修士論文

大規模シミュレーションのためのスマートデバイスアプリケーションの構成法

中川 颯馬

主指導教員 篠田 陽一

北陸先端科学技術大学院大学
先端科学技術専攻
（情報科学）

令和７年３月

Abstract

In times of disasters or emergencies, the number of people using smart device applications (App) can increase dramatically. For example, in the case of massive Nankai Trough earthquake, as many as 8.8 million people are expected to be affected. If a large number of these people simultaneously use a disaster-response App, it is essential for App to continue functioning as usual. To ensure this, performance testing and large-scale deployment simulations must be conducted in advance. This need is especially critical for mission-critical App designed for use during disasters. Although existing load testing tools can be used for large-scale simulations, the loads generated by these tools are typically based on predefined and static scenarios. This makes it difficult to replicate the complex conditions of actual usage. By using App themselves to generate loads, it becomes possible to create more complex scenarios that better reflect real-life situations. This approach enables a more realistic evaluation of App performance under diverse conditions.

Existing methods for simulating large-scale App deployment include approaches that use physical devices and approaches that use emulators. When relying on physical devices, either on-premise devices or cloud-based services can be used. However, this tends to be impractical from a cost or resource perspective. Although emulators can accommodate larger scale than physical devices, they require significant overhead to emulate hardware features such as displays, input devices, and sensors. As a result, simulations on the order of millions of instances become difficult.

To address these challenges, this research proposes converting an App into a headless App (vApp). This allows the App to operate without depending on physical devices or emulators, thereby simplifying large-scale simulation. One straightforward way to remove the frontend entirely would leave only the core logic. However, this approach makes it impossible to test user input logic. Therefore, this study proposes rebuilding the frontend using a design pattern commonly employed in App with GUI. This allows for generalization of the elements that need to be rebuilt. MVC (Model-View-Controller) model is adopted here as an example for frontend rebuilding. During the process of making App headless, three major points were considered.

The first point involves an automated control mechanism for vApp. Since vApp has no GUI, existing methods such as touch operations or screen transitions are unusable. Moreover, manual operation is impractical for large-scale simulations. Consequently, an agent-oriented simulator (Agent) was introduced to simulate user operations and behavior scenarios. This enables vApp to be controlled automatically.

The second point concerns the design of I/O model for vApp. In a large-scale

simulation, each Agent is paired with one vApp. However, since vApp has no GUI, conventional input and output mechanisms, such as touch operations or screen transitions, cannot be used. To address this limitation, operations and information must be exchanged through message communication between Agent and vApp. This communication mechanism is defined here as “I/O model.” Two models are proposed to facilitate this interaction. The first is vScreen Model, which focuses on screen structure and transitions. The second is vOps Model, which places more emphasis on the services provided by App.

The third point addresses a sensor information exchange model. When App runs on a physical device or emulator, it can directly obtain information such as GPS data or battery status. However, because a vApp does not run on an actual device or emulator, these types of sensor information do not physically exist. To overcome this limitation, this study leverages the fact that Agent simulates user behavior. Agent pseudo-generates various sensor data and sends it to the vApp. This mechanism allows vApp to behave as if it were operating on a physical device. Two models are proposed for exchanging sensor information between the vApp and Agent. One model handles data asynchronously (Asynchronous Model), while the other exchanges data synchronously (Synchronous Model).

To verify the effectiveness of the proposed approach, an example App was implemented as vApp using MVC pattern. Partial rebuilding of GUI in that vApp resulted in App variant. This made it possible to clarify the implementation differences and confirm the reproducibility of the method.

A comparative evaluation was then conducted using five different methods that combine various execution environments, forms of App, and operation methods. These methods were evaluated based on computing resources, required time, execution costs, ease of automation, ease of debugging, scalability, fidelity, and implementation overhead. The results show that the proposed method demonstrates clear advantages in many of these areas. Regarding computing resources and scalability, executing lightweight vApp on general-purpose servers enables simulations on the order of one million instances at reasonable cost and within reasonable time. In terms of execution costs and time, substantial savings were observed compared to other methods. Notably, the proposed method achieved approximately twenty-four times better cost efficiency than emulator-based approaches. Removing GUI also simplifies automation. Since there is no need to track screen transitions or user input, the control program becomes less complex than existing automated tools.

