto anterior.

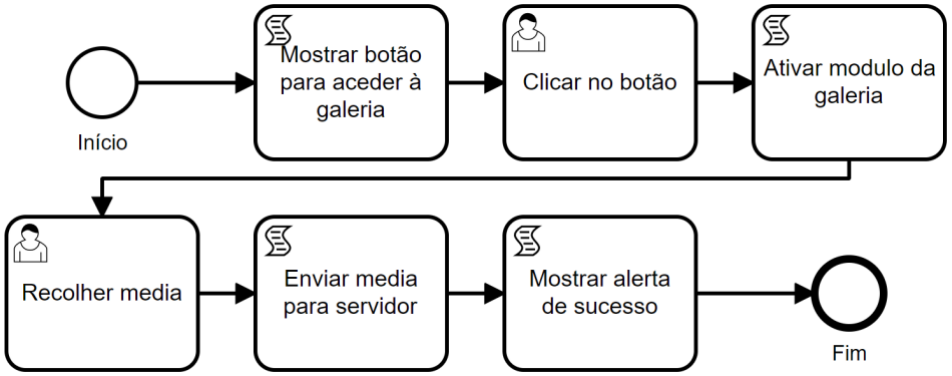


Figura 19 - Diagrama do fluxo de funcionamento das aplicações de acesso a conteúdo multimédia local

4.2.3.3. Acesso a conteúdo multimédia remoto

Assim como os envios de conteúdos multimédia para fontes remotas são ações que as aplicações modelo possibilitam, também o seu consumo é uma das funcionalidades assinaladas. Por consequência, foi criado mais um conjunto de aplicações (React Native e Flutter) que se dedicam a esta funcionalidade. Foi alojado no servidor um vídeo de 15 segundos com 60 FPS, ao qual a aplicação terá de aceder para fazer o consumo do mesmo. A escolha de um vídeo de 60 FPS foi tomada para forçar a aplicação a uma maior carga de processamento na sua renderização. Nesta aplicação, apresentada na Figura 20, tendo em conta que não é pretendido avaliar o carregamento inicial da mesma (pois já é avaliado noutra aplicação), e o tempo de execução restante é dependente do utilizador e do tempo de vídeo, não teria interesse avaliar a métrica temporal, tendo sido excluída. As restantes métricas foram avaliadas desde que o *launch* inicial é concluído até ao fim do processo de *streaming*.

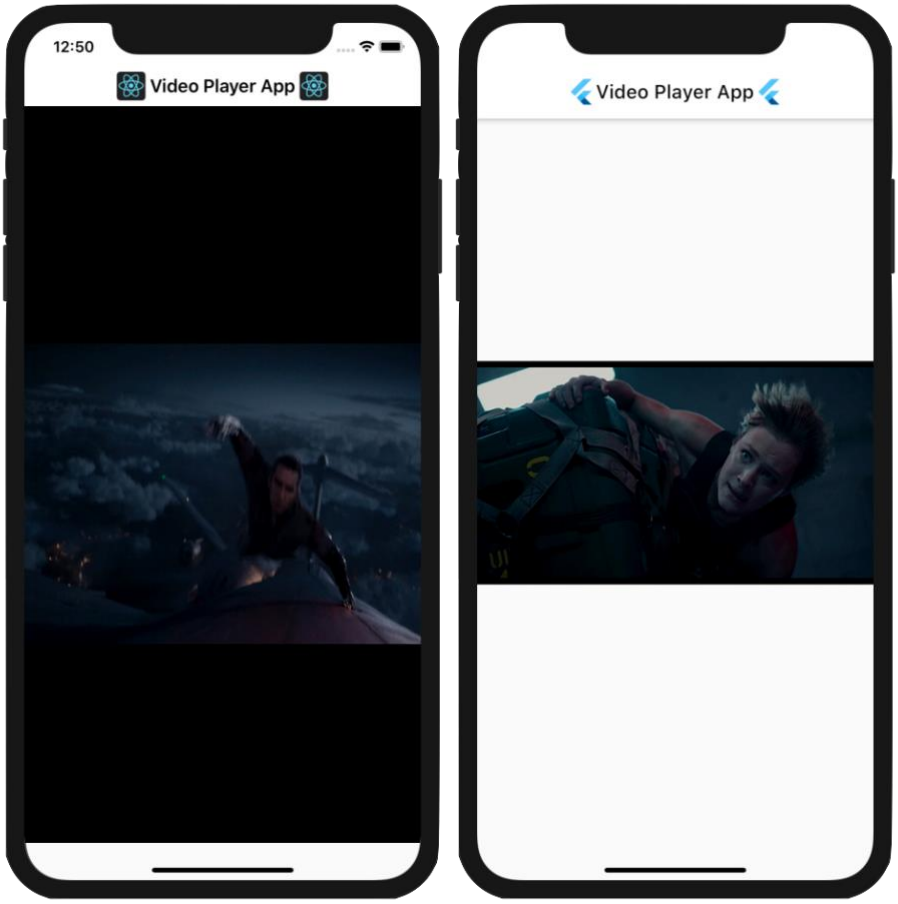


Figura 20 – Aplicação de *Streaming* em iOS (esquerda – React Native, e direita – Flutter)

A linha de desenvolvimento da aplicação é relativamente simples de acompanhar. Através do diagrama da Figura 21 é percetível que a aplicação estabelece conexão com o servidor e reproduz o vídeo, depois do utilizador dar início ao processo de *streaming*.

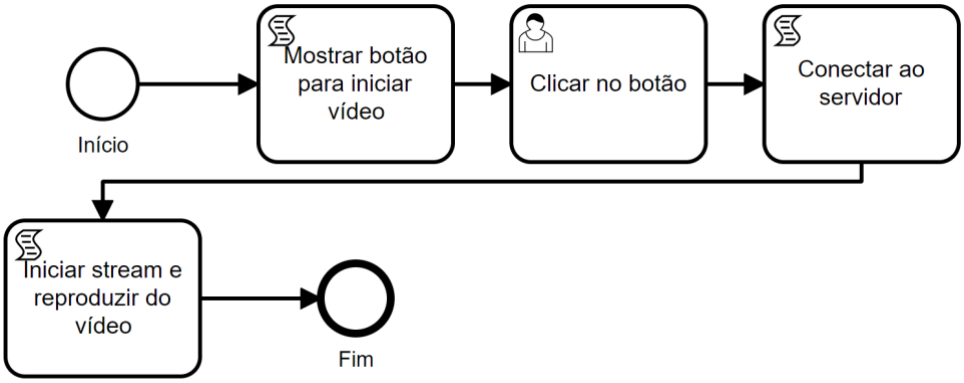


Figura 21 - Diagrama do fluxo de funcionamento da aplicação *streaming*

4.2.3.4. Animações

As animações são efeitos visuais que estão presentes na generalidade das aplicações modelo apresentadas na secção 2.6. Tendo isto em conta, foi criado mais um conjunto de aplicações

para testar o comportamento de cada *framework* no que toca à renderização de conteúdos animados. Uma vez que existem outras funcionalidades onde as aplicações resultantes fazem um maior uso do processamento da CPU, como é o caso das listas locais na criação dos elementos, é importante perceber quais os consumos de processamento gráfico.

Nas aplicações apresentadas na Figura 22 são carregadas vinte imagens distintas, cada uma renderizada 10 vezes com três tipos de animações diferentes: rotação, *fade in* e *scale*. Portanto as animações implementadas em cada uma das 200 posições em que está uma imagem foram aleatoriamente selecionadas através do cálculo do resto da divisão inteira do número da posição por três. Por exemplo para a imagem da posição número 150 o resto da divisão inteira por 3 é zero, o que corresponde portanto à animação “rotação” .

Figura 22 - Aplicação Animações em iOS (esquerda – React Native, e direita – Flutter)

Através do diagrama da Figura 23, é possível acompanhar o processo de execução de cada aplicação. Após iniciar e aguardar 1500 ms, são carregadas as imagens e respetivas animações. Assim que a última imagem for carregada, é iniciada uma cronometragem de três segundos, para que a aplicação renderize todas as animações múltiplas vezes. Terminado

o tempo de renderização, todo o contexto existente é removido e é apresentada uma simples mensagem a informar que as animações terminaram.

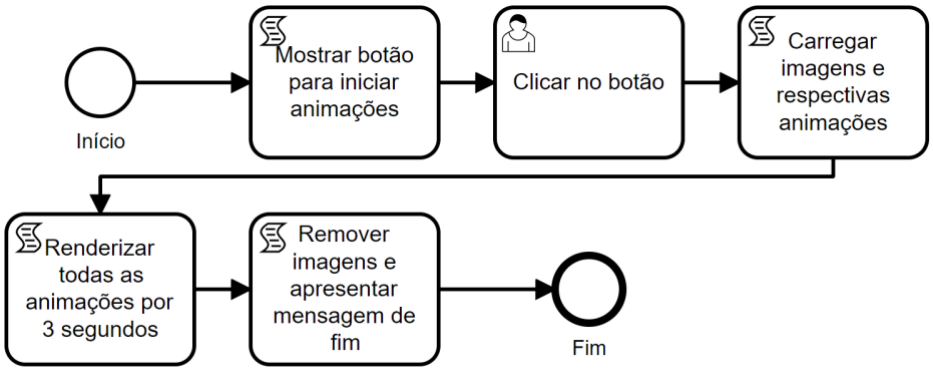


Figura 23 - Diagrama do fluxo de funcionamento da aplicação animações

Na avaliação desta aplicação, a métrica temporal também foi adaptada para cronometrar o tempo desde que o processo para carregar as imagens esteja para ser iniciado até ao ponto em que a última imagem está para ser renderizada. Desta forma, é possível comparar as diferenças de tempo de carregamento de aspetos mais visuais e a sua renderização. Tendo em conta que o resto da aplicação é cronometrada, não teria utilidade prolongar a monotorização do tempo de execução. Relativamente às restantes métricas, são monitorizadas em todo o decorrer da aplicação, excluindo o arranque da mesma.

(Esta página foi deixada em branco propositadamente)

5. Configurações e testes

A avaliação da *performance* de aplicações não consiste apenas na procura da mais rápida a executar um procedimento como, por exemplo, um cálculo matemático. Esta deve ir mais além do que tempo de execução de cálculos. Perceber qual das *frameworks* melhor gere os recursos que as respetivas aplicações consomem durante uma real utilização é um procedimento que pode responder com mais clareza à questão de qual das opções se adequa melhor aos requisitos de determinada aplicação móvel. Com a execução deste trabalho, um aspeto que se revelou bastante complexo foi todo o ambiente necessário para a existência de um processo de avaliação que obtenha resultados relativamente credíveis.

Antes de iniciar o processo de testes e avaliação de aplicações existem determinados aspetos que devem ser tidos em conta. Abaixo resumem-se estes aspetos, alguns deles já abordados em capítulos anteriores:

- Modo em que se deve testar uma aplicação móvel: *release mode*;
- Métricas mais relevantes a considerar: Consumo de CPU, RAM, Tempo de execução e FPS;
- Ferramentas para realizar a recolha das métricas: Instruments, Systrace e o comando Top;
- Funcionalidades a testar: Launch, Listas, Câmara, Conteúdos multimédia local, Conteúdos multimédia remotos e Animações;
- Ambiente de testes;
- Desenho/procedimento dos testes;

No processo de avaliação e de testes às aplicações criadas para este trabalho, houve a preocupação de utilizar as mesmas ferramentas e dispositivos para cada *framework* e sistema operativo. Também é importante referir que, nas aplicações criadas, relativamente ao aspeto visual, existiu sempre a preocupação de desenvolver as aplicações o mais próximas possíveis entre as diferentes tecnologias de desenvolvimento. Contudo, é possível encontrar diferenças devido ao comportamento nativo das *frameworks*, que foi mantido.

5.1. Release Mode

Possivelmente, um dos pontos mais importantes na avaliação da *performance* de uma aplicação é o modo de compilação em que esta se encontra. Qualquer aplicação, para ser

avaliada de forma correta, deve estar em *Release Mode*, caso contrário, a avaliação pode ser influenciada em relação a uma real experiência do utilizador final. No entanto, sempre que seja pretendido comparar múltiplas *frameworks*, como é o caso, ambas as *frameworks* que estão a ser avaliadas devem conter um modo dedicado ao efeito. A importância do modo de *release*, que faz com que seja necessário as aplicações serem avaliadas apenas neste modo, é que quando uma aplicação é compilada para *release mode*, esta é o resultado de uma compilação cujo o objetivo é obter a sua máxima otimização possível, assim como a minimização do seu *footprint* de memória RAM (quantidade de RAM que a aplicação ocupa), sofrendo assim otimizações que, por exemplo, em modo de desenvolvimento, nunca são aplicadas [25, 26].

A *framework* Flutter é um exemplo de uma *framework* que integra um modo para efeitos de avaliação. Este contém o modo de *profile*, que pode ser usado quando é pretendido realizar uma análise de *performance* a uma aplicação [65]. Já a *framework* React Native não integra nenhum modo dedicado a avaliações. É importante que ambas estejam em igualdade, portanto, ambas deverão ser avaliadas em *release mode*.

5.2. Ambiente de testes

Para que os testes sejam o mais fiáveis possível, e para as diversas abordagens de desenvolvimento, é necessário definir e criar um ambiente de testes que se mantenha constante durante toda esta fase. É importante manter o cenário de testes porque, caso contrário, os testes que forem produzidos podem resultar em conclusões enviesadas. Exatamente devido a esta preocupação, foi definido o ambiente de testes descrito abaixo.

Relativamente ao hardware utilizado na fase de desenvolvimento e testes das aplicações, foi tomada a devida atenção de atualizar devidamente com os sistemas operativos mais recentes, oficialmente lançados, logo no início do processo.

No decorrer dos testes, este hardware foi limitado ao nível de comunicações, para que a aplicação em teste não sofresse qualquer interferência externa, de forma a manter o ambiente o mais estável possível. Os dispositivos utilizados no decorrer de todo o processo foram:

- MacBook Pro;
- iPhone 8;
- Samsung S8+.

O MacBook Pro foi a máquina utilizada para desenvolver as aplicações, fazer a recolha dos dados dos testes e onde foi criado um servidor local para responder a aplicações que apresentassem necessidade de conteúdos remotos. As suas principais características e sistema operativo são:

- MacOS: Catalina 10.15.4;
- Processador: 2,2 GHz 6-Core Intel Core i7;
- Memória: 32 GB 2400 MHz DDR4;
- Placa Gráfica: Randeon Pro 555x 4G; Intel UHD Graphics 630 1536 MB.

Em termos de software de servidor a que os dispositivos acedem quando necessitam de conteúdos remotos, utilizou-se o Node.js [66], alojado no MacBook Pro, referenciado acima. A necessidade de criar um servidor local para responder aos pedidos das aplicações surgiu em virtude da minimização da influência de todo o ambiente externo à aplicação, assim como da latência inconstante que uma rede externa pode imputar num teste, desde que é feito o pedido por parte da aplicação até à sua resposta. Neste processo de comunicação, foi também isolada uma rede WiFi 5G dedicada apenas aos dispositivos utilizados, criando assim um ambiente de testes ainda mais isolado de interferências de rede entre o servidor e as aplicações.

O iPhone 8 e o Samsung S8+ foram os *smartphones* utilizados durante o desenvolvimento das diversas aplicações e, posteriormente, na execução dos testes às aplicações. A combinação destes dispositivos não foi ao acaso, pois procurou-se utilizar dois dispositivos próximos na sua geração, para perceber a existência de uma eventual diferença de *performance* entre os próprios sistemas operativos. As características técnicas destes dispositivos estão apresentadas abaixo:

- iPhone8:
    - Modelo: MQ6G2QL/A;
    - Sistema operativo: iOS 13.4.1;
    - Processador: A11 Bionic – Hexa-Core;
    - RAM: 2 GB;
    - Memoria Interna: 64 GB;
    - Câmara Frontal: 12 MP;
    - Câmara Traseira: 7 MP.
- Samsung S8+:

- Modelo: SM-G955F;
- Sistema operativo: Android 9;
- Processador: Snapdragon 835 – Octa-Core;
- RAM: 2 GB;
- Memoria Interna: 64 GB;
- Câmara Frontal: 12 MP;
- Câmara Traseira: 8 MP.

5.3. Procedimento dos testes

O procedimento de testes é tão importante quanto o ambiente dos mesmos, pois o objetivo do procedimento de testes também é garantir que independentemente do teste que esteja a ser realizado e/ou *framework* em avaliação, o seu procedimento deve ser o mais imparcial e isolado possível. Para isso, cada teste a ser realizado passou por uma única sequência de passos. É importante que a sequência seja respeitada para garantir que, entre cada iteração de testes, não seja reutilizada qualquer *cache* ou outro tipo de memória resultante da iteração anterior. O procedimento de testes adotado foi o seguinte:

1. Desinstalar a aplicação, caso exista;
2. Reiniciar o *smartphone*;
3. Correr a aplicação em *release mode*;
4. Realizar o teste.

É importante também garantir que os *smartphones* não têm outro tipo de comunicações externas no decorrer dos testes.

Depois de definido o procedimento de teste, é necessário executar o mesmo teste múltiplas vezes para assegurar a reprodutibilidade dos resultados e verificar que não são fruto do acaso. Assim sendo, é importante definir um número constante, que seja utilizado em todas as aplicações testadas, para que o resultado dos testes seja mais confiável. Para este número, foi definido o valor de 10 iterações, sendo o mesmo número de iterações utilizado pela Apple para os testes de *performance* escritos através do Xcode, como se pode ver exemplificado na Figura 24.

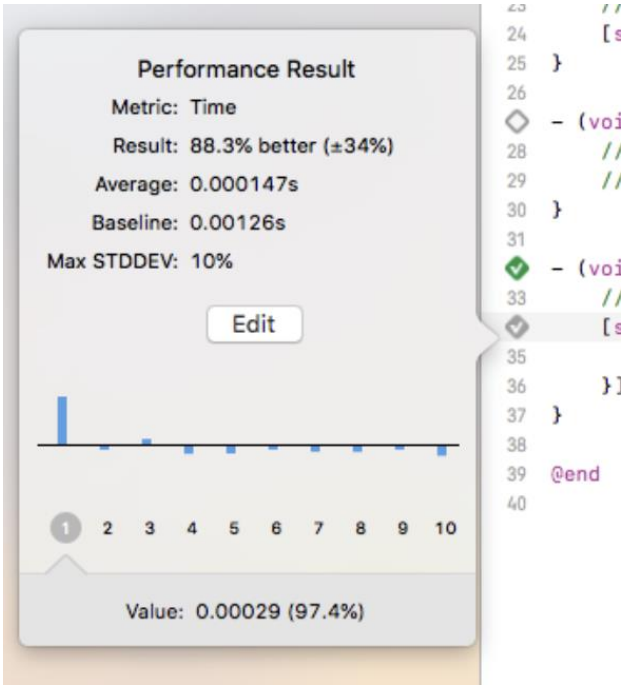


Figura 24 - Demonstração de teste de 10 iterações no Xcode

5.4. Ferramentas e métricas de avaliação

Para a avaliação das diversas funcionalidades produzidas para os diversos sistemas operativos, foram selecionadas várias ferramentas e métricas. Embora as métricas sejam as mesmas nas distintas plataformas, o mesmo não acontece com as ferramentas a utilizar. Isto deve-se à nativa incompatibilidade dos dois sistemas operativos em estudo.

No entanto, no caso da recolha do tempo de execução, na maioria das funcionalidades, seguiu-se o mesmo procedimento em ambas as plataformas. Esta métrica é recolhida através da diferença de marcas temporais, que são calculadas desde o início da funcionalidade até a um ponto em comum de execução das aplicações desenvolvidas em ambas as *frameworks*.

5.4.1. Ferramenta de medição em iOS

No sistema iOS, a ferramenta que mais se adequa aos objetivos pretendidos neste estudo é a Instruments [67]. Para a utilização desta ferramenta apenas é necessário ter instalado o IDE oficial da Apple, o Xcode. Com a ferramenta pronta, procede-se à criação de um perfil de avaliação com as métricas que são pretendidas.

Excluindo os tempos de execução, que são calculados com recurso a tempo, as métricas que temos como objetivo recolher com a Instruments são:

- O consumo de CPU, através do perfil Time Profiler;

- Consumo de memória RAM, onde o perfil responsável pela métrica é o Allocations;
- Número de *frames* renderizados por segundo, sendo o Core Animations FPS o perfil mais indicado para o efeito.

Na Figura 25, é apresentado um exemplo da representação gráfica das três métricas, fornecida pela Instruments.

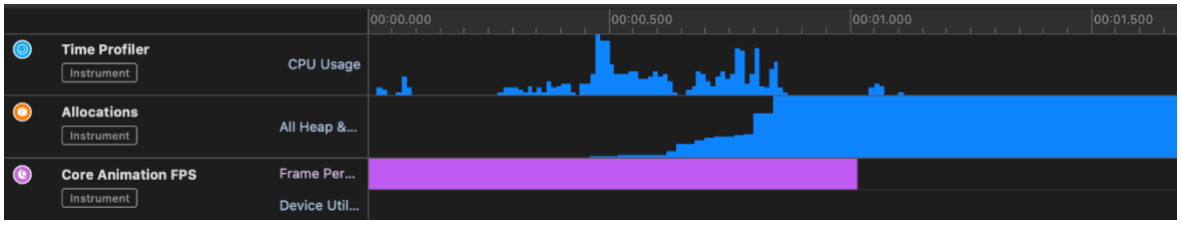


Figura 25 – Representação gráfica da medição de métricas no Instruments

A Instruments não só dá acesso à representação gráfica destas métricas, mas também a informações mais detalhadas como, por exemplo, máximos e médias de consumos. Todavia, no caso do Time Profiler, responsável pela monitorização dos consumos de CPU, as informações adicionais facultadas são relativamente a tempos de consumo de processos e *threads*. Assim, a única informação cuja recolha é exequível é o pico máximo de consumo, através do próprio gráfico.

5.4.2. Ferramentas de medição em Android OS

No Android o Systrace é a ferramenta usada para a recolha de *Janky frames* e o comando Top para recolher valores relativamente a consumos de CPU e RAM. No entanto, a ferramenta Top (comando do *package* ADB) apenas faz registos momentâneos de valores a serem monitorizados. Tendo isto em conta e considerando que todo o processo de execução das ferramentas teria de ser manual, e portanto mais exposto e suscetível a falhas, foram desenvolvidos dois *scripts bash* e um em Node.js, para que os testes pudessem seguir sempre o mesmo procedimento. Estes scripts incluem:

- Um *script* principal, denominado Profile Extraction, que é o responsável por grande parte do processo;
- Um *script* auxiliar, o Profile Top, dedicado à ferramenta Top; e
- Um *script* Profile Transformation, criado para proceder à transformação dos dados através de filtragens e execução de cálculos, assim como a categorização e organização de valores, de forma a possibilitar a criação de gráficos posteriormente.

O *script* principal, Profile Extraction, segue um conjunto de procedimentos que torna o processo de testes mais exato, automatizado e otimizado (Figura 26). No momento que é pretendido realizar a avaliação de uma aplicação Android, o *script* está preparado para apenas ser necessário ser executado na pasta do projeto realizado. Depois de iniciado, o *script* irá procurar pelo *package* da aplicação a testar e o utilizador apenas tem de confirmar se é o *package* correto. Definido o *package*, o utilizador será questionado se pretende incluir o *launch* da aplicação no teste, assim como qual a iteração deste, para o caso de ser uma retoma do teste, e irá pedir também para lhe ser indicada a *framework* a testar.

Com as informações pedidas ao utilizador, o *script* irá prosseguir para a criação de uma estrutura de pastas, que possibilita a identificação de quais as *frameworks* e aplicações que já foram testadas, assim como quais as iterações do teste que foram feitas. Com a estrutura de pastas criada, o *script* irá verificar a existência de dispositivos conectados e proceder automaticamente, colocando o dispositivo em modo de voo, de forma a reduzir possíveis interferências com redes externas. Depois destes passos, será verificada a existência da aplicação no dispositivo e, caso se confirme, a mesma será desinstalada. Após a certificação da inexistência da aplicação no dispositivo, este é reiniciado.

Assim que o utilizador informe que o dispositivo reiniciou corretamente, o *script* procede para a compilação da aplicação em *Release Mode*. Ao compilar as aplicações em *Release Mode*, ambas as *frameworks* iniciam de imediato a aplicação, caso a compilação seja bem-sucedida.

No caso da compilação não ser bem-sucedida, o *script* irá iniciar um conjunto de procedimentos de forma a tentar resolver a situação. Após a compilação ser realizada com sucesso e a mesma ser confirmada pelo utilizador, o *script* fecha automaticamente a aplicação. É neste ponto que o *script* aguarda que o utilizador dê a ordem para iniciar o teste. Assim que o utilizador autoriza a iniciação do teste, as ferramentas e o *script* secundário são iniciados. Caso o utilizador tenha informado que é para excluir o *launch* da aplicação, a mesma é iniciada antes das ferramentas.

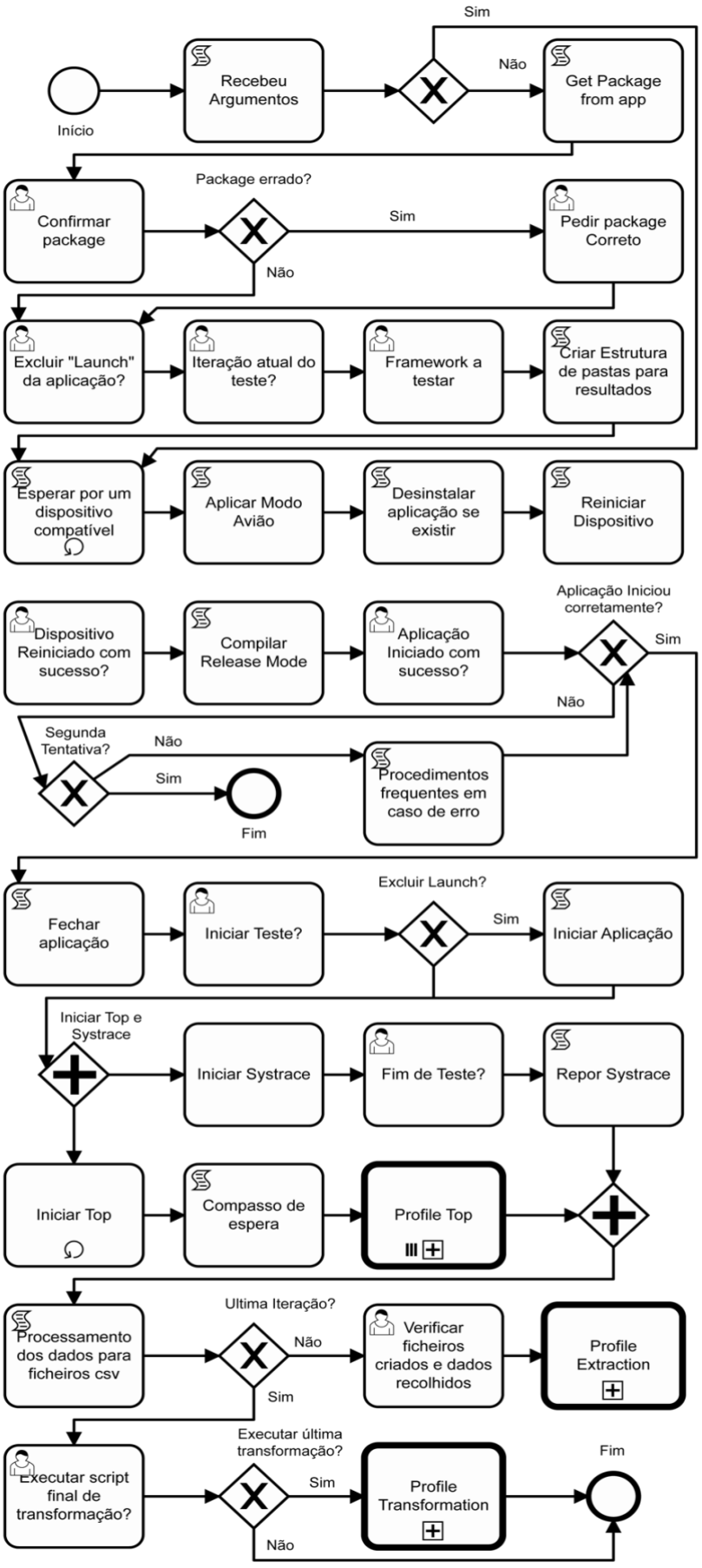


Figura 26 - Diagrama do *script* Profile Extraction

Depois de iniciadas as ferramentas, o utilizador recebe instruções para poder iniciar o teste, caso seja uma aplicação que necessite de alguma interação do mesmo. Assim que o teste é concluído, o *script* irá realizar transformações e conversões de ficheiro para facilitar a sua análise. No fim de todo este processo, caso seja a última iteração, o *script* questiona se pode executar o *script* Profile Transformation de transformação final dos dados e termina. No caso de não ser a última iteração, este irá ser automaticamente executado em recursividade até percorrer as 10 iterações.

O *script* secundário é mais reduzido, e mais simples, tal como se pode observar pelo seu algoritmo geral da Figura 27.

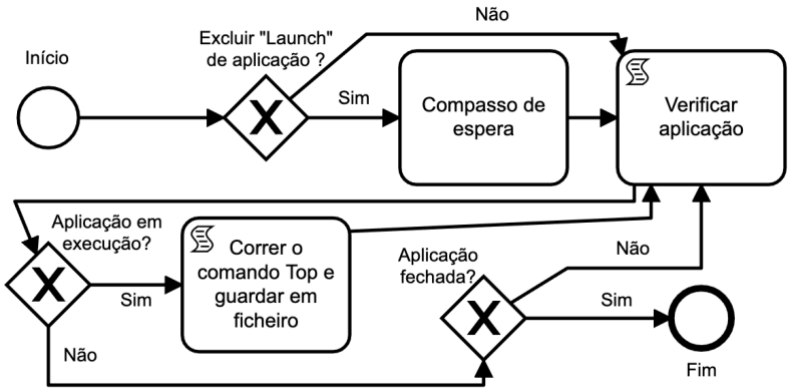


Figura 27 - Diagrama do script Profile Top

Este *script* é executado no momento dos testes às aplicações e o objetivo não é só facilitar a execução do comando Top, mas também automatizar a interrupção do registo, assim que a aplicação for terminada. No entanto, como abordado anteriormente, o comando Top apenas faz registos momentâneos, portanto, ainda que executado em ciclo recursivo, o tempo que leva desde que é executado até que o resultado seja guardado em ficheiro e siga para a próxima escrita é de aproximadamente dois segundos. Tendo em conta que são testadas funcionalidades isoladas, facilmente um teste pode ficar dentro dos 10 segundos, terminando com um valor relativamente baixo de registos. Para contornar esta situação, o *script* principal não só dá início ao *script* secundário, como o faz 10 vezes em processos paralelos, separados por meio segundo, de forma a que possa haver uma recolha mais rápida e dispersa, obtendo assim mais amostras no decorrer de um teste.

Por último, o *script* Profile Transformation (Figura 28) é responsável por percorrer toda a estrutura de pastas criada pelo *script* principal, obter os dados recolhidos e fazer os cálculos

necessários para colocar posteriormente os novos valores por tabelas em ficheiros CSV (*Comma-separated values*), prontos a serem analisados.

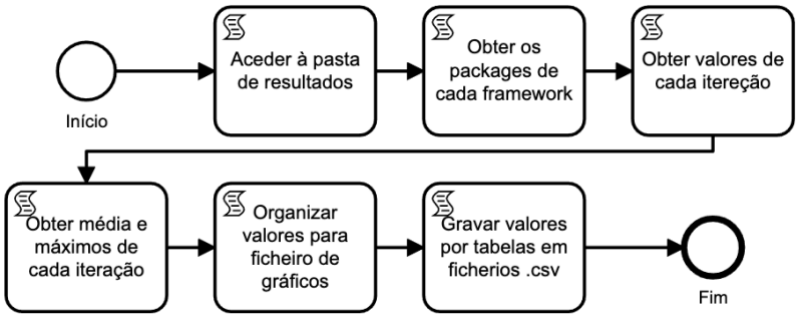


Figura 28 - Diagrama do script Profile Transformation

No processo de recolha dos valores e realização dos cálculos pretendidos, este *script* ainda obtém os dados de forma a serem posteriormente estruturados para a criação de gráficos de auxílio à análise de valores, quando necessário.

6. Resultados e discussão

Neste capítulo serão apresentados e discutidos os testes realizados às diversas aplicações criadas no decorrer deste trabalho, assim como a avaliação dos resultados recolhidos no processo de testes. O objetivo é perceber como foram aplicados os diversos pontos abordados nos capítulos anteriores e que resultados foram obtidos. Na comparação de valores é feita a média das iterações de cada teste e são comparados os números obtidos entre ambas as *frameworks* de desenvolvimento selecionadas neste trabalho: React Native e Flutter.

Devido às limitações das ferramentas, algumas métricas foram avaliadas de forma ligeiramente diferente entre os sistemas operativos iOS e Android. No caso do consumo de CPU no iOS, foram levantados os picos máximos de consumo de cada iteração.

Também para o sistema operativo Android OS o pico máximo é analisado, assim como a média dos consumos. Como acontece no caso do iOS, quando relevante, são usados gráficos oriundos dos registos recolhidos pela ferramenta Top, para auxiliar na análise dos resultados.

No consumo de memória RAM, no caso do iOS, são analisados os valores relativamente ao total de memória consumida durante o teste, enquanto que no sistema Android, os valores recolhidos são analisados relativamente à média dos valores e aos picos máximos dos mesmos. Como referido anteriormente, relativamente à métrica dos FPS, também existe uma diferença entre os sistemas operativos. No iOS é obtida a média e o valor máximo atingido, no Android OS são analisados os valores referentes à quantidade de *Janky frames*.

Em todas as situações, existiu sempre o esforço para obter as mesmas métricas para que, posteriormente, se pudesse também fazer uma comparação de *performance* das aplicações entre ambos os sistemas operativos.

Para cada funcionalidade, depois de comparar as métricas isoladamente, foi feita uma conclusão e avaliação para ambas as *frameworks* React Native e Flutter. De forma a criar uma escala de pontuação, para tornar a avaliação mais intuitiva e de compreensão mais generalizada, cada métrica foi cotada com um ponto, onde a *framework* que apresentasse melhor resultado obtinha esse mesmo valor. Posteriormente, foi feita a comparação entre as *frameworks* com recurso às pontuações que estas conquistaram. Na comparação geral das

pontuações foi feita ainda uma avaliação com base nas diferenças existentes nos valores registados em cada métrica. Estas diferenças foram usadas como valores de alavancagem, de forma a que cada ponto tivesse um peso consoante a vantagem existente em cada registo. Desta forma, a comparação realizada reflete com maior detalhe os valores reais registados.

Os valores obtidos em cada funcionalidade testada e nas várias métricas foram organizados em diversas tabelas, de maneira a que exista uma maior facilidade de observação da *framework* que obtém os melhores resultados, sem ser necessária uma análise detalhada dos valores em concreto. Deste modo, as tabelas são coloridas com verde nos resultados positivos e vermelho nos resultados negativos. Assim, quanto mais valores a verde uma *framework* apresentar, melhores resultados foram obtidos por esta para determinado conjunto funcionalidade/sistema operativo.

6.1. Funcionalidade Launch

6.1.1. Tempo de execução

Como referenciado no Capítulo 4, nas aplicações desenvolvidas para a funcionalidade de Launch, a métrica temporal teve de ser recolhida através de componentes visuais criados pelas ferramentas Instruments (iOS) e Systrace (Android OS). No caso da Instruments foi através de uma representação gráfica do consumo de processador feito pela aplicação que foi feito um levantamento próximo dos tempos de arranque. É possível utilizar esta técnica para obter um valor temporal relativamente ao arranque porque, como se pode verificar pelas Figura 29 e Figura 30, é percetível um padrão de consumos desde que a aplicação é iniciada até ao momento em que a aplicação acaba de ser renderizada na totalidade.

No caso da *framework* React Native, o padrão de consumo de CPU foi bastante conciso, apresentando apenas algumas pequenas variações, como também é possível observar na Figura 29.

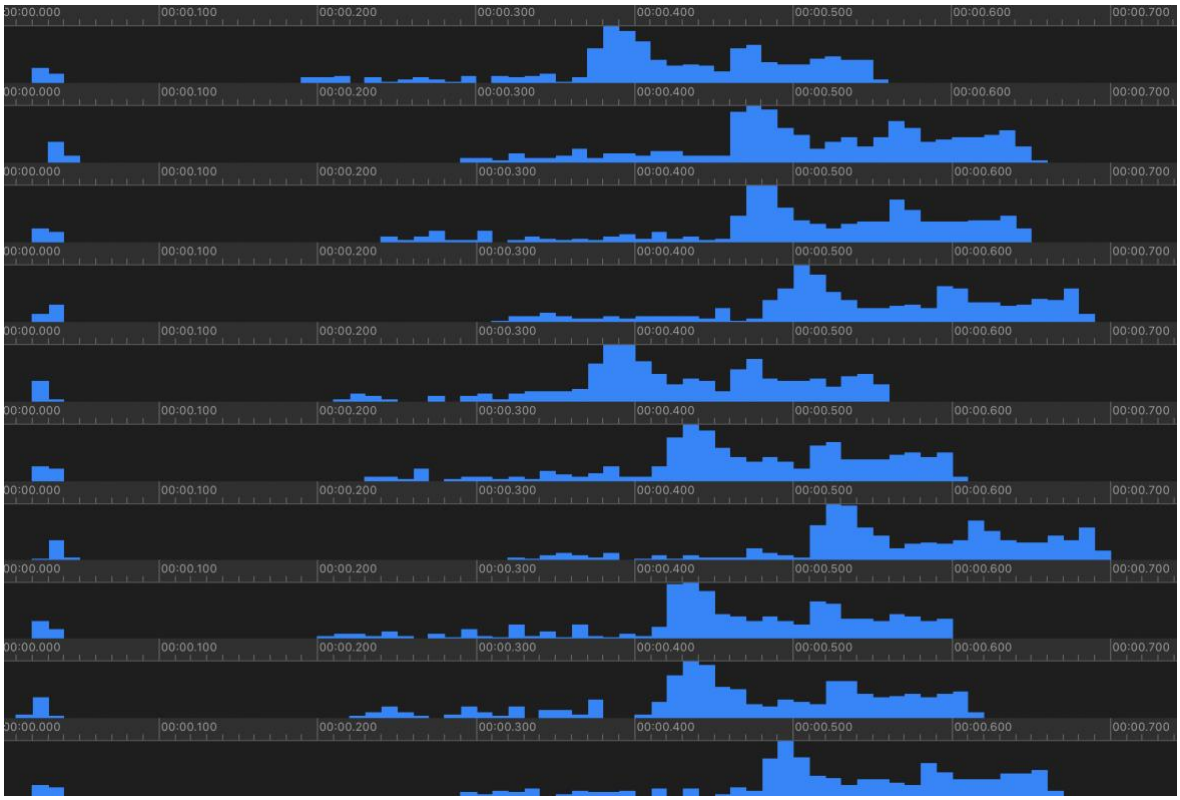


Figura 29 - React Native padrão de consumos na aplicação Launch em iOS

Relativamente à *framework* Flutter, esta também apresentou um padrão visível. No entanto, como se pode visualizar na Figura 30, a nível de consistência verificou-se ser um pouco menos regular.

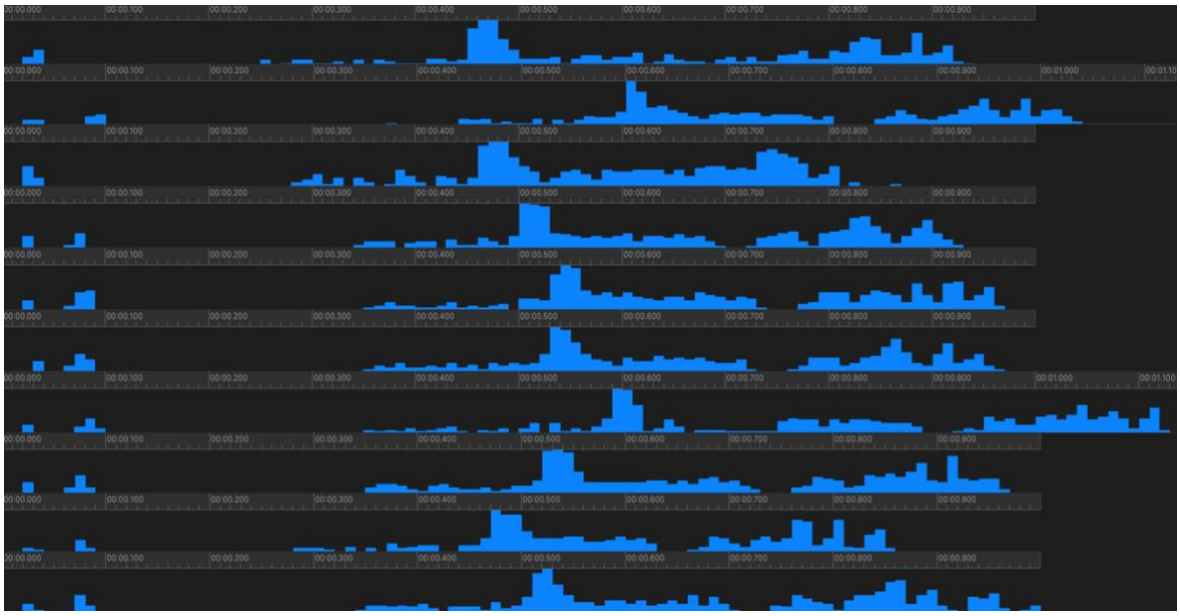


Figura 30 - Flutter padrão de consumos na aplicação Launch em iOS

No caso do Android OS, para a medição do tempo de arranque, não é possível obter o tempo de execução pela mesma métrica (consumo da CPU), no entanto a técnica acaba por ser idêntica. Apesar da ferramenta Systrace ter a capacidade de monitorizar também o consumo de CPU, esta não apresenta os consumos apenas referentes à aplicação em avaliação. Sendo assim também foi usada uma representação gráfica criada pelo Systrace mas é referente à UI *thread*, responsável pela renderização de conteúdo visual. Esta representação acaba por nos indicar também, através de um padrão, quando a aplicação é iniciada até estar totalmente carregada. Distintamente da ferramenta Instruments, que inicia o registo quando a aplicação inicia, a Systrace tem de se iniciar antes da aplicação em teste, para garantir que é registado todo o arranque da mesma. Por este motivo, nas imagens que se seguem, cada iteração tem um tempo de início distinto.

Tanto em React Native como em Flutter, como é possível confirmar pelas Figura 33 e Figura 34, pode-se observar o mesmo padrão de processos de renderização. Estas imagens são constituídas pelas 10 iterações executadas no teste à aplicação de cada *framework*. Através das imagens existem 2 processos onde a diferença entre as *frameworks* é mais fácil de identificar, o processo “activityStart” maioritariamente a cor de rosa (Figura 31), executado na inicialização de uma nova atividade/janela de uma aplicação e o processo “Choreographer#doFrame” a azul e verde (Figura 32), também executado no inicio da renderização de *frames* da nova janela [68].

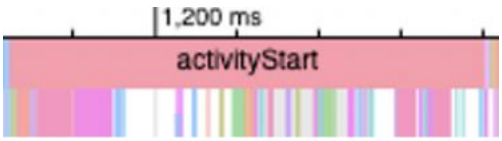


Figura 31 - Exemplo do processo "activityStart"

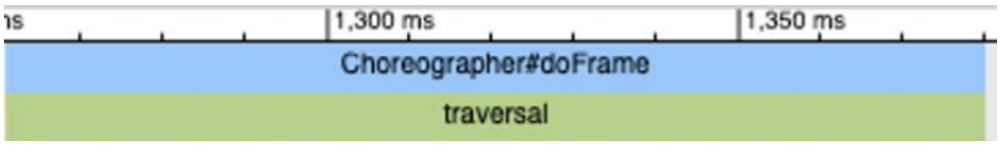


Figura 32 - Exemplo do processo "Choreographer#doFrame"

Ao analisarmos as Figura 33 (Apêndice C) e Figura 34 (Apêndice D) é percetível de que a Flutter acaba por apresentar estes processos mais perlongados temporalmente (sentido horizontal), acabando assim por consumir mais tempo na inicialização da aplicação.

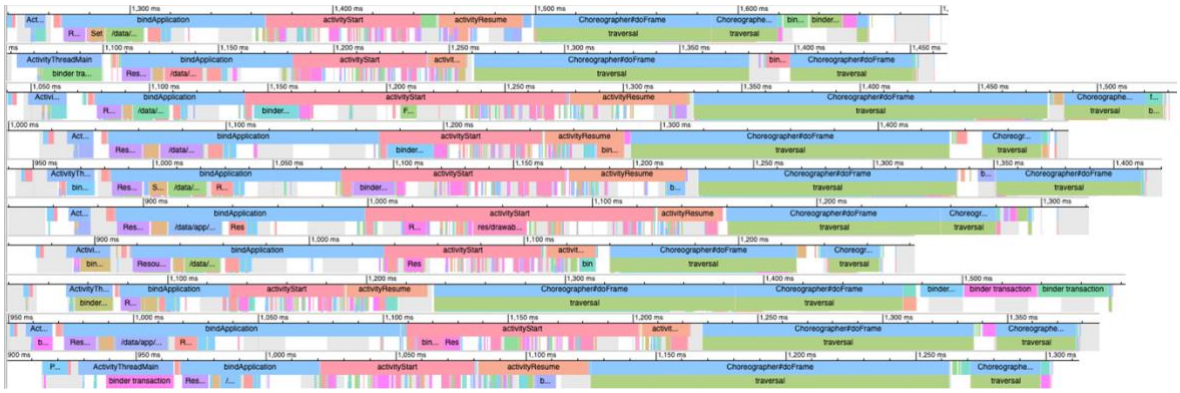


Figura 33 - React Native – As 10 iterações do teste de consumos gráficos na aplicação Launch em Android

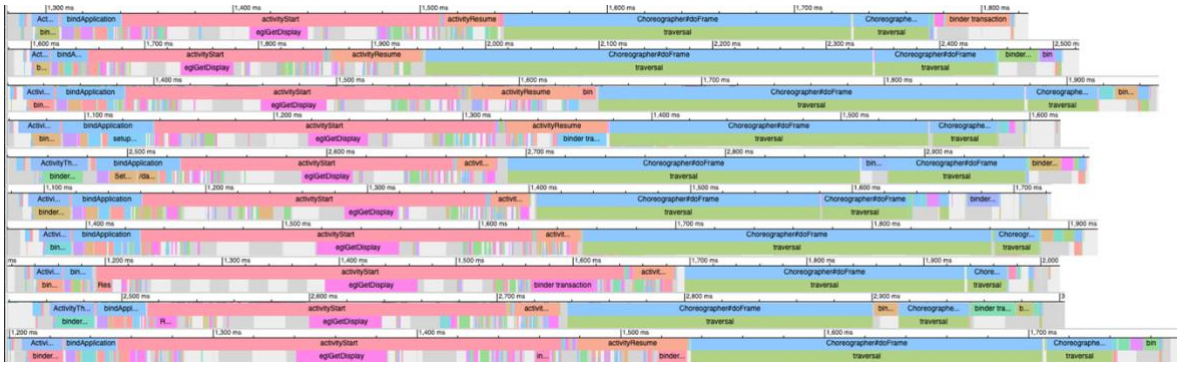


Figura 34 – Flutter – As 10 iterações do teste de consumos gráficos na aplicação Launch em Android

As Tabela 5 e Tabela 6, apresentadas abaixo, são resultantes do levantamento dos tempos de execução feitos pelas aplicações através das técnicas apresentadas acima.

Tabela 5 – Launch - Tempo de Execução em iOS

Iteração	React Native	Flutter
1	560 ms	940 ms
2	660 ms	1040 ms
3	650 ms	820 ms
4	690 ms	940 ms
5	560 ms	970 ms
6	610 ms	970 ms
7	710 ms	1130 ms
8	600 ms	970 ms
9	620 ms	860 ms
10	670 ms	1000 ms
Média	**633 ms**	**964 ms**

Iniciando pela análise do sistema operativo iOS, a diferença do tempo de arranque entre as *frameworks* é relativamente significativa, tendo a Flutter um consumo temporal 52% acima da React Native. Continuando com um maior consumo temporal, a Flutter no sistema operativo Android reduz ainda assim a diferença para 37% de tempo a mais.

Tabela 6 – Launch - Tempo de Execução em Android

Iteração	React Native	Flutter
1	440 ms	540 ms
2	420 ms	940 ms
3	540 ms	640 ms
4	500 ms	540 ms
5	480 ms	540 ms
6	480 ms	620 ms
7	420 ms	560 ms
8	560 ms	860 ms
9	430 ms	560 ms
10	400 ms	580 ms
Média	**467 ms**	**638 ms**

6.1.2. CPU

No consumo de CPU, a diferença entre as *frameworks* é relativamente mais curta no caso do iOS. Através da Tabela 7 é percetível que os máximos de consumos atingidos por ambas as aplicações são valores próximos, existindo, contudo, uma diferença de 5% de consumos superiores por parte da React Native.

Tabela 7 – Launch – CPU em iOS

Iteração	React Native	Flutter
1	300%	300%
2	250%	310%
3	280%	220%
4	290%	270%
5	270%	240%
6	260%	260%
7	270%	270%
8	270%	250%
9	280%	260%
10	310%	260%
Média	**278%**	**264%**

No sistema Android temos um elemento surpresa em relação ao caso do iOS. Os consumos de CPU máximos registados entre a React Native e a Flutter têm um fator de mais do dobro (cerca de 2,24) por parte da Flutter (Tabela 8) (20,10% em React Native contra 44,99% em Flutter).

Tabela 8 – Launch – CPU em Android

	React Native		Flutter	
Iteração	**Média CPU**	**Max CPU**	**Média CPU**	**Max CPU**
1	6,98%	15,60%	3,47%	37,90%
2	6,99%	15,60%	5,51%	81,80%
3	8,10%	30,00%	3,12%	35,40%
4	7,40%	22,50%	2,91%	35,40%
5	7,45%	29,70%	1,92%	38,70%
6	8,74%	19,30%	3,80%	36,60%
7	6,92%	15,10%	3,34%	27,50%
8	7,63%	26,60%	6,58%	84,30%
9	6,87%	12,90%	1,97%	34,40%
10	7,35%	13,70%	4,72%	37,90%
Média	**7,44%**	**20,10%**	**3,73%**	**44,99%**

No entanto, se verificarmos o gráfico da Figura 35, resultante de uma das iterações do teste executado, é percetível a razão pela qual a Flutter tem valores máximos extremamente superiores à React Native e, posteriormente, na média apresenta uma melhoria significativa, com aproximadamente metade dos consumos.

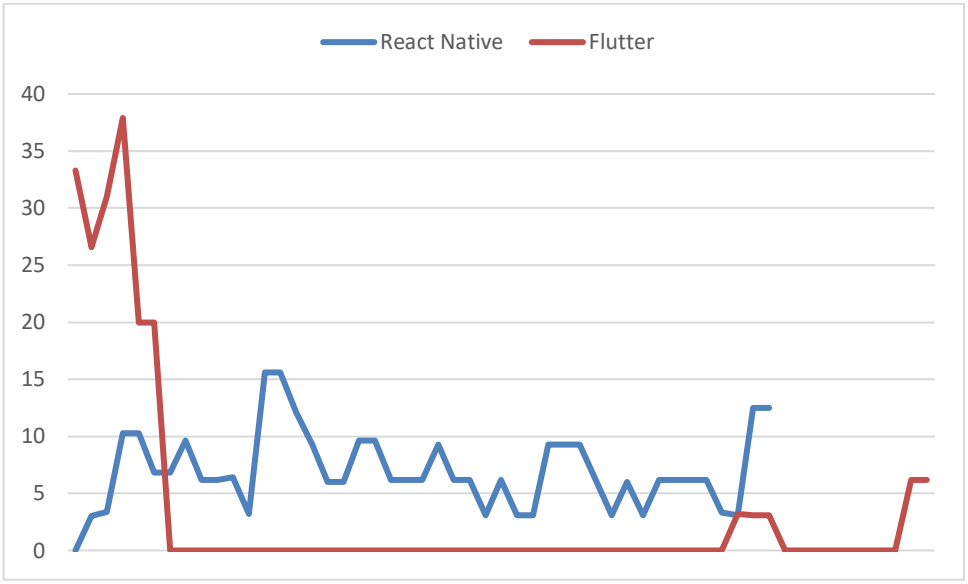


Figura 35 - Gráfico referente a consumos (%) de CPU praticados pela React Native e Flutter

Com a representação visual dos consumos de cada *framework*, pode dizer-se que a Flutter apresenta maiores consumos no início do arranque das aplicações, reduzindo significativamente os consumos no restante processo, enquanto a React Native é uma *framework* que tem os consumos distribuídos de uma forma mais constante temporalmente.