On the other hand, there are indications that the method may be inferior in terms of fidelity, as it is difficult to reproduce all user interaction patterns or device-specific behaviors. Additionally, implementation overhead remains an is-

sue, particularly due to the need to restore GUI if vApp is converted back into a fully functional App. Future work includes developing an automatic mechanism for generating vApp from existing App and refining Agent to simulate more detailed user behaviors. Further research will also involve conducting large-scale experiments with millions or tens of millions of simulations and clarifying which types of App and requirements are best suited for this method.

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

災害や非常事態時には、スマートデバイス向けアプリケーション（以下、App）の利用者が急激に増加する可能性がある。例えば、南海トラフ地震では、最大で約880万人の被災者が出ることが予想されている [1]。多くの被災者が同時に災害対応用Appを使用する場合、Appは通常通り動作することが欠かせない。したがって、事前に動作検証や大規模展開時のシミュレーションが必要である。特に、大規模災害時に使用される災害対応Appのようなミッションクリティカルな Appは、極端な負荷状況や予期せぬ障害に対しても検証する必要がある。

大規模展開時のシミュレーションは、従来の負荷テストツールを用いることで実施可能であるが、これらが生成する負荷は、あらかじめ設定されたシナリオに基づく機械的かつ静的な負荷に留まり、現実の使用状況を十分に再現することが難しい。しかし、App自体を用いることで、現実により近い複雑な負荷を生成することが可能となる。この方法により、実際の運用状況を反映した多様な負荷条件下でのシミュレーションが実現し、Appの動作をより現実的に評価することが可能となる。

Appの検証環境には、主に二つの既存手法がある。一つ目は、物理デバイスを用いて実施する方法である。この方法では、実際のデバイス上でAppを動作させるため、精度の高い動作検証が可能である。しかし、この方法ではシミュレーションを実施する際の規模の確保が困難となる。主な理由は、物理デバイスの数に限界があることである。大量のデバイスを用意するには、デバイスの調達コスト、設置スペース、電力供給、ネットワーク接続のインフラ整備など、多くのリソースが必要となる。これらのリソースは物理的な制約により無限に拡張できず、大規模なテスト環境を維持するには膨大なコストと管理労力が発生する。

二つ目は、Cuttlefish[2] などのスマートデバイスのエミュレータを用いる方法である。エミュレータは、GUIの再現にディスプレイや仮想ディスプレイを必要とし、その結果リソース消費が増大するため、数百万台規模のシミュレーションを実施する際にリソースの制約や管理の複雑さが生じ、実行が容易ではない。したがって、よりスケーラブルな手法が求められる。

1.2 目的

災害や非常事態時に、App の利用者が急激に増加する状況が想定され、ミッションクリティカルな App の場合は事前にこのような状況をシミュレーションする必要がある。しかし、既存手法では App の数百万台規模でのシミュレーションは、スケーラビリティの面で困難である。本研究ではこの課題を解決するため、App の大規模自動実行に適したプログラムの構造を提案し、エージェントシミュレータ指向（以下、Agent）との連携により、大規模シミュレーション実行を可能にする。Agent は、App に対するユーザーの操作をシミュレートしたものであり、事前に設定されたシナリオやアルゴリズムに基づいて特定のタスクを自律的に実行するソフトウェアプログラムを指す。Agent はユーザーの介入なしに操作を遂行し、状況に応じて適切な行動をとることが可能である。