6.1.3. RAM

Relativamente ao consumo de memória RAM, no caso do sistema operativo iOS (Tabela 9), a Flutter tem cerca do dobro (1,98 vezes) de alocação de memória em relação à React Native.

Tabela 9 – Launch – RAM em iOS

Iteração	React Native	Flutter
1	22,27 MiB	65,77 MiB
2	22,31 MiB	65,77 MiB
3	22,31 MiB	65,80 MiB
4	22,29 MiB	62,00 MiB
5	22,28 MiB	65,78 MiB
6	142,27 MiB	90,54 MiB
7	22,29 MiB	65,77 MiB
8	22,30 MiB	65,82 MiB
9	22,29 MiB	65,77 MiB
10	22,27 MiB	65,80 MiB
Média	**34,29 MiB**	**67,88 MiB**

Com resultados mais equilibrados que no sistema iOS, são apresentados na Tabela 10 os valores referentes ao sistema operativo Android OS. No consumo de RAM, estes valores máximos ou médios registados são extremamente próximos. Assim, a Flutter apresenta uma ligeira vantagem em ambas as situações (diferenças de cerca de 5% a mais de RAM para a React Native).

Tabela 10 – Launch – RAM em Android

	React Native		Flutter	
Iteração	**Média RAM**	**Max RAM**	**Média RAM**	**Max RAM**
1	4,92%	5,00%	4,69%	4,70%
2	5,01%	5,10%	4,70%	4,80%
3	4,99%	5,00%	4,69%	4,70%
4	5,01%	5,10%	4,79%	4,80%
5	5,01%	5,10%	4,73%	4,80%
6	4,90%	4,90%	4,72%	4,80%
7	4,90%	4,90%	4,68%	4,70%
8	4,97%	5,00%	4,64%	4,70%
9	5,00%	5,00%	4,77%	4,80%
10	4,90%	4,90%	4,79%	4,80%
Média	**4,96%**	**5,00%**	**4,72%**	**4,76%**

6.1.4. Launch – React Native vs Flutter

Agrupando os valores de todas as métricas para se poder extrair uma conclusão desta funcionalidade, é possível concluir que, no caso do iOS (Tabela 11), a *framework* React Native acaba por ter vantagem no geral tendo em conta que apresenta valores melhores em duas das três métricas, e na métrica onde está em desvantagem, a distinção é bastante reduzida.

Tabela 11 - Launch - React Native vs Flutter em iOS

Métrica	React Native	Flutter
Tempo	633 ms	964 ms
CPU	278%	264%
RAM	34,29 MiB	67,88 MiB
Pontos	**2**	**1**

No entanto, ao analisarmos os resultados finais no Android OS (Tabela 12), a situação é inversa. Aqui, a Flutter é a *framework* que apresenta um maior número de métricas a seu favo, apesar de apresentar as métricas do tempo de execução e do CPU – Max a seu desfavor.

Tabela 12 - Launch - React Native vs Flutter em Android

Métrica	React Native	Flutter
Tempo	467 ms	638 ms
CPU - Max	20,10%	44,99%
CPU - Média	7,44%	3,73%
RAM - Max	5,00%	4,76%
RAM - Média	4,96%	4,72%
Pontos	**2**	**3**

6.2. Listas

6.2.1. Tempo de execução

Na renderização de listas, é comum que apenas os elementos que estejam visíveis no ecrã em determinado momento sejam renderizados. Nas *frameworks* em estudo, assim como nas formas de desenvolvimento nativas, este é o comportamento natural. Assim, todas as aplicações criadas com listas foram automaticamente preparadas para que assim que a lista

fosse apresentada com os primeiros elementos, fosse feito um *scroll* automático na lista até que o último elemento fosse apresentado. Com este comportamento, pretendia-se que as *frameworks* fossem forçadas a renderizar todos os elementos das listas em cada aplicação.

Todavia, tratando-se de duas *frameworks* distintas, o procedimento de realizar o *scroll* nas listas é também distinto. Sendo a React Native a *framework* mais limitativa relativamente a este aspeto, foram tomadas medidas para que ambas as *frameworks* executassem esta automatização dentro do mesmo tempo de interação.

Para isso, foram cronometrados os tempos que a React Native tomava para realizar o *scroll* em cada aplicação com listas (Tabela 13 e Tabela 14). Após a recolha dos tempos, foi definida a média destes, como tempo de execução no lado da Flutter. A recolha destes tempos de *scroll* foi feita pelo mesmo processo que um teste de avaliação, executando a aplicação sem qualquer tipo de dados no dispositivo e repetindo o processo 10 vezes para cada número de elementos.

Tabela 13 – Listas Locais - React Native tempo de *scroll*

	iOS			Android		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	271 ms	1600 ms	4270 ms	404 ms	2158 ms	7322 ms
2	277 ms	1601 ms	4151 ms	371 ms	2086 ms	7770 ms
3	277 ms	1623 ms	4118 ms	387 ms	2230 ms	7322 ms
4	276 ms	1620 ms	4136 ms	372 ms	2188 ms	7494 ms
5	292 ms	1606 ms	4319 ms	352 ms	2199 ms	7075 ms
6	279 ms	1612 ms	4268 ms	425 ms	2265 ms	7469 ms
7	275 ms	1613 ms	4201 ms	412 ms	2280 ms	7323 ms
8	287 ms	1615 ms	4234 ms	393 ms	2173 ms	7606 ms
9	273 ms	1615 ms	4134 ms	419 ms	2336 ms	7675 ms
10	278 ms	1600 ms	4217 ms	401 ms	2212 ms	7650 ms
Média	**279 ms**	**1612 ms**	**4203 ms**	**393 ms**	**2213 ms**	**7451 ms**
Des. Padrão	**6,36**	**8,34**	**68,70**	**23,30**	**70,07**	**212,12**

Tabela 14 – Listas Remotas - React Native tempo de scroll

	iOS			Android		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	268 ms	1604 ms	3391 ms	346 ms	2019 ms	5634 ms
2	276 ms	1628 ms	3319 ms	353 ms	1935 ms	5705 ms
3	274 ms	1618 ms	3430 ms	338 ms	2086 ms	5597 ms
4	285 ms	1617 ms	3312 ms	305 ms	1784 ms	5689 ms
5	284 ms	1618 ms	3325 ms	335 ms	2052 ms	5870 ms
6	274 ms	1609 ms	3345 ms	336 ms	2052 ms	5408 ms
7	269 ms	1609 ms	3314 ms	354 ms	2017 ms	5458 ms
8	272 ms	1610 ms	3354 ms	334 ms	1982 ms	5456 ms
9	272 ms	1609 ms	3345 ms	351 ms	2151 ms	5873 ms
10	284 ms	1614 ms	3324 ms	322 ms	2136 ms	5688 ms
Média	**276 ms**	**1614 ms**	**3346 ms**	**337 ms**	**2021 ms**	**5638 ms**
Des. Padrão	**6,34**	**6,88**	**37,98**	**15,22**	**106,13**	**162,88**

Calculada a média dos tempos realizados pela React Native e usada como tempo de execução do *scroll* nas listas criadas pela Flutter, acontece que o resultado não foi o esperado. Era esperado que os tempos feitos pela Flutter fossem precisamente os facultados ao mesmo, no entanto, a Flutter não é rígido com o tempo que lhe é estipulado para a execução do *scroll* automático (Tabela 15 e Tabela 16). Este com portamento pode ser resultante da gestão que a *framework* faz dos recursos durante a animação que cria. No entanto, também existiram sempre flutuações temporais em ambas as *frameworks*, como se pode verificar pelo desvio padrão apresentado nas tabelas, e neste caso é a React Native quem tem a maior instabilidade no decorrer da execução.

Tabela 15 – Listas Locais - Flutter tempo de *scroll*

	iOS			Android		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	296 ms	1629 ms	4211 ms	421 ms	2234 ms	7504 ms
2	295 ms	1629 ms	4216 ms	426 ms	2236 ms	7487 ms
3	295 ms	1629 ms	4217 ms	420 ms	2237 ms	7505 ms
4	296 ms	1629 ms	4218 ms	419 ms	2250 ms	7506 ms
5	296 ms	1629 ms	4217 ms	437 ms	2253 ms	7508 ms
6	296 ms	1627 ms	4216 ms	438 ms	2251 ms	7499 ms
7	295 ms	1630 ms	4217 ms	418 ms	2236 ms	7506 ms
8	295 ms	1630 ms	4217 ms	421 ms	2235 ms	7506 ms
9	296 ms	1629 ms	4219 ms	419 ms	2236 ms	7490 ms
10	297 ms	1629 ms	4212 ms	420 ms	2235 ms	7501 ms
Média	296 ms	1629 ms	4216 ms	424 ms	2241 ms	7501 ms
Des. Padrão	**0,67**	**0,82**	**2,54**	**7,49**	**7,69**	**7,22**

Tabela 16 – Listas Remotas - Flutter tempo de *scroll*

	iOS			Android		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	283 ms	1620 ms	3351 ms	371 ms	1815 ms	5595 ms
2	282 ms	1620 ms	3357 ms	367 ms	1793 ms	5608 ms
3	295 ms	1619 ms	3360 ms	375 ms	1794 ms	5601 ms
4	298 ms	1626 ms	3355 ms	368 ms	1806 ms	5608 ms
5	283 ms	1632 ms	3365 ms	365 ms	1800 ms	5603 ms
6	282 ms	1617 ms	3353 ms	363 ms	1804 ms	5606 ms
7	294 ms	1619 ms	3352 ms	346 ms	1805 ms	5627 ms
8	285 ms	1618 ms	3360 ms	376 ms	1793 ms	5612 ms
9	293 ms	1619 ms	3353 ms	362 ms	1801 ms	5609 ms
10	288 ms	1629 ms	3351 ms	343 ms	1806 ms	5615 ms
Média	288 ms	1622 ms	3356 ms	364 ms	1802 ms	5608 ms
Des. Padrão	**6,15**	**5,17**	**4,69**	**11,10**	**7,02**	**8,64**

Tendo em conta que, mesmo na tentativa de que ambas as aplicações executassem o movimento de *scroll* em tempos idênticos, tal não foi possível, a métrica temporal, como abordado anteriormente não incorpora o processo de *scroll* das listas.

Considerou-se, portanto, a métrica temporal neste caso como aquela responsável por cronometrar o tempo que é consumido para obter os dados para carregar na lista.

No caso das listas locais, este tempo conta desde que é requerida a criação de um determinado número de elementos, até que estes estejam inseridos numa base de dados local SQLite e prontos para serem renderizados na lista.

São apresentados nas tabelas seguintes os tempos consumidos por cada tecnologia nos dois tipos de listas. No caso do sistema operativo iOS, não restam dúvidas de que a React Native é a *framework* que tem melhores velocidades nas listas locais (Tabela 17), enquanto que a Flutter domina no caso das listas remotas (Tabela 18).

Tabela 17 - Listas Locais - Tempo de execução em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	23 ms	33 ms	46 ms	62 ms	95 ms	179 ms
2	22 ms	29 ms	45 ms	61 ms	109 ms	159 ms
3	21 ms	33 ms	44 ms	61 ms	110 ms	177 ms
4	22 ms	34 ms	44 ms	63 ms	109 ms	161 ms
5	21 ms	34 ms	45 ms	61 ms	109 ms	160 ms
6	21 ms	34 ms	44 ms	60 ms	109 ms	177 ms
7	21 ms	34 ms	43 ms	60 ms	109 ms	177 ms
8	21 ms	34 ms	47 ms	59 ms	109 ms	175 ms
9	22 ms	33 ms	44 ms	61 ms	110 ms	177 ms
10	22 ms	33 ms	44 ms	61 ms	111 ms	175 ms
Média	**21,6 ms**	**33,1 ms**	**44,6 ms**	**60,9 ms**	**108 ms**	**171,7 ms**

Nas listas remotas, a cronometragem foi feita desde que é feito o pedido de mensagens ao servidor até que a aplicação obtenha as mensagens, as carregue e esteja pronta a renderizar a lista. Idêntica à situação das listas locais, uma das *frameworks*, a Flutter, também se realça com uma maior capacidade de gestão de tempo.

Tabela 18 - Litas Remotas - Tempo de execução em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	59 ms	68 ms	97 ms	64 ms	98 ms	46 ms
2	116 ms	61 ms	78 ms	47 ms	65 ms	42 ms
3	76 ms	106 ms	68 ms	63 ms	93 ms	36 ms
4	95 ms	78 ms	75 ms	45 ms	63 ms	51 ms
5	57 ms	56 ms	116 ms	29 ms	35 ms	76 ms
6	78 ms	68 ms	66 ms	22 ms	96 ms	42 ms
7	101 ms	58 ms	61 ms	32 ms	52 ms	65 ms
8	48 ms	66 ms	58 ms	30 ms	31 ms	41 ms
9	111 ms	184 ms	104 ms	50 ms	34 ms	41 ms
10	56 ms	99 ms	85 ms	40 ms	81 ms	59 ms
Média	**79,7 ms**	**84,4 ms**	**80,8 ms**	**42,2 ms**	**64,8 ms**	**49,9 ms**

Em Android, com valores entre os 30 e 40% dos tempos praticados pela Flutter em listas locais, independentemente das quantidades, a React Native é a *framework* que melhores tempos apresentou (Tabela 19).

Tabela 19 - Listas Locais - Tempo de Execução em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	43 ms	84 ms	121 ms	151 ms	260 ms	369 ms
2	43 ms	67 ms	115 ms	127 ms	271 ms	394 ms
3	46 ms	84 ms	131 ms	125 ms	250 ms	394 ms
4	54 ms	85 ms	129 ms	117 ms	227 ms	394 ms
5	48 ms	86 ms	110 ms	115 ms	264 ms	370 ms
6	53 ms	84 ms	126 ms	124 ms	239 ms	426 ms
7	47 ms	87 ms	128 ms	115 ms	250 ms	365 ms
8	42 ms	87 ms	103 ms	114 ms	264 ms	406 ms
9	47 ms	83 ms	130 ms	113 ms	227 ms	346 ms
10	49 ms	85 ms	132 ms	83 ms	241 ms	421 ms
Média	**47,2 ms**	**83,2 ms**	**122,5 ms**	**118,4 ms**	**249,3 ms**	**388,5 ms**

No entanto, no que diz respeito às listas remotas (Tabela 20), o cenário é o inverso. Nestas, a Flutter no sistema operativo Android também faz uso de apenas cerca de 30% do tempo praticado pela React Native, enquanto que no iOS existe uma similaridade maior em tempos, chegando próximo de 77% do valor da React Native.

Tabela 20 - Listas Remotas - Tempo de Execução em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	285 ms	235 ms	360 ms	52 ms	82 ms	96 ms
2	283 ms	293 ms	372 ms	58 ms	64 ms	88 ms
3	227 ms	264 ms	281 ms	112 ms	44 ms	76 ms
4	311 ms	251 ms	256 ms	168 ms	103 ms	69 ms
5	359 ms	315 ms	276 ms	73 ms	62 ms	85 ms
6	276 ms	208 ms	223 ms	77 ms	96 ms	133 ms
7	250 ms	283 ms	239 ms	55 ms	75 ms	69 ms
8	206 ms	260 ms	329 ms	68 ms	50 ms	76 ms
9	212 ms	294 ms	336 ms	47 ms	108 ms	59 ms
10	199 ms	224 ms	324 ms	83 ms	59 ms	58 ms
Média	**260,8 ms**	**262,7 ms**	**299,6 ms**	**79,3 ms**	**74,3 ms**	**80,9 ms**

6.2.2. CPU

No consumo de CPU, na funcionalidade de listas em iOS, a React Native não se encontra com valores muito próximos dos que são apresentados pela Flutter. Esta é a *framework* que exige menos dos dispositivos, em todas as variações de listas. Como é apresentado nas seguintes tabelas, tanto nas listas locais (Tabela 21) como nas remotas (Tabela 22), existe uma diferença de consumos relevante. No caso das listas locais, a diferença entre as variações das listas não é muito acentuada, mantendo-se a Flutter com aproximadamente 60% dos consumos da React Native.

Tabela 21 - Listas Locais - CPU em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	230%	310%	270%	100%	120%	170%
2	220%	220%	290%	170%	150%	250%
3	190%	260%	240%	140%	140%	150%
4	230%	230%	270%	170%	170%	130%
5	260%	230%	270%	130%	140%	150%
6	240%	240%	250%	180%	130%	150%
7	220%	230%	280%	140%	140%	180%
8	200%	240%	290%	110%	150%	150%
9	320%	250%	300%	130%	160%	130%
10	230%	270%	290%	150%	170%	160%
Média	**234%**	**248%**	**275%**	**142%**	**147%**	**162%**

No caso das listas remotas, há uma tendência de aumento da diferença entre as *frameworks* desfavorável à React Native, à medida que o tamanho das listas aumenta, iniciando numa proporção próxima de 31% maior (listas com 10 itens), e concluindo com uma proporção de cerca 57% maior (listas com 100 itens).

Tabela 22 - Listas Remotas - CPU em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	270%	430%	300%	180%	230%	200%
2	300%	400%	450%	230%	340%	280%
3	280%	410%	400%	210%	260%	230%
4	280%	320%	410%	190%	280%	300%
5	260%	330%	390%	230%	310%	250%
6	280%	300%	380%	220%	240%	260%
7	250%	370%	400%	170%	240%	220%
8	270%	390%	440%	190%	270%	250%
9	230%	350%	370%	220%	280%	220%
10	270%	330%	350%	220%	290%	260%
Média	**269%**	**363%**	**389%**	**206%**	**274%**	**247%**

Analisando agora a situação no sistema operativo Android OS, iniciando pelas listas locais e avaliando primeiro os valores máximos (Tabela 23), a situação é idêntica ao sistema iOS. A Flutter é a *framework* que apresenta melhores resultados, tendo iterações com consumos próximos da metade dos consumos da React Native, nas listas de 100 itens.

Tabela 23 - Listas Locais - Max CPU em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	173,00%	176,00%	190,00%	120,00%	143,00%	113,00%
2	180,00%	180,00%	203,00%	156,00%	132,00%	164,00%
3	160,00%	200,00%	218,00%	109,00%	133,00%	175,00%
4	175,00%	193,00%	205,00%	125,00%	148,00%	140,00%
5	179,00%	173,00%	164,00%	93,70%	141,00%	133,00%
6	173,00%	193,00%	153,00%	106,00%	150,00%	110,00%
7	172,00%	246,00%	212,00%	121,00%	151,00%	173,00%
8	179,00%	241,00%	193,00%	128,00%	103,00%	162,00%
9	135,00%	193,00%	220,00%	126,00%	146,00%	130,00%
10	196,00%	186,00%	200,00%	118,00%	154,00%	103,00%
Média	**172,20%**	**198,10%**	**195,80%**	**120,27%**	**140,10%**	**140,30%**

Pelos resultados obtidos pela média de consumos (Tabela 24), a React Native continua em desvantagem, piorando os valores ligeiramente em relação aos máximos. Por exemplo, a diferença dos valores máximos praticados pela *framework* nas listas de 50 itens passa de cerca 41% pior que a Flutter, para cerca de 73% através dos valores médios.

Tabela 24 - Listas Locais - Média CPU em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	15,14%	25,08%	35,84%	8,40%	17,85%	19,84%
2	17,08%	25,78%	42,24%	8,42%	14,69%	23,51%
3	12,38%	32,00%	42,01%	6,02%	18,45%	30,52%
4	15,04%	29,45%	43,22%	9,13%	11,77%	28,01%
5	15,10%	23,63%	34,27%	9,04%	18,47%	20,14%
6	11,67%	32,25%	37,72%	6,53%	16,51%	18,90%
7	12,98%	31,73%	46,38%	8,20%	18,84%	29,20%
8	15,18%	31,49%	42,10%	10,57%	14,12%	26,90%
9	12,52%	27,50%	42,18%	11,65%	15,82%	27,13%
10	14,53%	21,05%	41,70%	9,10%	15,34%	19,83%
Média	**14,16%**	**28,00%**	**40,77%**	**8,71%**	**16,19%**	**24,40%**

Nas listas remotas, o cenário é relativamente diferente e a React Native é a *framework* que melhor resultado obtém no geral. A Flutter, tendo em conta os máximos de consumo (Tabela 25), tem uma pior *performance* em todas as variações das listas. No entanto, ao analisarmos a média de consumos, o cenário é relativamente idêntico, mas com algumas alterações interessantes de se observarem.

Tabela 25 - Listas Remotas - Max CPU em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	145,00%	193,00%	214,00%	206,00%	313,00%	235,00%
2	166,00%	206,00%	187,00%	213,00%	256,00%	275,00%
3	141,00%	190,00%	273,00%	175,00%	219,00%	513,00%
4	167,00%	169,00%	206,00%	160,00%	256,00%	209,00%
5	175,00%	200,00%	190,00%	190,00%	257,00%	289,00%
6	183,00%	186,00%	250,00%	210,00%	267,00%	177,00%
7	150,00%	206,00%	214,00%	218,00%	367,00%	260,00%
8	114,00%	212,00%	196,00%	100,00%	270,00%	250,00%
9	137,00%	256,00%	184,00%	206,00%	386,00%	267,00%
10	156,00%	196,00%	151,00%	190,00%	325,00%	221,00%
Média	**153,40%**	**201,40%**	**206,50%**	**186,80%**	**291,60%**	**269,60%**

Pela média de consumos (Tabela 26), na lista de 10 itens, a diferença de resultados é relativamente idêntica à análise obtida pelos máximos. Porém, acontece que nas listas de 50 itens, os valores passam de aproximadamente 45% piores pela Flutter, no máximo de consumos, para cerca de 9%, sendo uma diferença de 36%, e no caso da lista de 100 itens, existe uma distanciação próxima de 48% entre a média e os máximos praticados pela Flutter.

Tabela 26 - Listas Remotas - Média CPU em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	28,76%	37,87%	46,34%	33,68%	38,59%	36,47%
2	29,77%	32,95%	38,27%	31,01%	40,46%	37,73%
3	27,41%	37,16%	43,06%	34,44%	41,26%	27,99%
4	32,80%	38,54%	41,21%	32,47%	34,87%	36,78%
5	28,94%	36,54%	41,25%	42,39%	34,31%	38,78%
6	30,65%	30,87%	44,59%	39,11%	34,44%	32,20%
7	30,19%	36,76%	44,41%	36,52%	41,98%	37,14%
8	32,98%	33,61%	41,04%	38,99%	40,26%	32,20%
9	24,84%	38,48%	42,64%	31,76%	35,68%	36,40%
10	28,61%	35,89%	37,88%	32,80%	47,40%	33,45%
Média	**29,49%**	**35,87%**	**42,07%**	**35,32%**	**38,93%**	**34,91%**

Por forma a perceber qual a realidade da situação, são apresentados gráficos correspondentes das iterações do teste realizado às listas de 50 e 100 itens, sendo as que apresentaram uma maior diferença entre os máximos e a média de valores de consumo.

No caso das listas de 50 itens, é visível pelo gráfico da Figura 36 que ambas as *frameworks* obtiveram três situações em que foi exigido um maior consumo. Duas dessas situações encontram-se entre os 150% a 200% de consumo em ambas, mas no caso da Flutter o terceiro momento escalou até valores superiores a 300%, acabando assim por apresentar um máximo que afeta o seu desempenho na comparação através de máximos.

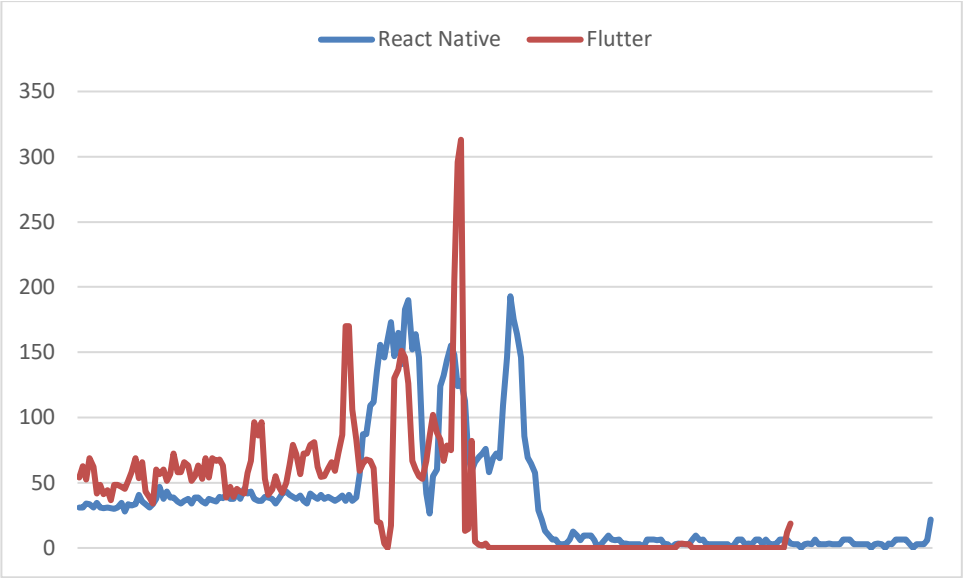


Figura 36 - Listas Remotas 50 Itens - gráfico de consumos (%) de CPU em Android

Na lista de 100 elementos (Figura 37), ambas as *frameworks* apresentam um aglomerado de momentos mais intensos e com diversas flutuações, mas no caso da Flutter, essas flutuações são mais curtas, como se pode verificar pelo primeiro pico de consumo em ambas as situações. No entanto, a Flutter tem um pico superior à React Native, daí nos máximos aparentar pior *performance*.

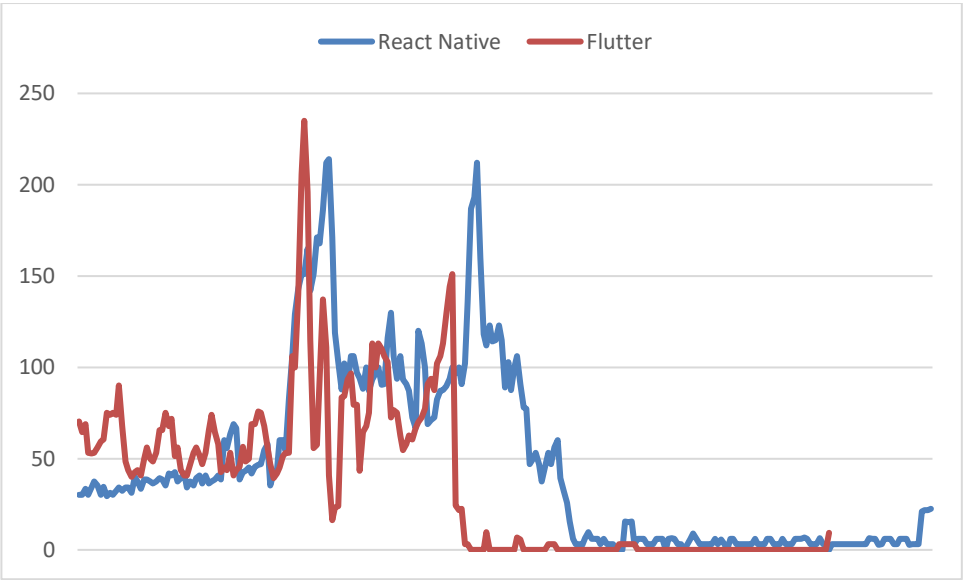


Figura 37 - Listas Remotas 100 Itens - gráfico de consumos (%) de CPU em Android

6.2.3. RAM

No consumo de memória RAM é bem claro qual a *framework* que é mais económica no sistema operativo iOS. A React Native, na maioria dos casos, tem consumos inferiores a metade dos que são praticados pela Flutter. Contudo, ao verificarmos o caso das listas locais (Tabela 27), é possível observar que quanto maior a quantidade de elementos a lista contém, mais proximidade existe nos valores coletados por ambas as *frameworks*.

Tabela 27 - Listas Locais - RAM em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	6,14 MiB	20,45 MiB	48,08 MiB	24,61 MiB	42,29 MiB	93,94 MiB
2	7,14 MiB	19,97 MiB	48,15 MiB	25,46 MiB	60,76 MiB	95,37 MiB
3	7,12 MiB	19,99 MiB	48,33 MiB	26,44 MiB	40,45 MiB	94,96 MiB
4	7,11 MiB	20,07 MiB	48,08 MiB	26,74 MiB	51,08 MiB	95,91 MiB
5	6,07 MiB	19,97 MiB	48,92 MiB	25,39 MiB	46,88 MiB	95,71 MiB
6	7,10 MiB	20,40 MiB	47,66 MiB	26,76 MiB	38,45 MiB	96,76 MiB
7	7,09 MiB	18,96 MiB	48,23 MiB	26,20 MiB	40,21 MiB	95,26 MiB
8	7,09 MiB	20,01 MiB	48,31 MiB	26,72 MiB	60,62 MiB	95,52 MiB
9	7,19 MiB	18,97 MiB	48,21 MiB	26,40 MiB	49,37 MiB	93,01 MiB
10	7,13 MiB	19,66 MiB	4,18 MiB	26,77 MiB	46,62 MiB	93,55 MiB
Média	**6,92 MiB**	**19,85 MiB**	**43,82 MiB**	**26,15 MiB**	**47,67 MiB**	**95,00 MiB**

Na aplicação das listas remotas (Tabela 28), a situação da Flutter é um pouco mais desfavorável relativamente às listas até 50 elementos, inclusive. Já no caso das listas com 100 elementos, embora ainda com consumos superiores, a Flutter desce a sua distância de maior consumo para aproximadamente 1,34 vezes superior, em relação à proporção de 2,17 vezes a mais que tem nas listas locais.

Tabela 28 - Listas Remotas - RAM em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	32,97 MiB	96,58 MiB	99,11 MiB	171,63 MiB	214,28 MiB	105,63 MiB
2	34,75 MiB	40,80 MiB	60,38 MiB	171,60 MiB	819,82 MiB	256,16 MiB
3	32,84 MiB	78,26 MiB	178,26 MiB	169,69 MiB	337,56 MiB	253,54 MiB
4	34,68 MiB	43,99 MiB	140,32 MiB	171,99 MiB	838,33 MiB	218,92 MiB
5	35,43 MiB	57,46 MiB	128,85 MiB	170,02 MiB	899,26 MiB	192,22 MiB
6	35,65 MiB	62,80 MiB	194,70 MiB	172,06 MiB	815,46 MiB	184,17 MiB
7	35,34 MiB	70,18 MiB	180,22 MiB	169,47 MiB	616,99 MiB	285,66 MiB
8	35,68 MiB	65,40 MiB	197,56 MiB	169,71 MiB	288,29 MiB	239,46 MiB
9	34,59 MiB	29,50 MiB	195,95 MiB	171,21 MiB	157,92 MiB	192,41 MiB
10	33,99 MiB	97,29 MiB	181,82 MiB	171,19 MiB	843,01 MiB	156,57 MiB
Média	**34,59 MiB**	**64,23 MiB**	**155,72 MiB**	**170,86 MiB**	**583,09 MiB**	**208,47 MiB**

No caso do sistema operativo Android OS, os resultados não têm qualquer relação com os praticados no iOS. Nas listas locais, seja com recurso aos máximos de consumos praticados (Tabela 29), seja pela média (Tabela 30), os resultados são comparativamente idênticos, sendo a Flutter a *framework* que tem a melhor *performance*.

Tabela 29 - Listas Locais - Max RAM em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	6,00 MiB	6,10 MiB	6,60 MiB	5,50 MiB	5,60 MiB	5,70 MiB
2	5,90 MiB	6,10 MiB	6,80 MiB	5,50 MiB	5,60 MiB	5,70 MiB
3	5,90 MiB	6,10 MiB	6,70 MiB	5,50 MiB	5,60 MiB	5,70 MiB
4	6,00 MiB	6,10 MiB	6,60 MiB	5,50 MiB	5,60 MiB	5,70 MiB
5	5,90 MiB	6,20 MiB	6,60 MiB	5,50 MiB	5,70 MiB	5,70 MiB
6	5,90 MiB	6,10 MiB	6,70 MiB	5,40 MiB	5,60 MiB	5,70 MiB
7	5,90 MiB	6,10 MiB	6,70 MiB	5,50 MiB	5,70 MiB	5,60 MiB
8	5,90 MiB	6,10 MiB	6,70 MiB	5,50 MiB	5,60 MiB	5,60 MiB
9	5,90 MiB	6,10 MiB	6,70 MiB	5,60 MiB	5,60 MiB	5,60 MiB
10	5,90 MiB	6,20 MiB	6,70 MiB	5,50 MiB	5,60 MiB	5,70 MiB
Média	**5,92 MiB**	**6,12 MiB**	**6,68 MiB**	**5,50 MiB**	**5,62 MiB**	**5,67 MiB**

Nas listas locais, os consumos são relativamente idênticos, com a Flutter em vantagem. No entanto, este não só tem uma vantagem do geral dos consumos, como com o aumento do número de elementos nas listas apresenta uma melhoria dos consumos relativamente aos que são praticados pela React Native.

Tabela 30 - Listas Locais - Média RAM em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	5,68 MiB	5,79 MiB	6,17 MiB	5,25 MiB	5,35 MiB	5,42 MiB
2	5,70 MiB	5,83 MiB	6,33 MiB	5,21 MiB	5,33 MiB	5,42 MiB
3	5,66 MiB	5,87 MiB	6,27 MiB	5,17 MiB	5,34 MiB	5,43 MiB
4	5,75 MiB	5,90 MiB	6,23 MiB	5,16 MiB	5,34 MiB	5,41 MiB
5	5,67 MiB	5,83 MiB	6,22 MiB	5,24 MiB	5,42 MiB	5,43 MiB
6	5,65 MiB	5,88 MiB	6,25 MiB	5,17 MiB	5,31 MiB	5,45 MiB
7	5,62 MiB	5,84 MiB	6,30 MiB	5,20 MiB	5,32 MiB	5,36 MiB
8	5,70 MiB	5,90 MiB	6,27 MiB	5,19 MiB	5,36 MiB	5,35 MiB
9	5,64 MiB	5,84 MiB	6,31 MiB	5,27 MiB	5,36 MiB	5,37 MiB
10	5,71 MiB	5,90 MiB	6,26 MiB	5,22 MiB	5,31 MiB	5,48 MiB
Média	**5,68 MiB**	**5,86 MiB**	**6,26 MiB**	**5,21 MiB**	**5,34 MiB**	**5,41 MiB**

Nas listas remotas, os valores máximos praticados não são um reflexo das médias dos valores. Pelos valores máximos praticados (Tabela 31), a React Native sai em vantagem nas listas de mais elementos, com consumos inferiores à Flutter. A Flutter apresenta assim consumos de aproximadamente 62% superiores em listas de 50 itens, e cerca de 17% superior em listas de 100 itens. Nas listas de 10 elementos, as *frameworks* têm uma *performance* idêntica.

Tabela 31 - Listas Remotas - Max RAM em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	8,00%	10,50%	8,50%	7,70%	17,50%	7,60%
2	7,90%	9,80%	9,10%	7,80%	17,50%	8,00%
3	7,90%	10,70%	8,90%	7,70%	15,50%	10,60%
4	8,00%	10,30%	10,10%	7,70%	16,60%	8,50%
5	7,90%	10,00%	9,20%	7,80%	15,70%	8,10%
6	7,90%	10,70%	9,50%	8,00%	18,20%	16,10%
7	8,00%	10,30%	8,50%	7,90%	15,90%	10,90%
8	8,00%	10,60%	8,50%	7,70%	17,40%	16,10%
9	8,00%	10,50%	10,20%	7,80%	17,10%	7,90%
10	8,20%	10,70%	8,80%	7,80%	16,90%	12,70%
Média	**7,98%**	**10,41%**	**9,13%**	**7,79%**	**16,83%**	**10,65%**

Vendo os valores pela média de consumos (Tabela 32), ambas as *frameworks* apresentam desempenhos idênticos, com ligeira vantagem da Flutter nas listas de 10 e 100 elementos, e da React Native na lista de 50 elementos.

Tabela 32 - Listas Remotas - Média RAM em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	6,75%	7,24%	7,37%	6,32%	7,48%	6,48%
2	6,77%	6,74%	7,74%	6,17%	7,28%	6,62%
3	6,84%	7,28%	7,36%	6,18%	6,87%	7,66%
4	6,80%	7,26%	8,24%	6,25%	6,89%	6,62%
5	6,80%	6,93%	7,40%	6,26%	7,76%	6,60%
6	6,80%	7,17%	7,64%	6,33%	7,91%	8,32%
7	6,69%	7,13%	7,30%	6,27%	7,81%	7,16%
8	6,82%	7,23%	7,43%	6,28%	7,49%	8,48%
9	6,70%	7,19%	8,09%	6,10%	7,72%	6,56%
10	6,63%	7,23%	7,48%	6,15%	7,34%	8,02%
Média	**6,76%**	**7,14%**	**7,60%**	**6,23%**	**7,46%**	**7,25%**

No decorrer dos testes, na renderização dos vídeos que se encontravam como itens das listas, a *framework* React Native não chegava a renderizar a primeira *frame* destes. Esta situação ocorria pelo facto de que era feito o *scroll* na lista e a React Native, por motivos de otimização, acabava por não renderizar o primeiro *frame* dos vídeos, enquanto que a Flutter renderizava esse conteúdo. Tendo este aspeto em conta, os resultados obtidos poderão estar ligados a essa situação, visto que é precisamente nas listas maiores onde ocorre a maior diferença.

Tendo em conta que foi nas listas de 50 elementos que se verificou uma maior distinção entre as *frameworks* e também tendo em conta o ponto anterior, são apresentados os consumos através do gráfico da Figura 38, referentes a uma das iterações do teste realizado às mesmas.

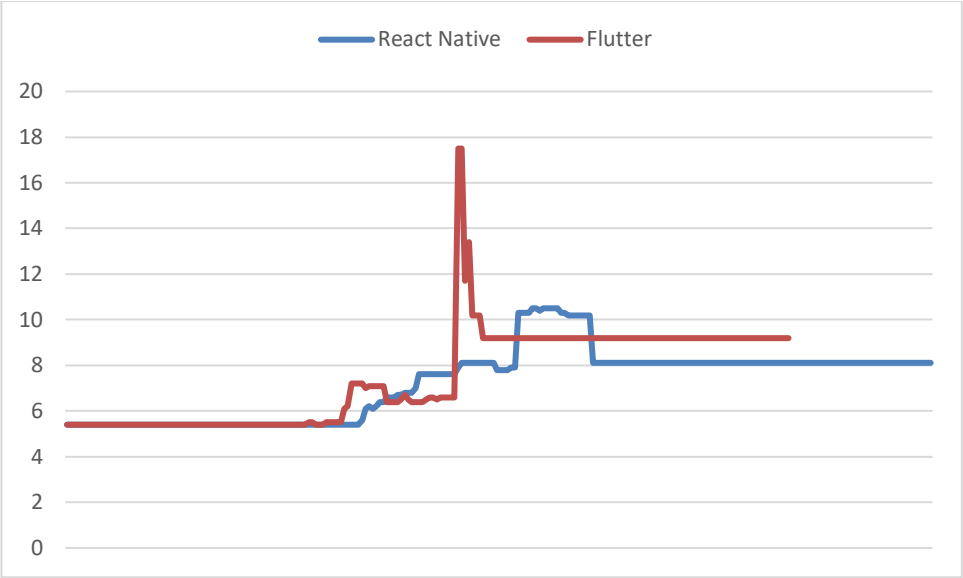


Figura 38 - Listas Remotas 50 Itens – gráfico dos consumos (%) de RAM em Android

É percetível que existe um momento em ambas as *frameworks* alocam uma maior quantidade de memória. No caso da Flutter, esta aloca um valor máximo relativamente superior ao alocado pela React Native. Esta situação poderá estar ligada à questão da renderização dos vídeos abordada anteriormente, podendo haver algum vídeo colocado numa posição que acaba por criar esta conjuntura.

Excluindo a situação das listas remotas no Android OS, em todas as outras situações de listas, quanto maior a quantidade de elementos, maior é o distanciamento das *frameworks* relativamente aos consumos de memória RAM praticados, sendo a Flutter a Framework que parece exercer uma melhor gestão quando se trata de maiores quantidades de elementos.

6.2.4. FPS

Nesta secção aborda-se a questão dos *frames* por segundo que cada *framework* renderizou no decorrer da funcionalidade das listas locais no sistema operativo iOS. Na questão dos valores máximos de FPS atingidos (Tabela 33), a Flutter domina as listas de 50 e 100 itens. Aqui, a Flutter consegue renderizar cerca de 1,25 vezes mais FPS para a lista de 50 itens, e de 2,68 vezes mais FPS para a lista de 100 FPS. A React Native renderizou, na lista de 10 elementos, cerca de 1,13 vezes mais FPS que a Flutter.

Tabela 33 - Listas Locais - Max FPS em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	21	28	22	21	20	60
2	20	29	23	21	60	60
3	21	25	22	14	20	60
4	20	21	21	19	39	60
5	21	21	23	21	27	60
6	21	33	22	21	20	60
7	21	28	21	13	21	60
8	21	21	23	19	60	60
9	21	22	23	17	25	60
10	24	27	24	21	27	60
Média	**21,10**	**25,50**	**22,40**	**18,70**	**31,90**	**60,00**

No entanto, a React Native é a *framework* que domina na média de valores de FPS renderizados (Tabela 34) nas listas de 10 e 50 elementos, com uma percentagem superior a 9% de FPS a mais. A Flutter, na lista com 100 elementos, tem uma vantagem consideravelmente superior de FPS a mais, que se aproxima de duas vezes e meia.

Tabela 34 - Lista Locais - Média FPS em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	9,7	9	10,2	8,7	5,7	25,1
2	6,8	8,1	10,6	10,2	13,5	25
3	10	8,7	10,4	9	4,6	23,5
4	8,8	8,4	10,3	8,3	10,6	26
5	10	8,2	11,1	9,8	8,1	24,8
6	9,3	13,5	10	9	3,8	25,1
7	7,8	9,5	9,4	6,3	6,6	24,8
8	9,7	8,2	9,6	8	13	24,7
9	11	8,7	10,7	10	10,3	25,9
10	15	11,6	10,1	10,5	7,9	24,7
Média	**9,81**	**9,39**	**10,24**	**8,98**	**8,41**	**24,96**

No caso das listas remotas, a Flutter é a *framework* que tem melhores resultados em todas as situações. Tanto a nível de máximos atingidos como relativamente à média de FPS registados no decorrer das aplicações. Considerando os valores máximos (Tabela 35), a Flutter tem uma *performance* superior a 44% em todos os casos, chegando mesmo a ter cerca de 2,63 vezes mais FPS que a React Native.

Tabela 35 - Listas Remotas - Max FPS em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	21	38	35	60	59	60
2	21	39	39	60	59	60
3	20	40	33	60	60	33
4	21	29	33	60	59	29
5	22	22	34	59	60	34
6	23	43	38	60	60	59
7	21	42	38	60	60	60
8	21	40	37	60	60	60
9	21	22	36	60	60	60
10	36	35	35	59	60	60
Média	**22,70**	**35,00**	**35,80**	**59,80**	**59,70**	**51,50**

Na média de valores (Tabela 36), a distância entre *frameworks* é ligeiramente inferior. No entanto a diferença atingida pela superioridade da Flutter situa-se aproximadamente entre os 70% e os 109%.

Tabela 36 – Listas Remotas – Média FPS em iOS

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	15,8	20	15,3	37,3	27,2	33,2
2	15,8	16,9	16,5	34,2	32,2	32,6
3	15	15,8	16,4	32	28,4	33
4	17,5	15	15,9	33,2	28	25,7
5	15	16,5	18,6	32,2	28,4	34
6	17,5	16,9	20,2	32	27,6	31,6
7	15	16,7	18,7	31,3	26,1	30,6
8	14	14,5	13,4	32	29,9	31,7
9	17	16,5	19,9	34	27,8	39
10	19	16,8	12,5	39,8	25,8	32
Média	**16,16**	**16,56**	**16,74**	**33,80**	**28,14**	**32,34**

Para o sistema operativo Android OS, a medição é sendo realizada de forma distinta, pelo que os valores são também distintos. No entanto, a conclusão final continua a ser a mesma, a Flutter é a *framework* que melhor *performance* apresenta. É de salientar um caso, nas listas locais (Tabela 37), em que a Flutter não registou qualquer *frame* abaixo dos 16,666 ms e a React Native registou uma média próxima de 5,5, demonstrando assim que a aplicação da Flutter foi renderizada sempre a uma taxa de atualização a rondar os 60 FPS, sendo um aspeto extremamente positivo.

Tabela 37 - Listas Locais - FPS - *Janky frames* em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	6	9	19	0	0	3
2	4	27	14	0	0	2
3	5	8	11	0	0	0
4	7	6	11	0	0	0
5	8	13	12	0	0	1
6	5	8	27	0	0	0
7	7	6	12	0	0	0
8	4	14	34	0	3	0
9	5	8	17	0	0	0
10	4	20	11	0	0	1
Média	**5,5**	**11,9**	**16,8**	**0**	**0,3**	**0,7**

Em termos de *Janky frames* ocorridos nas listas remotas (Tabela 38), o número por parte da React Native chega, em determinados casos, a ser mais de 50 vezes superior ao da Flutter.

Tabela 38 - Listas Remotas - FPS - Janky *frames* em Android

	React Native			Flutter		
Iteração	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
1	23	19	43	1	2	1
2	4	70	46	0	3	2
3	21	18	53	0	0	2
4	24	47	58	0	1	2
5	42	18	54	1	0	1
6	7	19	48	0	2	1
7	4	17	55	0	4	0
8	10	18	52	0	1	0
9	12	19	60	1	4	3
10	23	24	54	1	0	0
Média	**17**	**26,9**	**52,3**	**0,4**	**1,7**	**1,2**

6.2.5. Listas – React Native vs Flutter

Procedendo à análise dos valores conclusivos das listas locais de forma conjunta, podemos observar pelas Tabela 39 e Tabela 40, que estamos perante uma situação em que temos resultados completamente distintos para sistemas operativos também distintos. No caso do iOS (Tabela 39), temos a React Native que no geral é a melhor *framework*, embora a Flutter tenha melhores resultados nos consumos de CPU e melhores FPS em listas grandes.

Tabela 39 - React Native vs Flutter - Listas Locais em iOS

	React Native			Flutter		
Métrica	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
Tempo	21,6 ms	33,1 ms	44,6 ms	60,9 ms	108 ms	171,7 ms
CPU	234%	248%	275%	142%	147%	162%
RAM	6,92 MiB	19,85 MiB	43,82 MiB	26,15 MiB	47,67 MiB	95,00 MiB
FPS - Max	21,10	25,50	22,40	18,70	31,90	60,00
FPS - Média	9,81%	9,39%	10,24%	8,98%	8,41%	24,96%
Pontos	**9**			**6**		

No cenário do Android OS (Tabela 40), a React Native é a *framework* mais rápida, mas relativamente aos consumos de CPU, memória RAM e renderização de FPS, a Flutter domina visivelmente com melhores resultados.

Tabela 40 - React Native vs Flutter - Listas Locais em Android

	React Native			Flutter		
Métrica	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
Tempo	47,2 ms	83,2 ms	122,5 ms	118,4 ms	249,3 ms	388,5 ms
CPU - Max	172,20%	198,10%	195,80%	120,27%	140,10%	140,30%
CPU - Média	14,16%	28,00%	40,77%	8,71%	16,19%	24,40%
RAM - Max	5,92 MiB	6,12 MiB	6,68 MiB	5,50 MiB	5,62 MiB	5,67 MiB
RAM - Média	5,68 MiB	5,86 MiB	6,26 MiB	5,21 MiB	5,34 MiB	5,41 MiB
Janky Frames	5,5	11,9	16,8	0	0,3	0,7
Pontos	**3**			**15**		

Nas listas remotas é a mesma *framework* que obtém os melhores resultados em ambos os sistemas operativos. A Flutter é a melhor abordagem para o sistema operativo iOS (Tabela 41), exceto no que diz respeito aos consumos de memória volátil e RAM.

Tabela 41 - React Native vs Flutter - Listas Remotas em iOS

	React Native			Flutter		
Métrica	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
Tempo	79,7 ms	84,4 ms	80,8 ms	42,2 ms	64,8 ms	49,9 ms
CPU	269%	363%	389%	206%	274%	247%
RAM	34,59 MiB	64,23 MiB	155,72 MiB	170,86 MiB	583,09 MiB	208,47 MiB
FPS - Max	22,70	35,00	35,80	59,80	59,70	51,50
FPS - Média	16,16	16,56	16,74	33,80	28,14	32,34
Pontos	**3**			**12**		

No Android OS (Tabela 42), a React Native já tem algumas melhorias maioritariamente no que se refere aos consumos de CPU. Contudo, ainda assim, a Flutter no geral tem melhores valores registados.

Tabela 42 - React Native vs Flutter - Listas Remotas em Android

	React Native			Flutter		
Métrica	**10 Itens**	**50 Itens**	**100 Itens**	**10 Itens**	**50 Itens**	**100 Itens**
Tempo	260,8 ms	262,7 ms	299,6 ms	79,3 ms	74,3 ms	80,9 ms
CPU - Max	153,40%	201,40%	206,50%	186,80%	291,60%	269,60%
CPU - Média	29,49%	35,87%	42,07%	35,32%	38,93%	34,91%
RAM - Max	7,98%	10,41%	9,13%	7,79%	16,83%	10,65%
RAM - Média	6,76%	7,14%	7,60%	6,23%	7,46%	7,25%
Janky Frames	17	26,9	52,3	0,4	1,7	1,2
Pontos	**8**			**10**		

6.3. Câmara

6.3.1. CPU

A React Native, no que diz respeito ao processamento CPU no iOS (Tabela 43), tem um consumo relativamente próximo entre os vários tipos de conteúdos multimédia a capturar, com apenas 1% de diferença entre fotografia e vídeo, enquanto que a Flutter tem 28% de diferença neste aspeto. Relativamente às *frameworks*, verificou-se que uma tem melhores resultados na captura de imagens (Flutter com aproximadamente 11% de consumos a menos), e a outra tem melhores resultados na captura de vídeos (React Native com uma pequena vantagem a rondar os 3%).

Tabela 43 - Captura de conteúdos multimédia - CPU em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	260%	220%	200%	280%
2	210%	220%	190%	210%
3	220%	190%	160%	280%
4	210%	190%	180%	190%
5	260%	190%	180%	250%
6	220%	230%	180%	160%
7	200%	170%	190%	230%
8	210%	240%	270%	230%
9	180%	190%	200%	170%
10	200%	320%	190%	220%
Média	**217%**	**216%**	**194%**	**222%**

Os valores obtidos no sistema operativo Android OS, se formos a ter em conta os máximos praticados (Tabela 44), levaram à mesma conclusão final de que uma *framework*, a Flutter, é melhor nas fotografias com cerca de 12% a menos de consumos e outra, a React Native, é melhor em vídeos com valores próximos de 30% inferiores.