1.3 本論文の構成

本論文は、本章を含めて 6 章で構成される。第 2 章では App の検証環境の既存手法である、物理デバイス及びエミュレータを用いる手法の課題を整理し、App の自動操作の既存手法について述べる。第 3 章では本研究の提案手法である App のデザインパターンを用いた、App のヘッドレス化の方法について詳細に説明する。また、ヘッドレス化に伴い検討の必要が出た、vApp に操作を実行する機構、vApp に対する I/O モデル、デバイス特有のセンサ情報の取り扱いについて述べる。第 4 章では、提案手法の例題アプリケーションへの適用方法を示すことで、具体的な提案手法の適用方法について述べる。第 5 章では、App を大規模展開した際のシミュレーション手法を複数提示し、各手法の比較評価の結果と考察について述べる。評価項目として、計算リソース、所要時間、実行費用、忠実性などを用いて提案手法の有効性を検証した。最後に、第 6 章で本研究の総括を行い、今後の課題や展望について述べる。

第2章　既存手法

本章では、App を大規模に展開してシミュレーションを行う際に用いられる実行環境の既存手法を整理する。具体的には、物理デバイスを用いる方法、およびエミュレータを用いる方法の二つを取り上げる。さらに、App を自動操作する既存手法についても概観し、その課題を明らかにする。

2.1　物理デバイスを用いる手法

物理デバイスを用いる手法として、手元に存在する物理デバイスを用いる手法とクラウドベースのサービス [3, 4, 5, 6] を利用する手法が存在する。手元に存在する物理デバイスを用いる手法は、App の大規模展開時のシミュレーションをより忠実に再現することが可能であるが、調達コストなどの面から現実的ではない。

クラウドベースのサービスは、クラウド上に物理的なスマートデバイスが存在し、その上で App の実行をするというものである。このサービスの特徴として、複数の種類のスマートデバイスや OS のバージョンを利用可能であること、並行して複数デバイスでの自動テストが可能であること、リアルタイムでのフィードバックや詳細なログの取得が可能であることが挙げられる。また、物理デバイスを用意する手間がなくなるため、App の検証環境を迅速に構築することが可能である。しかし、クラウド上の物理デバイスに依存するため、利用コストや同時に実行可能な App の数が問題となる。

AWS Device Farm[3] を例に、クラウドベースの物理デバイス提供サービスが抱える制約を示す。まず課金体系として、定額制（月額 150 ドル/台でテスト無制限）と、従量課金制（1 分あたり 0.17 ドル）の二種類が用意されている。さらに、以下のような制約がある。

- **テストアプリのサイズ上限：**4GB
- **同時実行デバイス数の上限：**5 台
- **テスト時間の上限：**2.5 時間

例えば、AWS Device Farm では同時実行デバイス数が 5 台、Firebase Test Lab では有料枠拡張しても数十台程度など、単一アカウントあたりの並列実行に上限がある。数万〜数百万クラスのシミュレーションを要する場合は、テスト待ち行

列が生じて実時間が膨大となり、料金も爆発的に増加する。そのため、クラウドの物理デバイスサービスを利用しても大規模テストには依然限界がある。

2.2 エミュレータを用いる手法

本節では、スマートデバイスのエミュレータの概要及び、エミュレータを用いた App の大規模展開時のシミュレーションの制限について議論する。

2.2.1 エミュレータの概要

スマートデバイスのエミュレータには、Cuttlefish[2] のようなスタンドアロンのエミュレータ [2, 7, 8, 9, 10, 11] や、Xcode Simulator[12]、Android Emulator[13] など統合開発環境に内包されているものがある。

たとえば、Cuttlefish は、KVM（Kernel-based Virtual Machine）[14] と QEMU[15] を組み合わせることで、Android 向けエミュレータを効率的に実行している。具体的には、KVM が CPU 命令セットやメモリのハイパーバイザ機能を提供し、QEMU はディスプレイやセンサ、ネットワークなどの仮想ハードウェアをエミュレートする。このように、Cuttlefish 上で起動される Android OS は、あたかも実機同様に各種ハードウェアを操作できるような仕組みになっている。エミュレータの内部の仮想化技術やモジュールには、以下に挙げるものがある。

- **OS イメージの仮想化** Android や iOS などの OS イメージをホスト OS 上で動作させるためのハイパーバイザ／コンテナ技術を使用。
- **ハードウェア抽象化**: スマートフォン固有の CPU 命令セット、GPU、メモリ構成、ディスプレイ解像度などをエミュレートし、App からは実機同等に見える環境を提供い
- **センサや周辺機器のシミュレーション**: GPS 位置情報や加速度・ジャイロなどのセンサ情報を仮想的に生成し、App に通知する。カメラやマイクなどを仮想的に接続できる場合もある。
- **GUI 描画**: ホスト OS のウィンドウ上に仮想デバイスの画面を表示する機構。App の UI はあたかも実機の画面に表示されるようにレンダリングされる。

これらの機能により、開発者は実機を多数用意せずに多様な OS や端末構成での動作検証を実施できるため、一般的な単体テストやデバッグ時には非常に便利である。

2.2.2 大規模実行における制限

スマートデバイスのエミュレータを用いて、App の大規模展開時のシミュレーションを実施する方法が考えられる。しかし、エミュレータはスマートデバイスが持つディスプレイ、入力デバイス、センサといったハードウェア機能をエミュレートしているため、実行する際のコンピュータリソースのオーバーヘッドが大きい。そのため、実行可能な App の数に制限が生じる。例えば、以下の性能のサーバー 395 台を用いて Cuttlefish 上で同時に実行可能な App は 5,918 個に限られている [16]。

- **CPU**：64 コア/ARM v8.2/2.6 GHz
- **DRAM**：128GB
- **SSD**：1TB(NVMe)
- **NIC**：1Gbps

以上のことから、高性能なサーバを何百台と用意しても、エミュレータごとに必要となる GUI 描画やセンサエミュレーションのオーバーヘッドが大きく、総合的なリソース消費が増大する。結果として、数百万台を再現するには膨大なマシンリソースと運用管理コストが必要となり、現実的ではない。

2.3 App の自動操作手法

App の大規模展開時のシミュレーションでは、単に App を実行するだけでなく、実行中の App に対してユーザ操作をどのように与えるかが重要となる。実際の利用状況を再現するには、App が提供する機能や GUI をユーザが操作する流れを可能な限り忠実に模倣すべきである。既存の App の自動操作手法は、本来は App の GUI テストを自動検証するために開発されたものである。GUI テストは通常、労力とリソースを要するものであり、以前までは手動でテストが行われていた。これに対して、GUI テストを自動化するための多くの技術、ツール、フレームワークが開発されてきた [17]。

しかし、既存の App の自動操作手法は、元来は単体テストや中小規模の GUI 検証を主目的として開発されたものであり、大規模シミュレーションでの利用を想定して開発されていない。そのため、数万〜数百万クラスの大規模シミュレーションにおいては以下のような問題が生じる可能性がある。

- 画面レンダリングや UI イベント処理に付随するオーバーヘッドの増大
- 多数のインスタンスに対して操作を制御・同期するための管理コストの肥大化

- ランダムテストや単純スクリプトでは複雑な利用シナリオを再現しにくい

したがって、大規模展開時のシミュレーションを前提とすると、これらの既存自動操作手法をそのまま活用するのは難しいと考えられる。本節の以下のサブセクションでは、具体的な自動操作手法の種類と特徴を整理しながら、実際に大規模シミュレーションを行う上で想定される課題について述べる。

2.3.1 UI 要素を利用した手法

App の自動操作手法として、画面上の UI 要素に付与された識別子（ID、クラス名、タグ名など）を基に、特定の要素を認識して操作を行う手法がある。これにより、UI ツリーから要素を特定し、ボタンのクリックやテキスト入力などの操作をエミュレートし、手動操作に近い動作を再現可能である。また、この手法は、異なるデバイスや OS に対して一貫したテストプロセスを提供し、UI 要素が一意に識別可能である限り高精度な検証が可能である。この手法を用いた代表的なツールとして、Appium[18] や Selenium[19] のようなツールが挙げられる。