Tabela 44 - Captura de conteúdos multimédia - Max CPU em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	96,60%	68,90%	82,70%	53,10%
2	110,00%	56,60%	83,30%	76,40%
3	93,70%	60,00%	86,20%	114,00%
4	79,30%	60,00%	103,00%	90,30%
5	73,30%	53,30%	76,60%	80,00%
6	106,00%	54,80%	80,00%	88,80%
7	106,00%	74,10%	110,00%	106,00%
8	106,00%	57,10%	83,30%	80,50%
9	93,30%	66,60%	77,40%	85,20%
10	110,00%	61,20%	74,10%	97,30%
Média	**97,42%**	**61,26%**	**85,66%**	**87,16%**

Nos valores médios (Tabela 45), o cenário é completamente o inverso. A Flutter apresenta valores muito mais otimizados em ambos os conteúdos. Já a React Native, na recolha de vídeos, chega mesmo a ser mais de três vezes pior e, na recolha de imagens, aproximadamente duas vezes.

Tabela 45 - Captura de conteúdos multimédia - Média CPU em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	4,76%	4,11%	2,74%	0,95%
2	5,45%	4,22%	2,64%	1,31%
3	5,42%	4,63%	3,29%	1,45%
4	5,90%	4,16%	2,66%	1,40%
5	5,52%	4,73%	2,04%	1,53%
6	6,01%	4,63%	1,99%	1,44%
7	4,71%	4,71%	2,58%	1,44%
8	5,47%	4,90%	3,48%	1,40%
9	5,61%	4,62%	3,51%	1,52%
10	5,68%	4,13%	3,28%	1,45%
Média	**5,45%**	**4,48%**	**2,82%**	**1,39%**

Tendo em conta que não existe coerência entre os valores máximos e média, isto é, não apontam para o mesmo resultado, são apresentados abaixo dois gráficos para auxiliar a perceção destes resultados.

Relativamente à captura de fotografias, é visível pelo gráfico da Figura 39 que a React Native atinge máximos de consumo superiores e que no decorrer da aplicação mantém consumos superiores também.

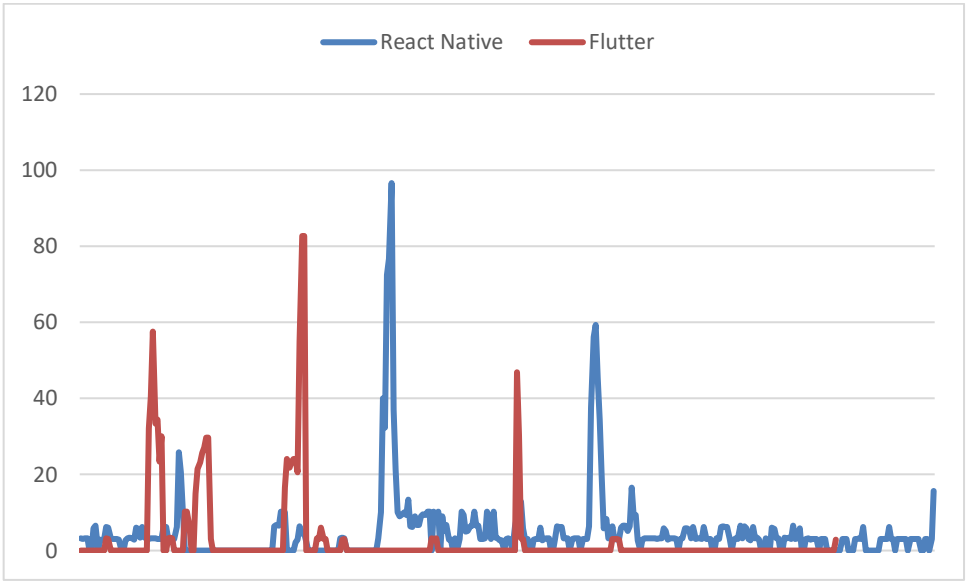


Figura 39 - Captura de Fotografias - gráfico dos consumos (%) de CPU em Android

Na categoria de vídeos, através da análise do gráfico da Figura 40, é percetível que a Flutter atinge um máximo relativamente superior à React Native, mas que no decorrer da aplicação os seus consumos são bastantes reduzidos.

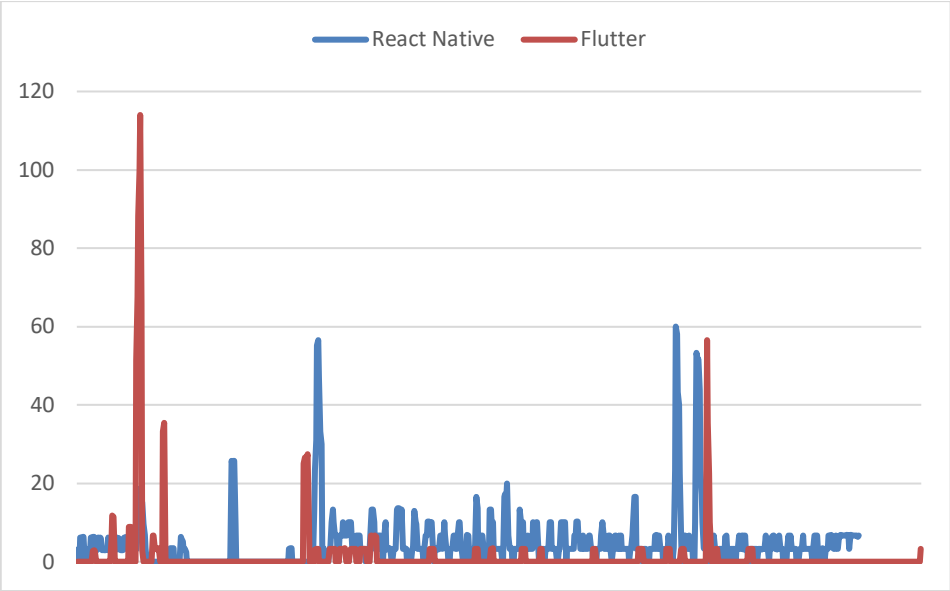


Figura 40 - Captura de Vídeos - gráfico dos consumos (%) de CPU em Android

6.3.2. RAM

Na análise dos consumos de memória RAM, existe uma situação que, embora lógica, não é tão linear e visível noutras métricas, nem sistemas operativos. No iOS, a diferença entre os consumos feitos para a captura de fotografias entre a câmara frontal e a traseira é bastante flagrante. A Tabela 46 procura representar estas diferenças através de um gradiente de cores, de forma a ser ainda mais percetível a distinção existente.

Tabela 46 - Captura de Fotografias - RAM em iOS

	React Native	Flutter
Iteração	Total	Total
1	760,69 MiB	119,41 MiB
2	725,75 MiB	116,24 MiB
3	655,92 MiB	100,37 MiB
4	701,39 MiB	114,59 MiB
5	712,92 MiB	112,13 MiB
6	490,4 MiB	82,7 MiB
7	461,28 MiB	85,29 MiB
8	451,39 MiB	88,21 MiB
9	454,02 MiB	87,44 MiB
10	463,03 MiB	87,18 MiB

Sendo as últimas cinco iterações correspondentes às fotografias capturadas com recurso à câmara frontal, é conclusivo que, neste caso, quanto melhor for a câmara utilizada, maior será o consumo de memória. Contudo, este padrão tão delimitado, não se aplica na captura de vídeos, nem em qualquer outra métrica.

Assim como acontece com os consumos de CPU no sistema operativo iOS, com os consumos de RAM (Tabela 47) chegamos à mesma conclusão: as duas *frameworks* têm uma categoria de multimédia em que são melhores. A React Native tem melhores consumos na captura de vídeos, com cerca de 65% dos consumos da Flutter, enquanto que na captura de imagens, a Flutter tem apenas consumos que rondam os 17% dos praticados pela React Native.

Tabela 47 - Captura de conteúdos multimédia - RAM em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	760,69 MiB	81,22 MiB	119,41 MiB	116,21 MiB
2	725,75 MiB	82,80 MiB	116,24 MiB	136,51 MiB
3	655,92 MiB	82,91 MiB	100,37 MiB	149,02 MiB
4	701,39 MiB	85,06 MiB	114,59 MiB	119,72 MiB
5	712,92 MiB	80,81 MiB	112,13 MiB	124,40 MiB
6	490,40 MiB	86,14 MiB	82,70 MiB	116,38 MiB
7	461,28 MiB	84,56 MiB	85,29 MiB	120,42 MiB
8	451,39 MiB	86,42 MiB	88,21 MiB	144,43 MiB
9	454,02 MiB	85,73 MiB	87,44 MiB	114,94 MiB
10	463,03 MiB	85,45 MiB	87,18 MiB	144,98 MiB
Média	**587,68 MiB**	**84,11 MiB**	**99,36 MiB**	**128,70 MiB**

Relativamente aos consumos praticados no sistema operativo Android, seja através do máximo de valores (Tabela 48), seja pela média (Tabela 49), a conclusão é idêntica: a Flutter é o que tem melhores consumos.

Tabela 48 - Captura de conteúdos multimédia - Max RAM em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	7,70%	6,30%	6,80%	6,10%
2	7,60%	6,30%	6,80%	5,20%
3	7,40%	6,30%	6,80%	5,80%
4	8,10%	6,30%	6,60%	5,70%
5	7,60%	6,30%	6,80%	5,70%
6	7,80%	6,10%	6,70%	5,80%
7	7,60%	6,40%	6,70%	5,80%
8	7,80%	6,30%	6,60%	5,80%
9	7,70%	6,30%	6,70%	5,50%
10	7,50%	6,20%	6,70%	5,70%
Média	**7,68%**	**6,28%**	**6,72%**	**5,71%**

A Flutter, no geral, apresenta um consumo que ronda os 90% dos praticados pela React Native, chegando a ter aproximadamente 12% de consumos a menos relativamente aos máximos atingidos na captura de imagens e cerca de 11% na média de captura de vídeos.

Tabela 49 - Captura de conteúdos multimédia - Média RAM em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	6,45%	5,94%	6,22%	5,61%
2	6,32%	5,95%	6,07%	4,94%
3	6,46%	5,92%	6,21%	5,31%
4	6,82%	5,93%	5,90%	5,21%
5	6,64%	5,99%	6,32%	5,29%
6	6,64%	5,75%	5,89%	5,28%
7	6,57%	6,11%	6,05%	5,22%
8	6,69%	5,94%	5,94%	5,44%
9	6,56%	5,92%	5,98%	5,08%
10	6,55%	5,70%	6,02%	5,27%
Média	**6,57%**	**5,92%**	**6,06%**	**5,26%**

6.3.3. FPS

Assim como observado nas métricas analisadas anteriormente no sistema iOS, a nível de FPS, cada *framework* é a melhor num determinado tipo de conteúdo. A React Native é melhor com vídeos e a Flutter com fotografias.

Relativamente aos picos de consumos (Tabela 50), a situação entre as *frameworks* é idêntica na categoria dos vídeos. No entanto, na fotografia verificou-se uma diferença de FPS a menos por parte da React Native, onde esta renderiza 93% dos FPS praticados pela Flutter.

Tabela 50 - Captura de conteúdos multimédia - Max FPS em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	45	48	48	60
2	41	55	41	49
3	47	57	46	53
4	42	55	47	54
5	39	53	41	52
6	49	52	51	56
7	45	52	51	56
8	43	58	50	48
9	35	50	48	49
10	54	54	52	51
Média	**44,00**	**53,40**	**47,50**	**52,80**

Na média de FPS (Tabela 51), os valores registados na captura de conteúdo vídeo são também relativamente idênticos, onde a Flutter renderiza 96% dos FPS da React Native. Já na captura de imagens, o cenário escala para valores superiores, neste caso, a React Native apenas regista aproximadamente 63% dos FPS renderizados pela Flutter, obtendo assim um pior resultado.

Tabela 51 - Captura de conteúdos multimédia - Média FPS em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	12,4	30,9	27,6	29,8
2	18,5	30,9	27,2	28,3
3	20,1	32,9	32,3	27,5
4	11,7	30,3	25,6	35,4
5	17,9	31,2	23,4	33
6	17	36,2	26,9	34
7	18,1	31,9	29	30,1
8	18,8	31,8	24,5	27
9	18,9	29,1	29,4	33,8
10	18,8	32,1	27,2	25,7
Média	**17,22**	**31,73**	**27,31**	**30,46**

Analisando agora o sistema operativo Android OS (Tabela 52), temos uma situação já frequente, onde a Flutter tem o melhor desempenho. Neste caso, a Flutter é a framework que domina a captura das duas categorias de multimédia. No caso da captura de fotografias, tem somente cerca de 46% dos *Janky frames* praticados pela React Native, e na categoria de vídeos, não registou qualquer *Janky frame*, contra 2,10 de média por parte da React Native.

Tabela 52 - Captura de conteúdos multimédia - FPS - Janky frames em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	5	2	2	0
2	3	3	2	0
3	3	2	2	0
4	3	2	2	0
5	5	2	1	0
6	2	2	3	0
7	4	2	2	0
8	14	2	2	0
9	3	2	3	0
10	4	2	2	0
Média	**4,60**	**2,10**	**2,10**	**0,00**

6.3.4. Câmara– React Native vs Flutter

Comparando o conjunto de todos os valores correspondentes a esta funcionalidade, é possível chegar a uma conclusão sobre as duas *frameworks* em estudo. A Flutter é a *framework* que apresenta os melhores resultados na captura de fotografias tanto em iOS (Tabela 53) como Android OS (Tabela 54).

Tabela 53 - React Native vs Flutter - Captura de Fotografias em iOS

Iteração	React Native	Flutter
CPU	217%	194%
RAM	587,68 MiB	99,36 MiB
FPS - Max	44,00	47,50
FPS - Média	17,22	27,31
Pontos	**0**	**4**

Tabela 54 - React Native vs Flutter - Captura de Fotografias em Android

Iteração	React Native	Flutter
CPU - Max	97,42%	85,66%
CPU - Média	5,45%	2,82%
RAM - Max	7,68%	6,72%
RAM - Média	6,57%	6,06%
FPS	4,60	2,10
Pontos	**0**	**5**

Relativamente ao conteúdo de vídeo, a React Native obtém a vantagem em todas as métricas no iOS (Tabela 55) e no Android OS apenas nos consumos máximos de CPU (Tabela 56). Nas restantes métricas do Android OS, é a Flutter que tem os melhores resultados.

Tabela 55 - React vs Flutter - Captura de Vídeo em iOS

Iteração	React Native	Flutter
CPU	216%	222%
RAM	84,11 MiB	128,70 MiB
FPS - Max	53,40	52,80
FPS - Média	31,73	30,46
Pontos	**4**	**0**

Tabela 56 - React vs Flutter - Captura de Vídeo em Android

Iteração	React Native	Flutter
CPU - Max	61,26%	87,16%
CPU - Média	4,48%	1,39%
RAM - Max	6,28%	5,71%
RAM - Média	5,92%	5,26%
FPS	2,10	0,00
Pontos	**1**	**4**

Concluindo, com recurso aos valores apresentados, das quatro aplicações resultantes nesta funcionalidade, a Flutter é a *framework* que tem melhores resultados no geral, sendo o melhor em três das quatro aplicações.

6.4. Acesso a conteúdo multimédia local

6.4.1. CPU

No acesso a conteúdos multimédia nos dispositivos iOS (Tabela 57), em ambas as situações, a Flutter tem melhores resultados, chegando a consumir apenas cerca de 75% do processamento consumido pela React Native.

Tabela 57 - Acesso a conteúdo multimédia local - CPU em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	190%	220%	170%	180%
2	220%	220%	140%	250%
3	210%	150%	200%	200%
4	210%	250%	160%	210%
5	240%	240%	150%	190%
6	270%	240%	200%	180%
7	160%	200%	150%	150%
8	280%	200%	160%	240%
9	280%	280%	200%	120%
10	260%	200%	210%	170%
Média	**232%**	**220%**	**174%**	**189%**

Identicamente ao que acontece na funcionalidade anterior, no sistema Android OS, os consumos de CPU obtêm resultados diferentes dependendo da perspetiva de análise: através dos máximos atingidos observamos um resultado (Tabela 58) e através da média de valores (Tabela 59), outro.

Tabela 58 - Acesso a conteúdo multimédia local - Max CPU em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	103,00%	51,70%	96,80%	80,00%
2	106,00%	53,30%	113,00%	108,00%
3	90,60%	65,50%	109,00%	84,30%
4	110,00%	55,10%	76,60%	58,00%
5	124,00%	54,50%	113,00%	84,80%
6	124,00%	52,90%	78,30%	176,00%
7	87,00%	54,50%	118,00%	75,00%
8	106,00%	44,80%	75,00%	78,10%
9	113,00%	72,70%	94,10%	72,90%
10	67,60%	64,50%	83,30%	97,20%
Média	**103,12%**	**56,95%**	**95,71%**	**91,43%**

Na seleção de fotografia, seguindo os valores médios, a Flutter tem consumos próximos de 76% dos feitos pela React Native e cerca de 93% dos valores máximos. Já na seleção de vídeos, a Flutter tem um consumo inferior de aproximadamente 45% pelos valores médios, no entanto, relativamente aos máximos, tem perto de 161% a mais.

Tabela 59 - Acesso a conteúdo multimédia local - Média CPU em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	5,68%	4,68%	4,57%	2,93%
2	5,66%	4,71%	4,05%	3,10%
3	5,47%	5,28%	4,34%	3,10%
4	5,62%	4,98%	3,43%	1,93%
5	5,50%	5,08%	4,44%	3,38%
6	5,64%	5,79%	3,97%	2,78%
7	5,28%	4,97%	4,93%	2,65%
8	5,56%	4,43%	4,34%	2,88%
9	5,61%	6,35%	4,08%	2,78%
10	5,24%	5,72%	3,88%	3,10%
Média	**5,53%**	**5,20%**	**4,20%**	**2,86%**

Tendo em conta que existe uma maior diferença entre a média de valores e os picos de consumos na tarefa de seleção de vídeos, é apresentado o gráfico da Figura 41, de forma a obtermos uma melhor perceção do que acontece.

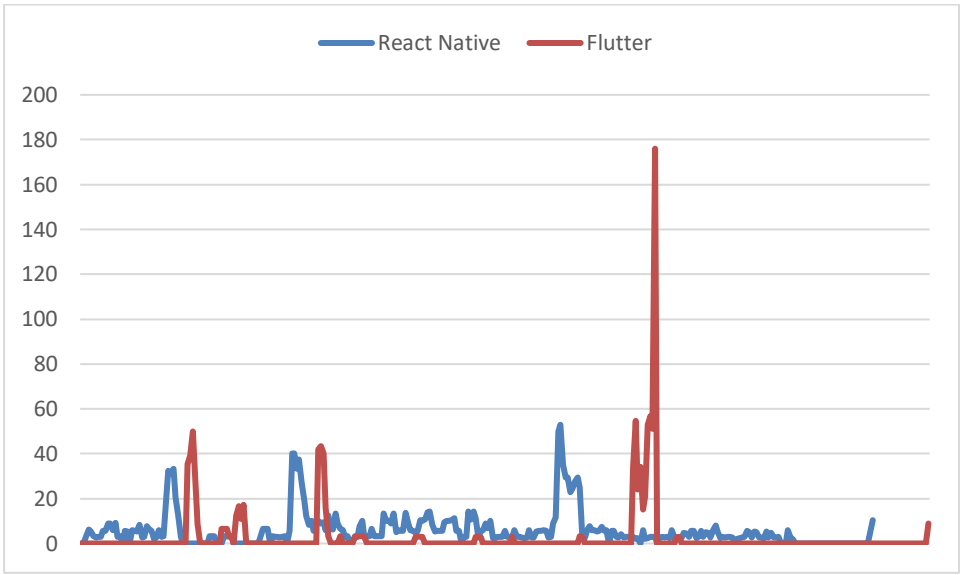


Figura 41 - Acesso a Vídeos - gráfico dos consumos (%) de CPU em Android

Com o recurso do gráfico, criado a partir de uma das iterações do teste, é de fácil perceção que apesar dos máximos atingidos pela Flutter serem bastante superiores, os valores

exercidos no decorrer da aplicação são mais reduzidos relativamente ao pico de consumo e ainda inferiores aos praticados pela React Native.

6.4.2. RAM

Na captura de conteúdos multimédia, as categorias estavam divididas pelas *frameworks*. Nesta funcionalidade de acesso aos conteúdos multimédia não é o que acontece. Neste caso, com uma vantagem próxima de 7% a menos no acesso a imagens e cerca de 16% no acesso a vídeos, a Flutter tem vantagem no que toca ao consumo de memória RAM em iOS (Tabela 60).

Tabela 60 - Acesso a conteúdo multimédia local - RAM em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	101,53 MiB	99,53 MiB	94,54 MiB	87,20 MiB
2	102,81 MiB	105,53 MiB	92,60 MiB	82,22 MiB
3	100,99 MiB	101,94 MiB	92,35 MiB	92,24 MiB
4	100,83 MiB	105,65 MiB	92,52 MiB	83,06 MiB
5	101,58 MiB	102,72 MiB	98,76 MiB	85,46 MiB
6	102,40 MiB	103,03 MiB	91,69 MiB	86,07 MiB
7	103,39 MiB	103,04 MiB	93,54 MiB	85,48 MiB
8	100,81 MiB	102,67 MiB	95,76 MiB	91,26 MiB
9	101,01 MiB	102,60 MiB	97,67 MiB	85,04 MiB
10	100,74 MiB	101,93 MiB	94,93 MiB	87,65 MiB
Média	**101,61 MiB**	**102,86 MiB**	**94,44 MiB**	**86,57 MiB**

Ao analisarmos os valores obtidos no Android OS, tanto pelos máximos exercidos (Tabela 61) como pela média de valores (Tabela 62), a distinção entre as *frameworks* praticamente não existe.

Tabela 61 - Acesso a conteúdo multimédia local - Max RAM em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	7,60%	6,30%	6,80%	5,50%
2	7,70%	6,30%	6,90%	5,50%
3	7,70%	6,40%	6,40%	5,60%
4	7,70%	6,30%	6,80%	5,50%
5	7,60%	6,30%	6,40%	5,60%
6	7,70%	6,20%	6,80%	5,50%
7	7,40%	6,30%	6,40%	5,50%
8	7,70%	6,30%	6,70%	5,50%
9	7,70%	6,30%	6,80%	5,50%
10	7,60%	6,40%	6,70%	5,50%
Média	**7,64%**	**6,31%**	**6,67%**	**5,52%**

Com uma percentagem próxima dos 13% de consumos inferiores em ambos os tipos de conteúdo e tanto em máximos como média de consumos, a Flutter é a *framework* com melhor *performance* no consumo de RAM.

Tabela 62 - Acesso a conteúdo multimédia local - Média RAM em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	6,66%	6,04%	6,07%	5,19%
2	6,72%	6,05%	6,04%	5,18%
3	6,66%	6,06%	5,17%	5,21%
4	6,72%	6,01%	6,09%	5,18%
5	6,73%	6,02%	5,35%	5,22%
6	6,77%	5,98%	6,00%	5,18%
7	6,66%	6,08%	5,13%	5,21%
8	6,59%	6,10%	6,05%	5,18%
9	6,70%	6,00%	6,08%	5,17%
10	6,63%	6,07%	5,98%	5,21%
Média	**6,68%**	**6,04%**	**5,80%**	**5,19%**

6.4.3. FPS

Na renderização de *frames* por segundo, como acontece nas restantes métricas em iOS, o resultado é diferente do que na *funcionalidade* anterior. Com valores não muito distintos entre categorias, nos máximos a Flutter obtém uma maior quantidade de FPS, próximo dos 12% em relação à React Native (Tabela 63).

Tabela 63 - Acesso a conteúdo multimédia local - Max FPS em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	33	39	42	42
2	45	49	55	53
3	46	47	52	44
4	51	50	57	47
5	45	37	42	59
6	45	51	52	54
7	48	51	56	49
8	47	41	53	52
9	44	46	52	56
10	47	49	44	59
Média	**45,10**	**46,00**	**50,50**	**51,50**

Já relativamente à média dos FPS (Tabela 64), ainda em vantagem a Flutter tem uma *performance* cerca de 10% superior no acesso a imagens enquanto no acesso a vídeos apenas tem uma vantagem perto de 4%.

Tabela 64 - Acesso a conteúdo multimédia local - Média FPS em iOS

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	17,5	23,3	12,3	23,6
2	18,2	22,2	21,2	28
3	17,2	25,5	18,9	17,8
4	18,2	23,4	19,2	25,7
5	18,3	23,8	20,09	27,3
6	18,4	27,2	22,9	26
7	18,2	25,5	20,7	25,8
8	15,3	22,4	20,8	27,4
9	17,9	24,5	20,9	26,3
10	17,6	25,2	17,9	24,2
Média	**17,68**	**24,30**	**19,49**	**25,21**

Relativamente aos *Janky frames* praticados no sistema operativo Android OS (Tabela 65), mais uma vez é a Flutter que obtém melhores resultados, chegando a ter mais de seis vezes de *Janky frames* a menos que a React Native no acesso a vídeos.

Tabela 65 - Acesso a conteúdo multimédia local - FPS - Janky *frames* em Android

	React Native		Flutter	
Iteração	**Fotografia**	**Vídeo**	**Fotografia**	**Vídeo**
1	5	3	1	0
2	4	3	2	2
3	6	3	1	3
4	6	6	2	3
5	6	3	2	0
6	7	3	1	1
7	6	16	3	0
8	5	3	3	0
9	7	16	2	1
10	6	14	1	1
Média	**5,80**	**7,00**	**1,80**	**1,10**

6.4.4. Acesso a conteúdo multimédia local – React Native vs Flutter

Como foi percetível no decorrer da análise dos dados, a Flutter é a *framework* que melhor *performance* apresenta no acesso à galeria de imagens, tanto no sistema operativo iOS (Tabela 66) como no Android OS (Tabela 67).

Tabela 66 - React Native vs Flutter - Acesso a Fotografia em iOS

Iteração	React Native	Flutter
CPU	232%	174%
RAM	101,61 MiB	94,44 MiB
FPS - Max	45,10	50,50
FPS - Média	17,68	19,49
Pontos	**0**	**4**

Tabela 67 - React Native vs Flutter - Acesso a Fotografia em Android

Iteração	React Native	Flutter
CPU - Max	103,12%	95,71%
CPU - Média	5,53%	4,20%
RAM - Max	7,64%	6,67%
RAM - Média	6,68%	5,80%
FPS	5,80	1,80
Pontos	**0**	**5**

No sistema operativo iOS, todas as métricas são dominadas pela Flutter (Tabela 68). No Android OS (Tabela 69), a React Native, apesar de ter máximos de consumo de CPU inferiores, não tem qualquer vantagem nas restantes métricas.

Tabela 68 - React Native vs Flutter - Acesso a Vídeos em iOS

Iteração	React Native	Flutter
CPU	220%	189%
RAM	102,86 MiB	86,57 MiB
FPS - Max	46,00	51,50
FPS - Média	24,30	25,21
Pontos	**0**	**4**

Tabela 69 - React Native vs Flutter - Acesso a Vídeos em Android

Iteração	React Native	Flutter
CPU - Max	56,95%	91,43%
CPU - Média	5,20%	2,86%
RAM - Max	6,31%	5,52%
RAM - Média	6,04%	5,19%
FPS	7,00	1,10
Pontos	**1**	**4**

6.5. Acesso a conteúdo multimédia remoto

6.5.1. CPU

Na funcionalidade de *streaming*, em que é reproduzido um vídeo de 60 FPS, relativamente ao consumo de CPU no sistema operativo iOS (Tabela 70), a Flutter pratica metade dos consumos feitos pela React Native, apresentando assim um melhor desempenho.

Tabela 70 - Acesso a conteúdo multimédia remoto CPU em iOS

Iteração	React Native	Flutter
1	270%	180%
2	300%	200%
3	250%	120%
4	270%	120%
5	260%	180%
6	280%	110%
7	260%	90%
8	340%	120%
9	250%	120%
10	300%	130%
Média	**278%**	**137%**

Já no sistema operativo Android OS (Tabela 71), a React Native tem cerca de 30% de consumos inferiores relativamente aos máximos atingidos pela Flutter e cerca de 35% na média de consumos.

Tabela 71 - Acesso a conteúdo multimédia remoto CPU em Android

	React Native		Flutter	
Iteração	**Média**	**Max**	**Média**	**Max**
1	22,78%	76,60%	39,01%	110,00%
2	28,94%	96,50%	43,78%	110,00%
3	28,69%	100,00%	40,89%	112,00%
4	26,20%	103,00%	37,33%	113,00%
5	25,45%	80,00%	42,82%	115,00%
6	28,35%	79,30%	39,02%	113,00%
7	23,77%	69,40%	42,91%	110,00%
8	26,63%	79,30%	39,92%	132,00%
9	24,89%	76,60%	43,80%	171,00%
10	32,51%	91,10%	38,66%	117,00%
Média	**26,82%**	**85,18%**	**40,81%**	**120,30%**

6.5.2. RAM

Em relação à memória RAM, no sistema operativo iOS (Tabela 72), temos a React Native com consumos significativamente inferiores, de cerca de 10% dos consumos que são praticados pela Flutter.

Tabela 72 - Acesso a conteúdo multimédia remoto - RAM em iOS

Iteração	React Native	Flutter
1	25,52 MiB	232,35 MiB
2	24,86 MiB	229,28 MiB
3	31,53 MiB	218,07 MiB
4	28,07 MiB	236,77 MiB
5	24,18 MiB	236,90 MiB
6	26,49 MiB	234,08 MiB
7	25,42 MiB	313,12 MiB
8	24,99 MiB	244,74 MiB
9	28,77 MiB	245,16 MiB
10	22,70 MiB	247,38 MiB
Média	**26,25 MiB**	**243,79 MiB**

No caso do Android OS (Tabela 73), é a Flutter que tem melhores consumos, cerca de 88% dos praticados pela React Native.

Tabela 73 - Acesso a conteúdo multimédia remoto - RAM em Android

	React Native		Flutter	
Iteração	**Média**	**Max**	**Média**	**Max**
1	6,13%	6,40%	5,39%	5,50%
2	6,21%	6,40%	5,67%	5,80%
3	6,28%	6,50%	5,44%	5,50%
4	6,25%	6,50%	5,47%	5,60%
5	6,23%	6,50%	5,67%	5,80%
6	6,24%	6,40%	5,46%	5,50%
7	6,31%	6,50%	5,62%	5,70%
8	6,24%	6,40%	5,41%	5,50%
9	6,18%	6,40%	5,42%	5,50%
10	6,21%	6,40%	5,42%	5,50%
Média	**6,23%**	**6,44%**	**5,50%**	**5,59%**

6.5.3. FPS

A seleção de um vídeo de 60 FPS teve o propósito de perceber se as *frameworks* tinham a capacidade de atingir esses valores na renderização de conteúdo multimédia. Como se pode comprovar pela Tabela 74, ambas as *frameworks* atingiram esses valores. No entanto, a média de FPS não foi muito próxima dos 60. A React Native, sendo a que teve melhores resultados, obteve uma média a rondar os 49 FPS, cerca de 15% a mais que a Flutter.

Tabela 74 - Acesso a conteúdo multimédia remoto - FPS em iOS

	React Native		Flutter	
Iteração	**Média**	**Max**	**Média**	**Max**
1	51,3	60	37,4	60
2	53,4	60	35,5	60
3	43,4	60	38	60
4	54,7	60	46,3	60
5	50,8	60	46,8	60
6	52,3	60	47,8	60
7	53,8	60	46,1	60
8	51,2	60	47,8	60
9	30,1	60	40,6	60
10	51,3	60	41,4	60
Média	**49,23**	**60**	**42,77**	**60**

Na renderização do mesmo conteúdo no sistema operativo Android OS (Tabela 75), a React Native apresenta um cenário bastante negativo relativamente à Flutter. Com um valor de *Janky frames* próximo de 63 vezes superior que a Flutter, a React Native atinge o maior volume de *Janky frames* em todas as funcionalidades analisadas até ao momento.

Tabela 75 - Acesso a conteúdo multimédia remoto - FPS - Janky *frames* em Android

Iteração	**React Native**	**Flutter**
1	101	0
2	88	0
3	6	5
4	5	4
5	107	0
6	138	1
7	11	0
8	15	3
9	253	1
10	156	0
Média	**88**	**1,4**

6.5.4. Acesso a conteúdo multimédia remoto – React Native vs Flutter

Na conclusão final, relativamente a esta funcionalidade de *streaming*, ambas as *frameworks* estão em igualdade em termos de pontuações. Isto não acontece devido à *performance* relativamente às métricas, mas sim pelo facto de que cada uma é dominante num sistema operativo distinto. Assim, nesta funcionalidade, a React Native mostra-se mais compatível com o sistema operativo iOS (Tabela 76), enquanto que a Flutter é melhor em Android OS (Tabela 77).

Tabela 76 - React Native vs Flutter - Acesso a conteúdo multimédia remoto em iOS

Iteração	React Native	Flutter
CPU	278%	137%
RAM	26,25 MiB	243,79 MiB
FPS - Max	60	60
FPS - Média	49,23	42,77
Pontos	**3**	**2**

Tabela 77 - React Native vs Flutter - Acesso a conteúdo multimédia remoto em Android

Iteração	React Native	Flutter
CPU - Max	85,18%	120,30%
CPU - Média	26,82%	40,81%
RAM - Max	6,44%	5,59%
RAM - Média	6,23%	5,50%
FPS	88	1,4
Pontos	**2**	**3**

6.6. Animações

6.6.1. Tempo de execução

Na métrica de tempo de execução que, nesta funcionalidade, é responsável por cronometrar o tempo que cada *framework* consome a carregar e renderizar por completo diversas imagens animadas, a Flutter é a mais rápida tanto em iOS (Tabela 78), como em Android OS (Tabela 79), com uma fração próxima de 11% do tempo gasto pela React Native no sistema iOS e de 16% no Android OS.

Tabela 78 - Animações - Tempo de Execução em iOS

Iteração	React Native	Flutter
1	355 ms	38 ms
2	288 ms	38 ms
3	290 ms	39 ms
4	347 ms	35 ms
5	341 ms	36 ms
6	342 ms	35 ms
7	344 ms	35 ms
8	347 ms	36 ms
9	344 ms	37 ms
10	347 ms	38 ms
Média	**334,5 ms**	**36,7 ms**

Tabela 79 - Animações - Tempo de Execução em Android

Iteração	React Native	Flutter
1	788 ms	116 ms
2	703 ms	115 ms
3	681 ms	116 ms
4	679 ms	122 ms
5	845 ms	114 ms
6	878 ms	113 ms
7	723 ms	116 ms
8	665 ms	112 ms
9	667 ms	113 ms
10	668 ms	119 ms
Média	**729,7 ms**	**115,6 ms**

6.6.2. CPU

Dispondo de uma vantagem próxima dos 6% de consumo inferior, a React Native é a *framework* que melhores resultados tem no sistema operativo iOS, relativamente ao consumo de CPU (Tabela 80).

Tabela 80 - Animações - CPU em iOS

Iteração	React Native	Flutter
1	360%	530%
2	370%	420%
3	580%	570%
4	540%	530%
5	520%	520%
6	540%	550%
7	490%	580%
8	510%	560%
9	530%	520%
10	510%	490%
Média	**495%**	**527%**

No entanto, ao analisarmos os valores praticados no sistema operativo Android OS (Tabela 81), a Flutter tem consumos inferiores à React Native, tanto a nível de máximos atingidos, com cerca de 74% dos valores da React Native, como na média dos registos, onde consome perto de 76% do processamento requerido pela React Native.

Tabela 81 - Animações - CPU em Android

	React Native		Flutter	
Iteração	Média	Max	Média	Max
1	38,37%	312,00%	32,01%	209,00%
2	42,60%	362,00%	29,33%	187,00%
3	33,78%	234,00%	30,77%	181,00%
4	40,25%	278,00%	30,78%	233,00%
5	42,68%	309,00%	29,81%	221,00%
6	34,12%	216,00%	24,86%	191,00%
7	36,00%	255,00%	34,05%	278,00%
8	37,16%	294,00%	35,90%	230,00%
9	42,09%	351,00%	10,89%	234,00%
10	37,08%	218,00%	26,04%	190,00%
Média	**38,41%**	**282,90%**	**28,44%**	**215,40%**

6.6.3. RAM

À semelhança de outras métricas e de outras funcionalidades também, relativamente ao consumo de memória RAM no sistema operativo iOS (Tabela 82), a React Native obtém vantagem sobre a Flutter, com apenas uma fração próxima de 40% dos consumos que esta última pratica.

Tabela 82 - Animações - RAM em iOS

Iteração	React Native	Flutter
1	317,32 MiB	552,50 MiB
2	192,08 MiB	540,92 MiB
3	159,83 MiB	539,64 MiB
4	151,69 MiB	316,54 MiB
5	156,56 MiB	305,47 MiB
6	169,39 MiB	568,98 MiB
7	194,06 MiB	609,04 MiB
8	165,72 MiB	366,64 MiB
9	169,12 MiB	277,50 MiB
10	143,30 MiB	423,39 MiB
Média	**181,907**	**450,062**

Todavia, a vantagem da React Native não se observa no sistema operativo Android OS (Tabela 83), onde a Flutter tem aproximadamente 90% dos consumos praticados pela React Native, tanto a nível de picos de consumos como de média.

Tabela 83 - Animações - RAM em Android

	React Native		Flutter	
Iteração	**Média**	**Max**	**Média**	**Max**
1	6,82%	8,10%	6,29%	7,30%
2	6,90%	8,00%	6,24%	7,20%
3	6,84%	8,20%	6,20%	7,30%
4	6,80%	8,10%	6,31%	7,40%
5	6,87%	8,20%	6,38%	7,40%
6	7,03%	8,20%	6,13%	7,30%
7	7,04%	8,40%	6,22%	7,30%
8	6,94%	8,20%	6,14%	7,20%
9	6,83%	8,10%	6,08%	7,60%
10	6,91%	8,20%	6,12%	7,30%
Média	**6,90%**	**8,17%**	**6,21%**	**7,33%**

6.6.4. FPS

Durante este teste, era necessária a renderização de 200 imagens com diversas animações. Sendo esta a funcionalidade produzida neste trabalho que possivelmente exige mais da capacidade de renderização, pode-se concluir que a Flutter é a framework que melhor lida com efeitos gráficos, como é o caso das animações. No sistema operativo iOS (Tabela 84), a vantagem que a Flutter tem sobre a React Native é cerca de 30% a mais na média de FPS e perto de 35% a mais no máximo de FPS.

Tabela 84 - Animações - FPS em iOS

	React Native		Flutter	
Iteração	**Média**	**Max**	**Média**	**Max**
1	16,1	57	19,4	55
2	8,6	38	19,5	52
3	15,3	37	17,9	55
4	4,5	24	13,6	53
5	6,3	25	7,9	49
6	15,9	38	18,5	52
7	17,5	58	19	51
8	10	35	10,8	56
9	11,2	37	12,3	52
10	11,6	39	12,1	50
Média	**11,7**	**38,8**	**15,1**	**52,5**

Comparativamente ao que acontece no iOS, no caso do Android OS (Tabela 85) a Flutter tem uma melhor *performance* na renderização de FPS. A React Native, com um valor de *Janky frames* superior a 153 vezes a mais em relação à Flutter, apresenta assim a pior situação registada entre as *frameworks* de todas as perspetivas analisadas neste trabalho.

Tabela 85 - Animações - FPS - Janky *frames* em Android

Iteração	**React Native**	**Flutter**
1	22	0
2	179	0
3	31	0
4	21	0
5	25	0
6	176	0
7	189	1
8	171	0
9	135	5
10	125	1
Média	**107,4**	**0,7**

6.6.5. Animações – React Native vs Flutter

Procedendo à análise geral dos valores obtidos nesta funcionalidade das animações, no iOS (Tabela 86), a Flutter tem a vantagem por apenas uma métrica, mas no sistema operativo Android OS (Tabela 87) é a *framework* que melhores resultados tem em todas as métricas avaliadas.

Tabela 86 - React Native vs Flutter - Animações em iOS

Iteração	React Native	Flutter
Tempo de Execução	334,5 ms	36,7 ms
CPU	495%	527%
RAM	181,907	450,062
FPS - Max	38,8	52,5
FPS - Média	11,7	15,1
Pontos	**2**	**3**

Tabela 87 - React Native vs Flutter - Animações em Android

Iteração	React Native	Flutter
Tempo de Execução	729,7 ms	115,6 ms
CPU - Max	282,90%	215,40%
CPU - Média	38,41%	28,44%
RAM - Max	8,17%	7,33%
RAM - Média	6,90%	6,21%
FPS	107,4	0,7
Pontos	**0**	**6**

6.7. Resumo de resultados

Tendo em conta que, no decorrer da avaliação das aplicações criadas para cada funcionalidade, os resultados foram cotados através de pontos, é apresentada na Tabela 88 uma visão mais abrangente da *performance* das *frameworks* com recurso a esses pontos.

Tabela 88 - Resumo de Resultados - Pontos por sistema operativo, *framework* e funcionalidade

	iOS		Android	
Funcionalidade	**React Native**	**Flutter**	**React Native**	**Flutter**
Launch	2	1	2	3
Listas Locais	9	6	3	15
Listas Remotas	3	12	8	10
Captura de Imagens	0	4	0	5
Captura de Vídeos	4	0	1	4
Acesso a Imagens	0	4	0	5
Acesso a Vídeos	0	4	1	4
Streaming	3	2	2	3
Animações	2	3	0	6
Total	**23**	**36**	**17**	**55**

Tal como é apresentado na tabela, que utiliza o mesmo esquema de cores usado anteriormente para auxiliar na perceção dos resultados, a Flutter é a *framework* que melhores resultados apresenta em ambos os sistemas operativos, cerca de 157% melhor no iOS e mais de 3 vezes em Android OS. Neste último, não há grandes informações adicionais a acrescentar, visto que todas as métricas são dominadas pela Flutter. No entanto, do lado do iOS, temos a React Native que, embora com menos pontos, sobressai com vantagem em 4 das 9 funcionalidades.

Efetuando uma análise mas na perspetiva das métricas (Tabela 89), em termos de tempos de execução, ambas as *frameworks* apresentam a mesma pontuação, independentemente do sistema operativo. Relativamente ao iOS, é visível que a React Native é melhor no que diz respeito aos consumos de RAM. No entanto, se compararmos os valores no Android OS, a Flutter tem melhor *performance* em todas as métricas, excluindo o tempo de execução que, como já abordado anteriormente, tem uma *performance* equivalente.

Tabela 89 - Resumo de Resultados - Pontos por sistema operativo, *framework* e métrica

	iOS		Android	
Métricas	**React Native**	**Flutter**	**React Native**	**Flutter**
Tempo de Execução	4	4	4	4
CPU	2	11	10	16
RAM	10	3	3	23
FPS	7	18	0	12
Total	**23**	**36**	**17**	**55**

Analisando os resultados globais, sem fazer distinção entre sistemas operativos, obtemos a perspetiva de valores apresentada na Tabela 90. Com a apresentação dos valores desta forma, podemos observar os valores de cada *framework* e concluir igualmente que a Flutter é a que apresenta melhores resultados globais em oito das nove funcionalidades consideradas. A React Native apenas tem uma pequena vantagem na captura de vídeos, tendo um nível equivalente no *launch* e *streaming*. No total de pontos, a Flutter conquista precisamente duas vezes mais do que os obtidos pela React Native.

Tabela 90 - Resumo de Resultados - Pontos por *framework* e funcionalidade

Funcionalidade	React Native	Flutter
Launch	4	4
Listas Locais	12	21
Listas Remotas	11	22
Captura de Imagens	0	9
Captura de Vídeos	5	4
Acesso a Imagens	0	9
Acesso a Vídeos	1	8
Streaming	5	5
Animações	2	9
Total	**40**	**91**

Através da Tabela 91, também do mesmo ponto de vista, mas relativamente às métricas, a React Native apenas obtém uma *performance* equivalente nos tempos de execução, enquanto que todas as outras métricas são largamente dominadas pela Flutter.

Tabela 91 - Resumo de Resultados - Pontos por *framework* e métrica

Métricas	React Native	Flutter
Tempo de Execução	8	8
CPU	12	27
RAM	13	26
FPS	7	30
Total	**40**	**91**

De forma a fazer uma comparação ainda mais imparcial, a pontuação obtida pelas *frameworks* foi alavancada pelo valor da divisão do maior valor obtido na métrica pelo menor. Nos casos em que o menor valor é zero, foi considerada a diferença entre estes. Deste modo, a pontuação obtida irá refletir com maior precisão a real diferença existente entre *frameworks* e sistemas operativos.

Na Tabela 92, estão apresentadas as pontuações alavancadas obtidas por sistema operativo e *framework*. Através desta representação é percetível que a diferença entre as tecnologias no sistema operativo iOS é inferior à analisada acima por pontos sem alavancagem. A React Native passa de cerca de 64% dos pontos da Flutter para cerca de 90% no sistema iOS. No entanto, no sistema operativo Android OS, a diferença entre as *frameworks* intensificasse, passando assim a React Native de cerca de 31% dos pontos da Flutter para aproximadamente apenas 6%.

Tabela 92 - Resumo de Resultados – Pontos alavancados por sistema operativo, *framework* e funcionalidade

	iOS		Android	
Funcionalidade	**React Native**	**Flutter**	**React Native**	**Flutter**
Launch	3,50	1,05	3,60	4,09
Listas Locais	21,62	11,40	8,68	85,12
Listas Remotas	15,36	20,52	10,08	116,80
Captura de Imagens	0,00	9,70	0,00	7,49
Captura de Vídeos	4,61	0,00	1,42	7,55
Acesso a Imagens	0,00	4,63	0,00	7,91
Acesso a Vídeos	0,00	4,51	1,61	10,49
Streaming	11,44	3,03	2,93	65,14
Animações	3,54	11,76	0,00	164,63
Total	**60,07**	**66,60**	**28,32**	**469,22**

Efetuando a análise por métrica, com recurso à Tabela 93, apesar da React Native ter pior performance em todas as métricas excluindo uma, este mostra ser significativamente melhor na gestão de memória RAM no sistema operativo iOS. Ainda assim a Flutter também tem uma grande vantagem não só na gestão da memória RAM como também nos FPS do sistema operativo Android OS.

Tabela 93 - Resumo de Resultados - Pontos alavancados por sistema operativo, *framework* e métrica

	iOS		Android	
Métricas	**React Native**	**Flutter**	**React Native**	**Flutter**
Tempo de Execução	11,45	13,92	10,04	16,84
CPU	2,09	15,94	14,45	25,64
RAM	38,98	8,18	3,83	25,51
FPS	7,54	28,56	0,00	401,24
Total	**60,07**	**66,60**	**28,32**	**469,22**

Procedendo à comparação dos valores nas aplicações desenvolvidas excluindo a divisão por sistemas operativos (Tabela 94), a única funcionalidade em que a React Native mostra dominar com melhor *performance* é o *launch* das aplicações. Com uma *performance* média de mais de seis vezes superior, a Flutter mais uma vez reforça o seu domínio no desenvolvimento multiplataforma relativamente à React Native.

Tabela 94 - Resumo de Resultados - Pontos alavancados por *framework* e funcionalidade

Funcionalidade	React Native	Flutter
Launch	7,11	5,15
Listas Locais	30,30	96,52
Listas Remotas	25,44	137,31
Captura de Imagens	0,00	17,19
Captura de Vídeos	6,03	7,55
Acesso a Imagens	0,00	12,54
Acesso a Vídeos	1,61	15,00
Streaming	14,37	68,17
Animações	3,54	176,39
Total	**88,39**	**535,82**

Excluindo os sistemas operativos, para a comparação mais direta das *frameworks* a nível de métricas (Tabela 95), é explícito que o ponto forte da React Native em relação à Flutter é a gestão de memoria RAM que este faz no sistema operativo iOS. Fora a métrica do consumo de memória volátil, RAM, a Flutter tem uma vantagem acentuada em relação à React Native.

Tabela 95 - Resumo de Resultados - Pontos alavancados por *framework* e métrica

Métricas	React Native	Flutter
Tempo de Execução	21,50	30,76
CPU	16,55	41,57
RAM	42,80	33,69
FPS	7,54	429,79
Total	**88,39**	**535,82**

7. Conclusão

Sendo este o último capítulo, pretende-se fazer um balanço de todo o trabalho realizado, através de uma análise crítica das aprendizagens que foram adquiridas durante o mesmo.

A concluir este ponto será ainda referenciado o trabalho pretendido para o futuro, por forma a completar todo este processo que foi desenvolvido neste trabalho.

7.1. Análise crítica

Na fase conclusiva deste relatório, pretendo realçar determinados pontos que se revelaram importantes no decorrer deste trabalho. O objetivo inicial deste trabalho foi o enriquecimento de uma área de grande evolução contínua nos últimos tempos. No entanto, no decorrer da análise do contexto existencial a nível de estudos já realizados na área, foi detetada uma necessidade mais aprimorada.

Os diversos estudos existentes relativamente a avaliações de alternativas tecnológicas disponíveis no mercado para o desenvolvimento de aplicações multiplataforma apresentaram algumas inconsistências nos procedimentos levados a cabo, incluindo estudos que não referenciavam o processo que foi usado ou a importância de diversos pormenores como, por exemplo, o *Release Mode* ou outro equivalente nas aplicações a serem avaliadas.

Esta situação veio reforçar a ideia de que existia a necessidade de ser proposto um processo de análise comparativa destas tecnologias, por forma a auxiliar todas as organizações que tenham de tomar uma decisão acerca da *framework* a adotarem para os seus projetos.

No decorrer deste trabalho existiram diversos pontos que vieram criar uma maior dificuldade do que a esperada. Tendo em conta que o objetivo principal deste projeto era definir um processo de comparação entre várias *frameworks* de desenvolvimento de aplicações móveis, não era esperado que um dos maiores desafios fosse relacionado com as ferramentas de monitorização de consumos por parte das aplicações.

Perante esta situação, compreendi a razão pela qual determinados estudos optavam por soluções menos credíveis para a comparação das suas aplicações, sendo esta a solução mais fácil perante as circunstâncias. Com o objetivo de levar a cabo uma avaliação mais credível, assim como fazer sempre uso de ferramentas oficiais e confiáveis, tomei a decisão de utilizar

diversas ferramentas de maior complexidade de utilização e criar *scripts* para desenhar um procedimento de testes generalizado e aplicável às várias funcionalidades em questão. Desta forma, foi proposto todo um processo de automatização de testes e registo de métricas de avaliação, combinando e recorrendo sempre às ferramentas oficiais dos fabricantes dos sistemas operativos Android OS e iOS.

A comparação entre as *frameworks* React Native e Flutter foi executada e analisada através de todo o processo definido no decorrer deste trabalho.

Da comparação entre estas duas *frameworks*, pode concluir-se que, maioritariamente, a Flutter é a melhor solução para as funcionalidades selecionadas, e independentemente do sistema operativo alvo do smartphone (iOS ou Android OS). Contudo, a comparação entre as *frameworks* foi realizada de forma a abranger um maior leque de pontos de vista e requisitos. Assim, dependendo do objetivo de quem estiver a analisar estes resultados, poderá tirar mais facilmente as suas conclusões para um maior número de cenários.

Tendo em conta que o mercado de *smartphones* é dominado pelo sistema operativo Android OS e supondo que o objetivo da aplicação a ser desenvolvida é ter a melhor *performance* neste sistema operativo, não existem dúvidas que a Flutter é realmente uma boa alternativa para aplicações que reúnam um conjunto de funcionalidades semelhantes àquelas desenvolvidas e analisadas.

Ainda assim, se o objetivo for maioritariamente o sistema operativo da Apple, o iOS, já existem diversas funcionalidades em que a React Native se destaca, acabando por ser uma boa alternativa. Exemplos incluem o *launch* de aplicações, aplicações que envolvam armazenamento local e listas, captura de conteúdos de vídeo e ainda no consumo de conteúdos *streaming*. Do ponto de vista das métricas, a React Native também mostra ser uma boa alternativa para uma aplicação económica em termos de memória RAM, principalmente no caso do sistema operativo iOS.

Depois da análise de todos os valores, uma conclusão que se pode também derivar é a de que, se o objetivo for o desenvolvimento de uma aplicação sem grandes efeitos visuais para o sistema operativo iOS, a React Native é a melhor solução, principalmente se a aplicação envolver armazenamento local SQLite. Já no caso de ser pretendida uma aplicação que envolva melhores efeitos visuais, sendo este um dos pontos fortes da Flutter, tanto a nível de *performance* como a nível da facilidade de implementação, e uma aplicação que obtenha

bons consumos no geral, nos dois sistemas operativos dominantes no mercado, a Flutter apresenta-se como a *framework* mais adequada.

7.2. Trabalho futuro

As perspetivas futuras focam-se na melhoria do processo de testes, bem como na elaboração de uma solução o mais autónoma possível. Ambiciona-se o desenvolvimento de uma ferramenta, com base naquelas oficiais e utilizadas, que venha a ser referência no mercado de avaliação de aplicações.

Desta forma, é pretendido tornar a ferramenta de *scripting* criada numa aplicação mais amadurecida, disponível para qualquer computador, sem necessidade de criar todo um meio ambiente de testes complexo, e que neste caso foi necessário.

A aplicação pretendida terá vários níveis de utilização, destinando-se tanto a programadores que pretendam avaliar as aplicações que desenvolvem, como a outras entidades com conhecimentos menos aprofundados de programação. Será incorporado todo um processo automático de fusão de diversas funcionalidades já implementadas em múltiplas *frameworks*. Esta vertente da aplicação irá permitir que qualquer entidade, mesmo sem ter conhecimento algum relativamente às tecnologias a comparar e até mesmo sem nenhuns conceitos de programação, possa realizar uma comparação de forma muito mais facilitada.

Mesmo com inúmeras dificuldades já ultrapassadas no trabalho desenvolvido, neste trabalho futuro, acredito na existência de alguns pontos que também possam vir a ser exigentes.

Todavia, acredito vivamente que futuramente irá surgir uma ferramenta para preencher esta necessidade e gostaria de ser uma das pessoas a contribuir para tal feito.

(esta página foi deixada intencionalmente em branco)

Referências

[1] Statista Inc., “Mobile OS market share 2018”, 2019. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/ (acedido Set. 23, 2020).

[2] Drifty Co., “Cross-Platform Mobile App Development”, 2020. https://ionicframework.com/ (acedido Set. 23, 2020).

[3] Google Inc., “Flutter - Beautiful native apps in record time”, 2018. https://flutter.dev/ (acedido Set. 23, 2020).

[4] M. Gonsalves, “Evaluating the Mobile Development Frameworks Apache Cordova and Flutter and Their Impact on the Development Process and Application Characteristics”, 2018.

[5] A. Biørn-Hansen, T. M. Grønli, e G. Ghinea, “Animations in cross-platform mobile applications: An evaluation of tools, metrics and performance”, *Sensors (Switzerland)*, vol. 19, n. 9, pp. 2081-., Mai. 2019, doi: 10.3390/s19092081.

[6] J. Saarinen, “EVALUATING CROSS-PLATFORM MOBILE APP PERFORMANCE WITH VIDEO-BASED Faculty of Information Technology and Communication Sciences”, 2019.

[7] J. Jagiełło, “Performance Comparison Between React Native and Flutter”, 2019.

[8] E. Johansson e J. Söderberg, “Evaluating performance of a React Native feature set”, 2018.

[9] F. Asp Handledare, A. Palanisamy, O. Karlsson Examinator, e K. Sandahl, “A comparison of Ionic 2 versus React Native and Android in terms of performance, by comparing the performance of applications”, 2017.

[10] S. Huber e L. Demetz, “Performance analysis of mobile cross-platform development approaches based on typical UI interactions”, *ICSOFT 2019 - Proc. 14th Int. Conf. Softw. Technol.*, n. Icsoft 2019, pp. 40–48, 2019, doi: 10.5220/0007838000400048.

[11] F. U. R. Quazi e N. Sinha, “Android-Platform Based Determination of Fastest

Cross-Platform Framework", *Int. J. Comput. Sci. Mob. Comput.*, vol. 7, n. 9, pp. 1–12, 2018.

[12] N. Kadam e N. Singh, "Performance Breakdown of React Native Framework Compare To Native Applications", *J. Gujarat Res. Soc.*, vol. 21, n. 14, pp. 282–291, 2019.

[13] R. Eskola, "React Native Performance Evaluation", 2018.

[14] C. CONTASEL, R. RUGHINIS, D. TRANCA, e C. Dumitru, "Impact of Cross-Platform Development Frameworks on the Performance of Mobile Communications for Short Distances", *14th Int. Sci. Conf. eLearning Softw. Educ. Bucharest, April 19-20, 2018*, pp. 403–410, 2018.

[15] O. Dahl, "Exploring End User ' s Perception of Flutter Mobile Apps", 2019.

[16] M. Rodríguez-Sánchez Guerra, "Cross-platform development frameworks for the development of hybrid mobile applications: Implementations and comparative analysis", 2018.

[17] M. Scharfstein e Gaurf, "OTT Video-Oriented Mobile Applications Development Using Cross-Platform UI Frameworks", *J. Chem. Inf. Model.*, vol. 53, n. 9, pp. 1689–1699, 2013, doi: 10.1017/CBO9781107415324.004.

[18] L. Andersson, "Usability and User Experience in Mobile App Frameworks", 2018.

[19] G. Callebaut, M. Willocx, J. Vossaert, V. Naessens, e L. De Strycker, "A Study of Cordova and Its Data Storage Strategies".

[20] W. Wu, "React Native vs Flutter, cross-platform mobile application frameworks", *Metrop. Univ.*, March, 2018.

[21] A.-K. Evert, "Cross-Platform Smartphone Application Development with Kotlin Multiplatform", *Degree Proj. Comput. Sci. Eng.*, 2019.

[22] A. J. Alférez Zamora, "Estudio comparativo de frameworks multiplataforma para desarrollo de aplicaciones móviles", 2018.

[23] O. Gill, "Using React Native for Mobile Software Development", 2018.

[24] D. Avdic, “React Native vs Xamarin – Mobile for industry”.

[25] N. Lockwood, *iOS Core Animation: Advanced Techniques*. 2013.

[26] Apple Inc., “Profiling in Depth - WWDC 2015 - Videos - Apple Developer”. https://developer.apple.com/videos/play/wwdc2015/412/ (acedido Set. 23, 2020).

[27] W. Jobe, “Native Apps Vs. Mobile Web Apps”, *Int. J. Interact. Mob. Technol.*, vol. 7, n. 4, p. 27-., Out. 2013, doi: 10.3991/ijim.v7i4.3226.

[28] A. Osmani, “Getting Started with Progressive Web Apps | Google Developers”, 2015. https://developers.google.com/web/updates/2015/12/getting-started-pwa (acedido Set. 23, 2020).

[29] The Apache Software Foundation, “Apache Cordova”. https://cordova.apache.org/ (acedido Set. 23, 2020).

[30] Adobe Systems Inc., “PhoneGap”. https://phonegap.com/ (acedido Set. 23, 2020).

[31] Facebook Inc., “JavaScript Environment · React Native”. https://reactnative.dev/docs/javascript-environment.html (acedido Set. 24, 2020).

[32] Facebook Inc., “React Native · A framework for building native apps using React”. https://reactnative.dev/ (acedido Out. 03, 2020).

[33] C. Vue Native, “Vue Native”. https://vue-native.io/ (acedido Out. 03, 2020).

[34] GitHub Inc., “Releases · flutter/flutter”. https://github.com/flutter/flutter/releases (acedido Nov. 12, 2020).

[35] GitHub Inc., “Releases · ionic-team/ionic-framework”. https://github.com/ionic-team/ionic-framework/releases (acedido Nov. 12, 2020).

[36] GitHub Inc., “Releases · facebook/react-native”. https://github.com/facebook/react-native/releases (acedido Nov. 12, 2020).

[37] GitHub Inc., “Releases · GeekyAnts/vue-native-core”. https://github.com/GeekyAnts/vue-native-core/releases/ (acedido Nov. 12, 2020).

[38] GitHub Inc., “GitHub - facebook/react-native: A framework for building native apps with React.” https://github.com/facebook/react-native (acedido Set. 23, 2020).

[39] GitHub Inc., “GitHub - flutter/flutter: Flutter makes it easy and fast to build beautiful apps for mobile and beyond.” https://github.com/flutter/flutter (acedido Set. 23, 2020).

[40] GitHub Inc., “GitHub - ionic-team/ionic-framework: A powerful cross-platform UI toolkit for building native-quality iOS, Android, and Progressive Web Apps with HTML, CSS, and JavaScript.” https://github.com/ionic-team/ionic-framework (acedido Set. 23, 2020).

[41] “Flutter architectural overview - Flutter”. https://flutter.dev/docs/resources/architectural-overview (acedido Set. 23, 2020).

[42] GitHub Inc., “Releases · flutter/flutter · GitHub”. https://github.com/flutter/flutter/releases?after=v0.0.11 (acedido Set. 23, 2020).

[43] Google Inc., “Flutter - What technology is Flutter built with?”, 2018. https://flutter.dev/docs/resources/faq#what-technology-is-flutter-built-with (acedido Set. 23, 2020).

[44] Google Inc., “Flutter - How does Flutter run my code on Android?” https://flutter.dev/docs/resources/faq#run-android (acedido Set. 23, 2020).

[45] Apple Inc., “Instruments Help”. https://help.apple.com/instruments/mac/10.0/ (acedido Out. 01, 2020).

[46] Apple Inc., “Xcode - Apple Developer”. https://developer.apple.com/xcode/ (acedido Out. 01, 2020).

[47] Google Inc., “Android Studio and SDK tools | Android Developers”. https://developer.android.com/studio (acedido Out. 01, 2020).

[48] Google Inc., “Android Device Monitor | Desenvolvedores Android | Android Developers”. https://developer.android.com/studio/profile/monitor (acedido Set. 24, 2020).

[49] M. Willocx, J. Vossaert, e V. Naessens, “Comparing performance parameters of mobile app development strategies”, *Proc. - Int. Conf. Mob. Softw. Eng. Syst. MOBILESoft 2016*, pp. 38–47, 2016, doi: 10.1145/2897073.2897092.

[50] Google Inc., “Overview of system tracing | Android Developers”. https://developer.android.com/topic/performance/tracing (acedido Out. 01, 2020).

[51] Google Inc., “Test UI performance | Android Developers”. https://developer.android.com/training/testing/performance (acedido Out. 01, 2020).

[52] Google Inc., “Dumpsys meminfo | Android Developers”. https://developer.android.com/studio/command-line/dumpsys#meminfo (acedido Out. 01, 2020).

[53] Google Inc., “Logcat command-line tool | Android Developers”. https://developer.android.com/studio/command-line/logcat (acedido Out. 01, 2020).

[54] Google Inc., “Capture and read bug reports | Android Developers”. https://developer.android.com/studio/debug/bug-report (acedido Out. 01, 2020).

[55] GeeksforGeeks Inc., “Linux - Top”. https://www.geeksforgeeks.org/top-command-in-linux-with-examples/ (acedido Set. 24, 2020).

[56] Google Inc., “Android Debug Bridge (adb) | Android Developers”, *Developer.Android.Com*, 2018. https://developer.android.com/studio/command-line/adb?hl#shellcommands (acedido Set. 24, 2020).

[57] A. Venkatraman, “A successful finale to the decade: mobile highlights of 2019”, *App Annie Blog*, 2019. https://www.appannie.com/en/insights/market-data/a-successful-finale-to-the-decade-mobile-highlights-of-2019/ (acedido Set. 24, 2020).

[58] Advanced Development Methods Inc., “Home | Scrum.org”. https://www.scrum.org/ (acedido Out. 01, 2020).

[59] SCRUM ALLIANCE Inc., “Scrum Alliance Certification | Transform your workplace”. https://www.scrumalliance.org/ (acedido Out. 01, 2020).

[60] Kanbanize Inc., “Kanban Explained in 10 Minutes | Kanbanize”, 2018. https://kanbanize.com/kanban-resources/getting-started/what-is-kanban (acedido Set. 24, 2020).

[61] D. J. Anderson, *Kanban: successful evolutionary change for your technology business. Blue Hole Press*. 2010.

[62] P. Saed e Y. Yahya, “Loading time effects: A case study of Malaysian Examination Syndicate web portal”, *Proc. 2011 Int. Conf. Electr. Eng. Informatics, ICEEI 2011*, n. July, pp. 3–7, 2011, doi: 10.1109/ICEEI.2011.6021664.

[63] SQLite Consortium, “SQLite Home Page”. https://www.sqlite.org/index.html (acedido Out. 01, 2020).

[64] S. Kemp, “Digital 2020: Global Digital Overview — DataReportal – Global Digital Insights”, *Datareportal.com*. p. 1, 2020.

[65] Google Inc., “Flutter’s build modes - Flutter”. https://flutter.dev/docs/testing/build-modes (acedido Mai. 21, 2020).

[66] F. OpenJS, “Node.js”. https://nodejs.org/en/ (acedido Out. 04, 2020).

[67] Apple Inc., “Instruments Help”. https://developer.apple.com/library/archive/documentation/AnalysisTools/Conceptual/instruments_help-collection/Chapter/Chapter.html (acedido Set. 24, 2020).

[68] Google Inc., “Choreographer.FrameCallback | Android Developers”. https://developer.android.com/reference/android/view/Choreographer.FrameCallback (acedido Out. 17, 2020).

Apêndices

Apêndice A

Lista das tecnologias utilizadas para a criação das aplicações nos diversos estudos analisados.

(Esta página foi deixada em branco propositadamente)

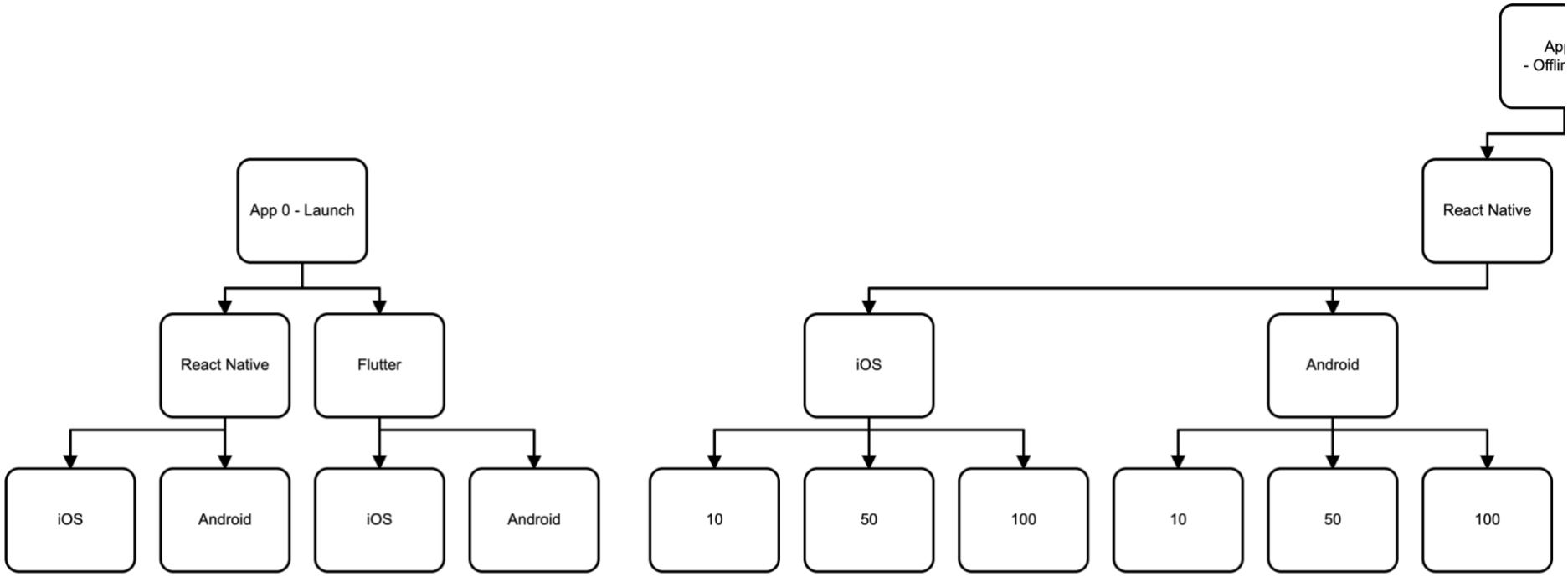
Title
Animations In Cross-Platform Mobile Applications: An Evaluation Of Tools, Metrics And Performance
Performance Comparison Between React Native And Flutter
Evaluating Performance Of A React Native Feature Set
A Comparison Of Ionic 2 Versus React Native And Android In Terms Of Performance, By Comparing The Performance Of Applications
Performance Analysis Of Mobile Cross-Platform Development Approaches Based On Typical Ui Interactions
Android-Platform Based Determination Of Fastest Cross-Platform Framework
Performance Breakdown Of React Native Framework Compare To Native Applications
React Native Performance Evaluation
Impact Of Cross-Platform Development Frameworks On The Performance Of Mobile Communications For Short Distances
Evaluating The Mobile Development Frameworks Apache Cordova And Flutter And Their Impact On The Development Process And Application Characteristics
Evaluating Cross-Platform Mobile App Performance With Video-Based Measurements
Exploring End User'S Perception Of Flutter Mobile Apps
Cross-Platform Development Frameworks For The Development Of Hybrid Mobile Applications: Implementations And Comparative Analysis
Ott Video-Oriented Mobile Applications Development Using Cross-Platform Ui Frameworks
Usability And User Experiencein Mobile App Frameworks: Subjective, But Not Objective, Differencesbetween A Hybrid And A Native Mobile Application
A Study Of Cordova And Its Data Storage Strategies
React Native Vs Flutter, Cross-Platform Mobile Application Frameworks
Cross-Platform Smartphone Application Development With Kotlin Multiplatform: Possible Impacts On Development Productivity, Application Size And Startup Time
Estudio Comparativo De Frameworks Multiplataforma Para Desarrollo De Aplicaciones Móviles
Using React Native For Mobile Software Development
React Native Vs Xamarin – Mobile For Industry

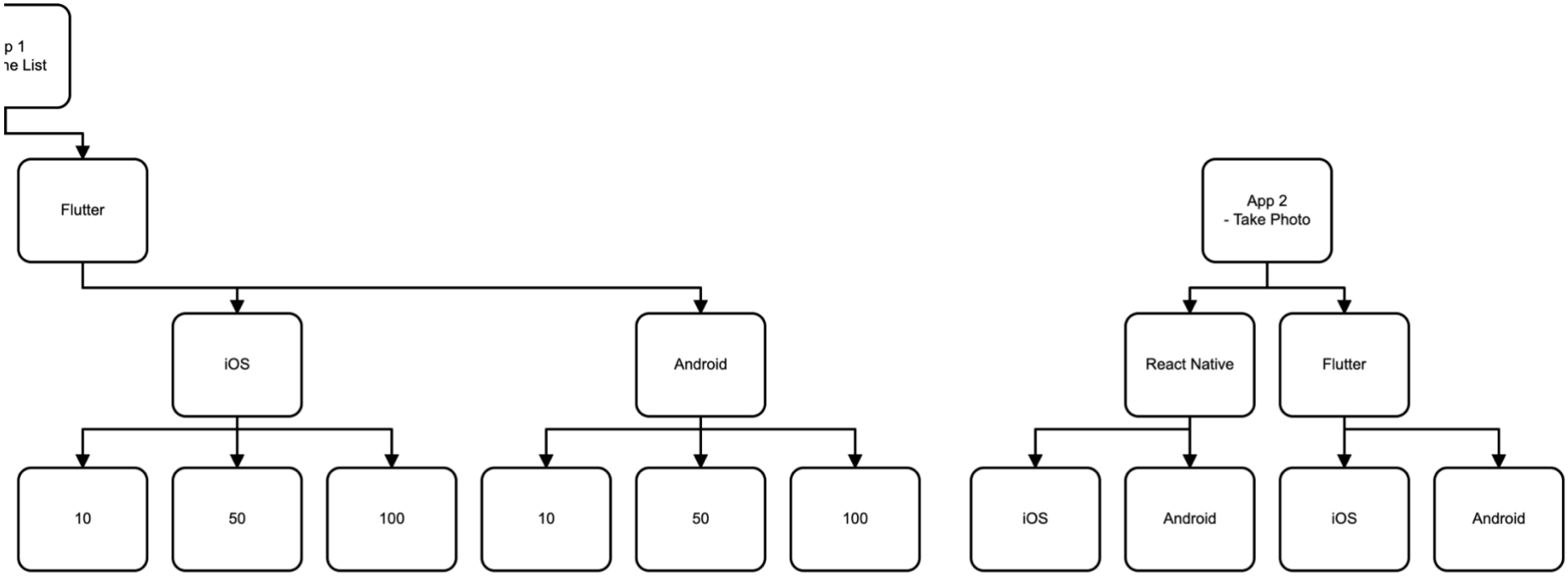
Frameworks										
Native Android	Native iOS	React Native	Ionic	Xamarin	Weex	Titanium	Cordova	PhoneGap	Flutter	Kotlin Multiplatform
X	X	X	X	X	-	-	-	-	-	-
-	-	X	-	-	-	-	-	-	X	-
-	-	X	-	-	-	-	-	-	-	-
X	-	X	X	-	-	-	-	-	-	-
X	-	X	X	-	-	-	-	-	-	-
-	-	X	X	-	-	-	-	X	-	-
X	-	X	-	-	-	-	-	-	-	-
X	-	X	-	-	-	-	-	-	-	-
X	-	-	-	X	-	-	-	X	-	-
X	X	-	X	-	-	-	-	-	X	-
X	X	X	-	-	-	X	X	-	X	-
X	-	-	-	-	-	-	-	-	X	-
-	-	X	X	-	X	-	-	-	X	-
-	-	X	-	X	-	-	-	-	X	-
-	X	X	-	X	-	-	X	-	-	-
X	X	-	-	-	-	-		-	-	-
-	-	X	-	-	-	-	-	-	X	-
X	X	-	-	-	-	-	-	-	-	X
-	-	X	-	X	-	-	-	-	X	-
-	X	X	-	-	-	-	-	-	-	-
-	-	X	-	X	-	-	-	-	-	-

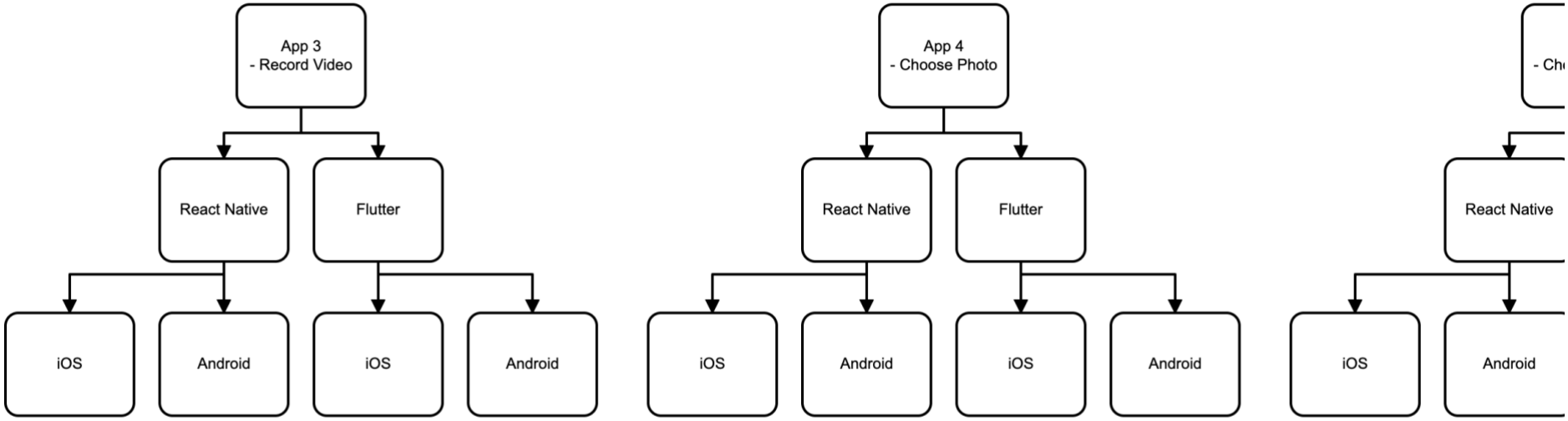
Apêndice B

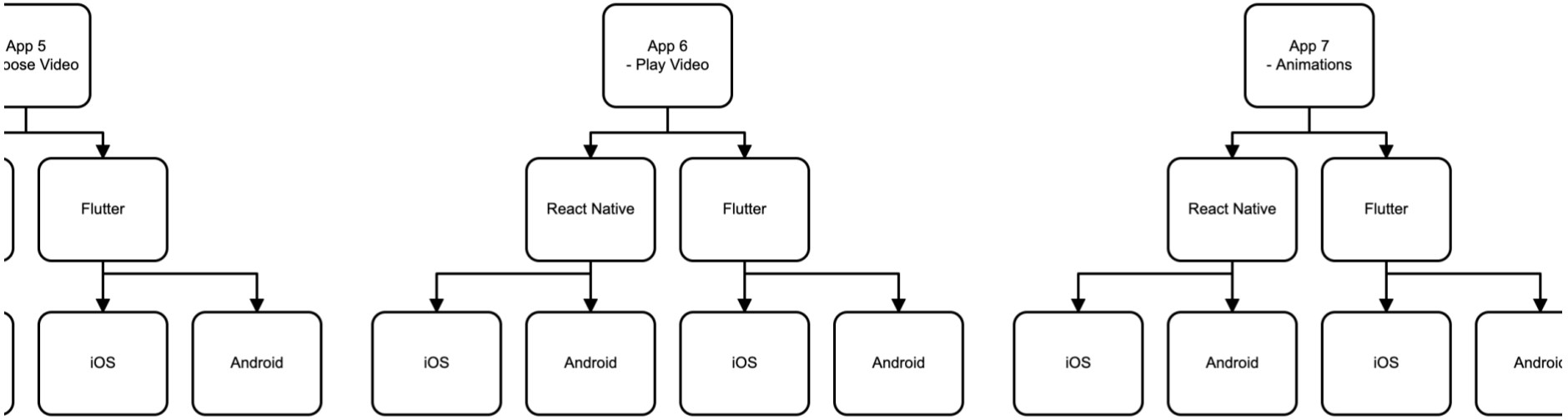
Diagrama da origem das 52 aplicações resultantes deste trabalho.

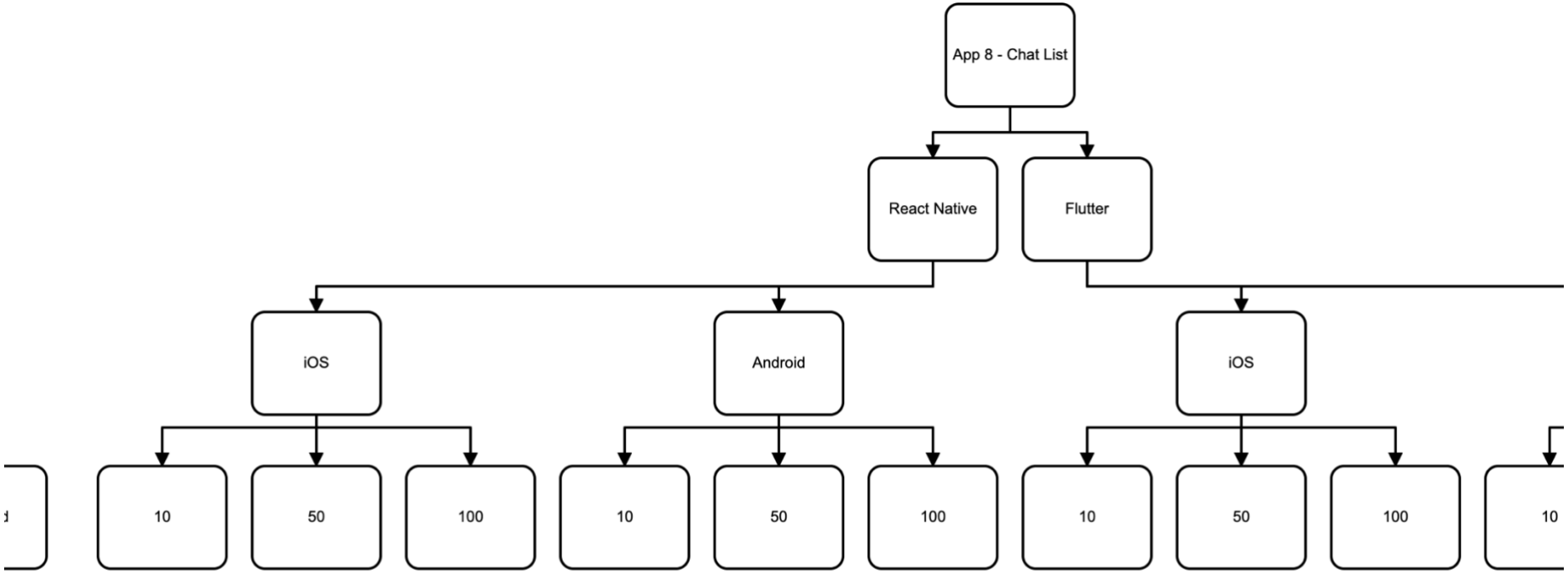
(Esta página foi deixada em branco propositadamente)

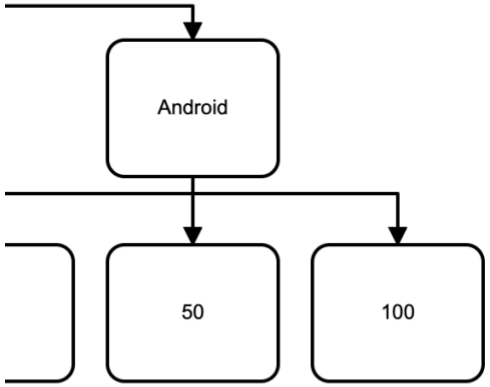

App 0 - Launch
React Native
Flutter
iOS
Android
iOS
Android
Ap
- Offlir
React Native
iOS
Android
10
50
100
10
50
100


p 1
ne List
Flutter
iOS
Android
10
50
100
10
50
100
App 2
- Take Photo
React Native
Flutter
iOS
Android
iOS
Android


App 3
- Record Video
React Native
Flutter
iOS
Android
iOS
Android
App 4
- Choose Photo
React Native
Flutter
iOS
Android
iOS
Android
React Native
iOS
Android


App 5
oose Video
Flutter
iOS
Android
App 6
- Play Video
React Native
Flutter
iOS
Android
iOS
Android
App 7
- Animations
React Native
Flutter
iOS
Android
iOS
Androi

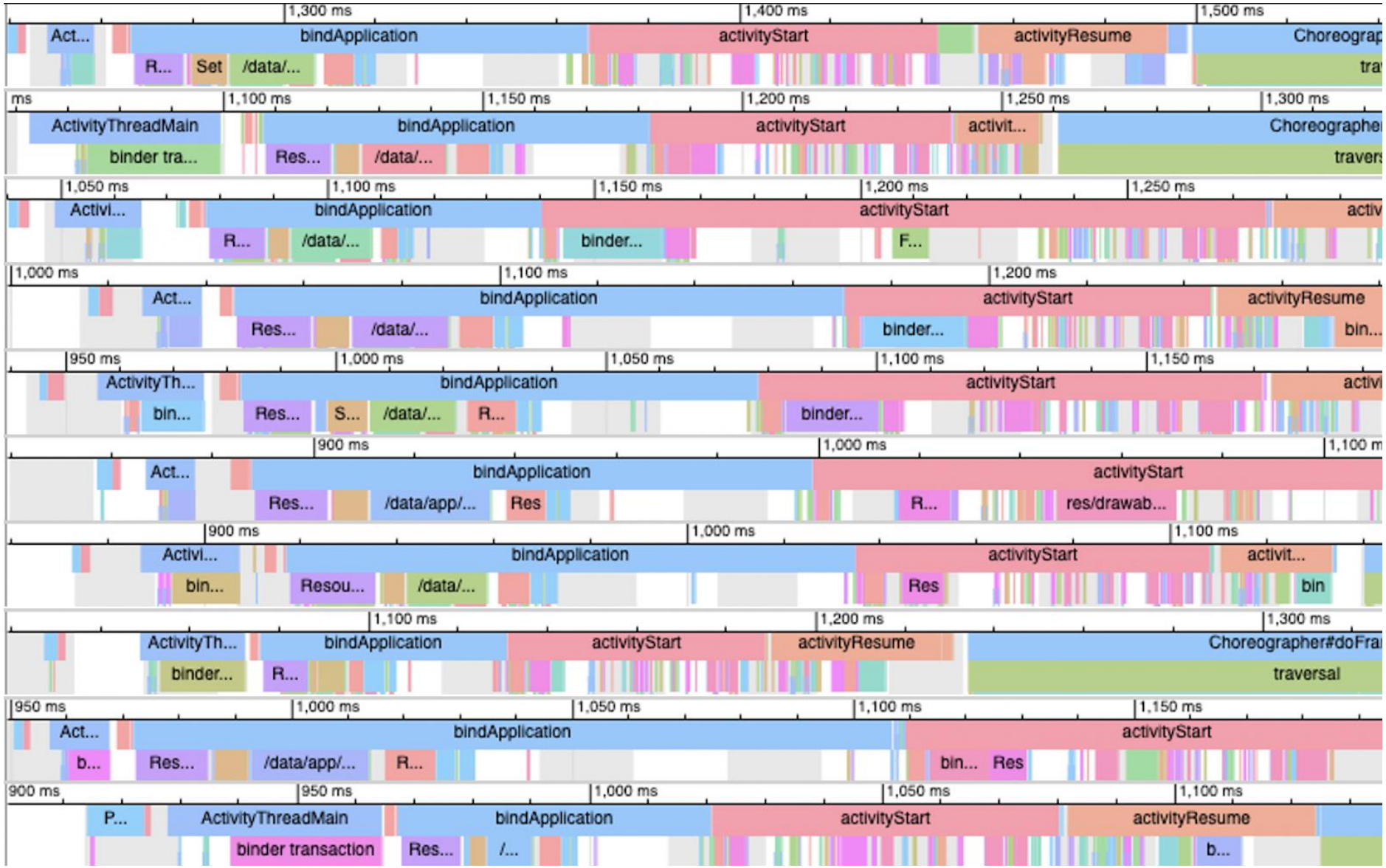

App 8 - Chat List
React Native
Flutter
iOS
Android
iOS
10
50
100
10
50
100
10
50
100
10

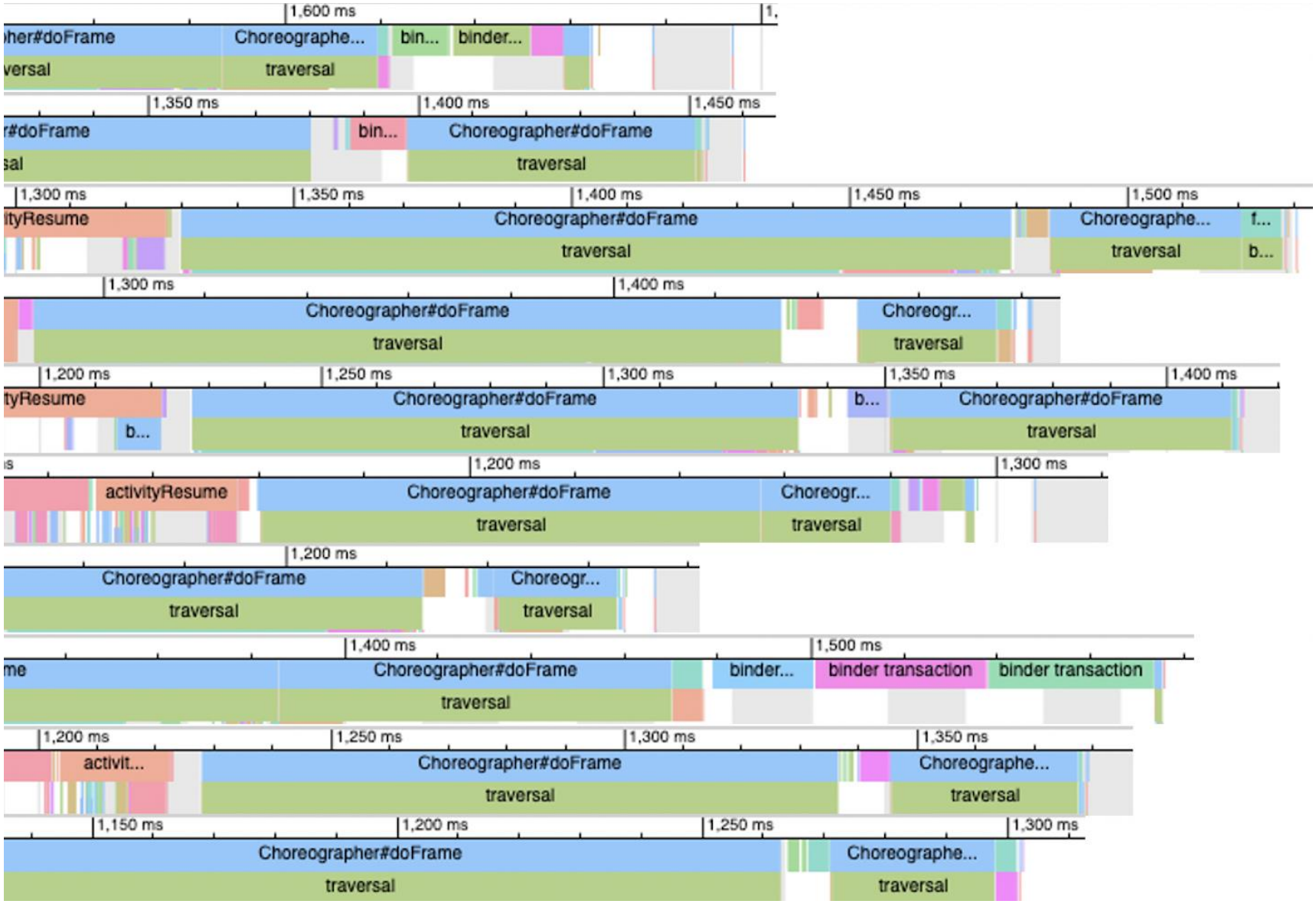

Android
50
100

Apêndice C

As 10 iterações do teste de consumos gráficos na aplicação Launch em Android pelo React Native.

(Esta página foi deixada em branco propositadamente)

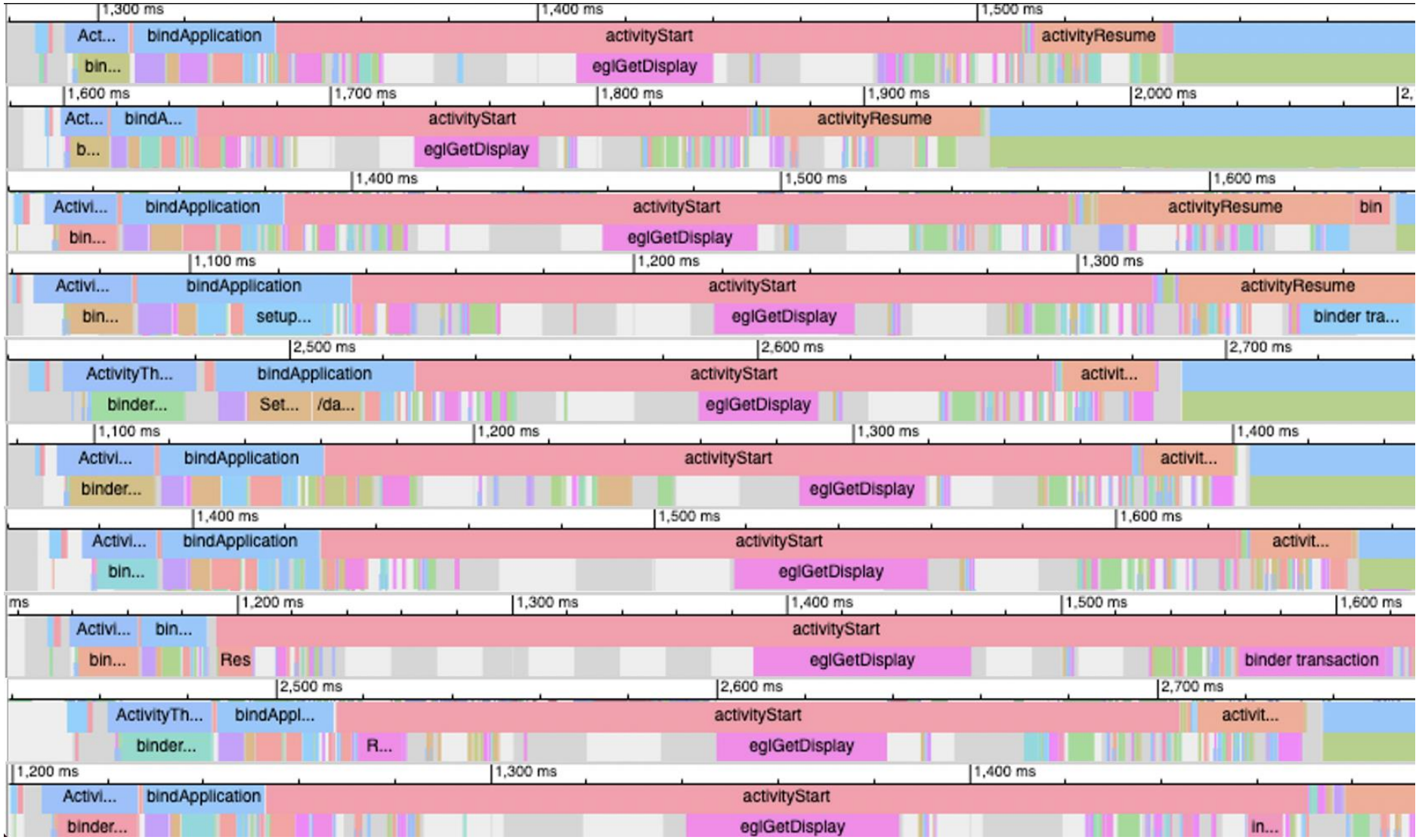

1,300 ms
1,400 ms
1,500 ms
Act...
bindApplication
activityStart
activityResume
Choreograp
R...
Set
/data/...
tra
ms
1,100 ms
1,150 ms
1,200 ms
1,250 ms
1,300 ms
ActivityThreadMain
bindApplication
activityStart
activit...
Choreographe
binder tra...
Res...
/data/...
travers
1,050 ms
1,100 ms
1,150 ms
1,200 ms
1,250 ms
Activi...
bindApplication
activityStart
activ
R...
/data/...
binder...
F...
1,000 ms
1,100 ms
1,200 ms
Act...
bindApplication
activityStart
activityResume
Res...
/data/...
binder...
bin...
950 ms
1,000 ms
1,050 ms
1,100 ms
1,150 ms
ActivityTh...
bindApplication
activityStart
activi
bin...
Res...
S...
/data/...
R...
binder...
900 ms
1,000 ms
1,100 m
Act...
bindApplication
activityStart
Res...
/data/app/...
Res
R...
res/drawab...
900 ms
1,000 ms
1,100 ms
Activi...
bindApplication
activityStart
activit...
bin...
Resou...
/data/...
Res
bin
1,100 ms
1,200 ms
1,300 ms
ActivityTh...
bindApplication
activityStart
activityResume
Choreographer#doFra
binder...
R...
traversal
950 ms
1,000 ms
1,050 ms
1,100 ms
1,150 ms
Act...
bindApplication
activityStart
b...
Res...
/data/app/...
R...
bin...
Res
900 ms
950 ms
1,000 ms
1,050 ms
1,100 ms
P...
ActivityThreadMain
bindApplication
activityStart
activityResume
binder transaction
Res...
/...
b...

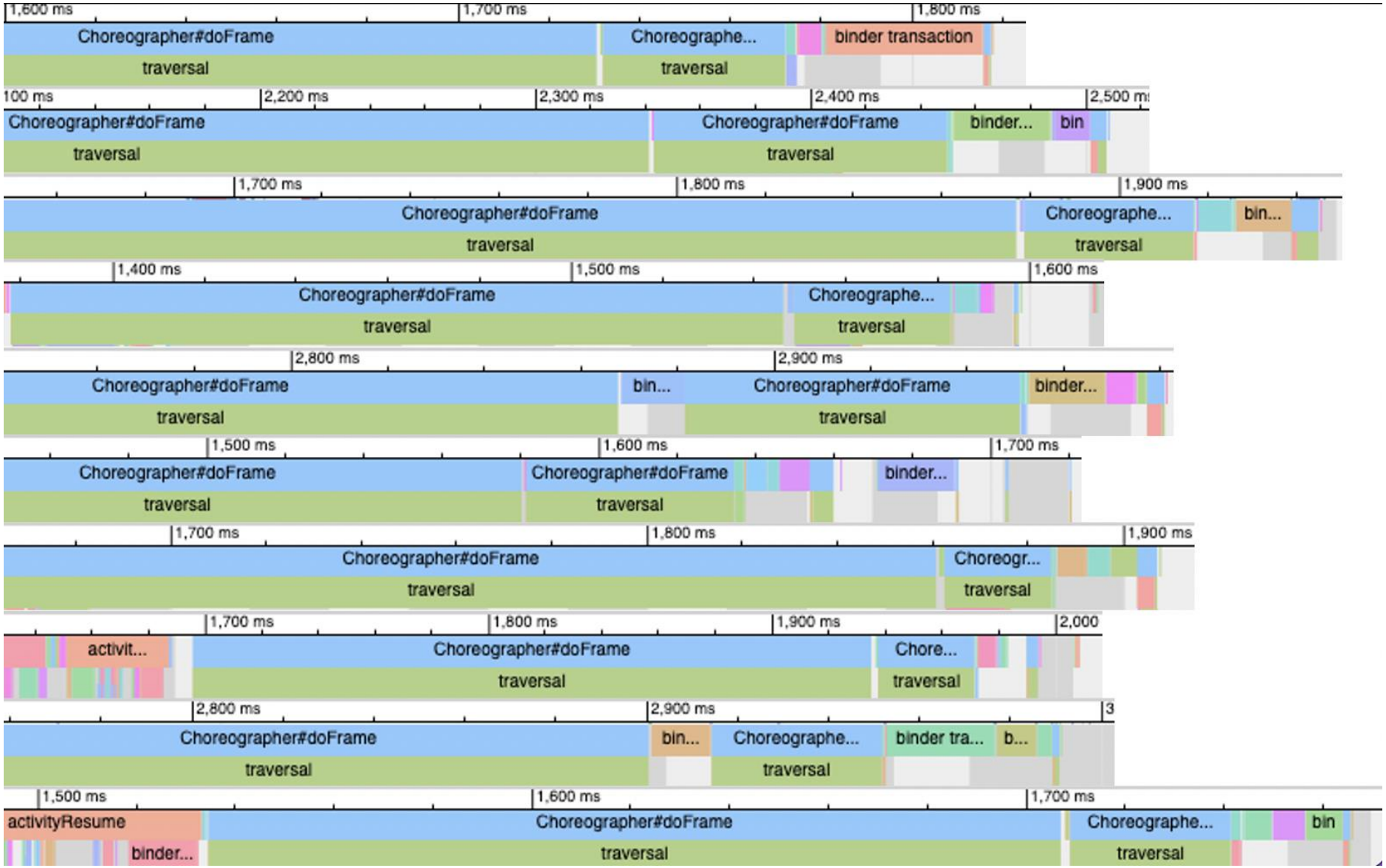

1,600 ms
her#doFrame
Choreographe...
bin...
binder...
versal
traversal
1,350 ms
1,400 ms
1,450 ms
r#doFrame
bin...
Choreographer#doFrame
sal
traversal
1,300 ms
1,350 ms
1,400 ms
1,450 ms
1,500 ms
ityResume
Choreographer#doFrame
Choreographe...
f...
traversal
traversal
b...
1,300 ms
1,400 ms
Choreographer#doFrame
Choreogr...
traversal
traversal
1,200 ms
1,250 ms
1,300 ms
1,350 ms
1,400 ms
tyResume
Choreographer#doFrame
b...
Choreographer#doFrame
b...
traversal
traversal
1,200 ms
1,300 ms
activityResume
Choreographer#doFrame
Choreogr...
traversal
traversal
1,200 ms
Choreographer#doFrame
Choreogr...
traversal
traversal
1,400 ms
1,500 ms
me
Choreographer#doFrame
binder...
binder transaction
binder transaction
traversal
1,200 ms
1,250 ms
1,300 ms
1,350 ms
activit...
Choreographer#doFrame
Choreographe...
traversal
traversal
1,150 ms
1,200 ms
1,250 ms
1,300 ms
Choreographer#doFrame
Choreographe...
traversal
traversal

Apêndice D

As 10 iterações do teste de consumos gráficos na aplicação Launch em Android pelo Flutter.

(Esta página foi deixada em branco propositadamente)


1,300 ms
1,400 ms
1,500 ms
Act...
bindApplication
activityStart
activityResume
bin...
eglGetDisplay
1,600 ms
1,700 ms
1,800 ms
1,900 ms
2,000 ms
Act...
bindA...
activityStart
activityResume
b...
eglGetDisplay
1,400 ms
1,500 ms
1,600 ms
Activi...
bindApplication
activityStart
activityResume
bin
bin...
eglGetDisplay
1,100 ms
1,200 ms
1,300 ms
Activi...
bindApplication
activityStart
activityResume
bin...
setup...
eglGetDisplay
binder tra...
2,500 ms
2,600 ms
2,700 ms
ActivityTh...
bindApplication
activityStart
activit...
binder...
Set...
/da...
eglGetDisplay
1,100 ms
1,200 ms
1,300 ms
1,400 ms
Activi...
bindApplication
activityStart
activit...
binder...
eglGetDisplay
1,400 ms
1,500 ms
1,600 ms
Activi...
bindApplication
activityStart
activit...
bin...
eglGetDisplay
ms
1,200 ms
1,300 ms
1,400 ms
1,500 ms
1,600 ms
Activi...
bin...
activityStart
bin...
Res
eglGetDisplay
binder transaction
2,500 ms
2,600 ms
2,700 ms
ActivityTh...
bindAppl...
activityStart
activit...
binder...
R...
eglGetDisplay
1,200 ms
1,300 ms
1,400 ms
Activi...
bindApplication
activityStart
binder...
eglGetDisplay
in...


1,600 ms
1,700 ms
1,800 ms
Choreographer#doFrame
Choreographe...
binder transaction
traversal
traversal
100 ms
2,200 ms
2,300 ms
2,400 ms
2,500 ms
Choreographer#doFrame
Choreographer#doFrame
binder...
bin
traversal
traversal
1,700 ms
1,800 ms
1,900 ms
Choreographer#doFrame
Choreographe...
bin...
traversal
traversal
1,400 ms
1,500 ms
1,600 ms
Choreographer#doFrame
Choreographe...
traversal
traversal
2,800 ms
2,900 ms
Choreographer#doFrame
bin...
Choreographer#doFrame
binder...
traversal
traversal
1,500 ms
1,600 ms
1,700 ms
Choreographer#doFrame
Choreographer#doFrame
binder...
traversal
traversal
1,700 ms
1,800 ms
1,900 ms
Choreographer#doFrame
Choreogr...
traversal
traversal
1,700 ms
1,800 ms
1,900 ms
2,000
activit...
Choreographer#doFrame
Chore...
traversal
traversal
2,800 ms
2,900 ms
3
Choreographer#doFrame
bin...
Choreographe...
binder tra...
b...
traversal
traversal
1,500 ms
1,600 ms
1,700 ms
activityResume
Choreographer#doFrame
Choreographe...
bin
binder...
traversal
traversal