しかし、大規模実行を想定すると以下の課題が顕著になる。

- UI 要素の取得や状態の監視には GUI 描画リソースが必要不可欠であり、数万・数百万規模で並列実行する際には過剰な計算資源や GPU 資源を要する
- テストシナリオが画面遷移や UI レイアウトの変更に影響を受けやすく、App の更新時にスクリプトのメンテナンス負荷が増大する

2.3.2 画像認識を利用した手法

画像認識を活用した自動検証手法では、画面上の要素を画像としてキャプチャし、それを基に要素を特定して操作を実行する [20]。この手法は特に UI 要素に一意の識別子が存在しない場合や、動的に生成される要素を扱う際に有効である。視覚的な要素（例：アイコンやボタンの外観）に基づいて操作を自動化でき、UI の変更にも柔軟に対応できる点が大きな利点である。代表的なツールとして SikuliX[21] が挙げられる。

しかし、大規模実行を想定すると以下の課題が顕著になる。

- 画面キャプチャや画像認識処理が頻繁に行われるため、数百万インスタンス規模での同時実行を想定すると、この手法は過大なオーバーヘッドを伴い現実的ではない。
- 端末や画面解像度の違いにより認識精度が低下する可能性があり、多種多様な端末を対象とする大規模シミュレーションでは設定や調整の煩雑さが課題となる。

2.3.3 ランダムテストを用いる手法

ランダムテストを用いる手法は、App の異常終了や不具合の検出を目的として、ランダムに UI イベントを送出する手法である。代表的な技術として、Monkey Testing[22] や Monkey Runner[23] が挙げられる。Monkey Testing は、App に対してランダムにタップやスワイプなどのイベントを送出し、クラッシュや異常挙動を誘発する手法である。Monkey Runner は、スクリプトを通じて一定の制御下で操作を行うことが可能なフレームワークである。これらの手法は、GUI を描画しながらイベントキューを管理するエミュレータや実機を対象に適用される。

しかし、大規模実行を想定すると以下の課題が生じる。

- ランダムな操作では業務シナリオや緊急時の利用シナリオを再現しにくい
- 多数の GUI 環境を同時に立ち上げる際に、計算資源やネットワーク帯域の消費が著しく増大する
- GUI 描画のリソース負荷が高く、数万〜数百万インスタンス規模の並列実行には適さない

ランダムテストは、性能評価や障害検知を目的としたスモールスケールのテストには有用であるが、数万〜数百万規模の大規模シミュレーションへの適用は現実的ではない場合が多い。

第3章　提案手法

本章では、App の大規模展開時のシミュレーションを容易に実施するための提案手法について説明する。

3.1　概要

既存の物理デバイスやエミュレータを用いた手法では、数百万台規模の App を同時に実行するうえでスケーラビリティの問題が生じる。特にエミュレータは、ディスプレイや入力デバイス、センサといったハードウェア機能をソフトウェアでエミュレートするため、大きな計算リソースを消費してしまう。その結果、大規模な同時実行が困難となる。そこで本研究では、エミュレータを使用せずに App を実行可能とするため、App を「ヘッドレス化」して GUI 部分を除去する手法を提案する。ヘッドレス化を施した App は「vApp」と定義し、ディスプレイや入力デバイスといったハードウェアへの依存を排除することで、より大規模かつ効率的な同時実行を実現可能にする。一方で、ヘッドレス化に伴い、これまで GUI やハードウェア機能を通じて行われていたユーザー操作やセンサ情報の受け渡し手段を再構築しなければならない。

本研究では、以下の 3 点に着目して vApp を設計し、スケーラブルな大規模シミュレーションを可能にする。

- App のヘッドレス化の方法
- vApp に対する I/O モデル
- デバイス特有のセンサ情報の取り扱い方法

上記の考慮事項を踏まえて vApp を構築し、Agent と連携させることで、実環境に近いユーザー操作や動的な負荷状況を再現した大規模シミュレーションを実施可能とする。

3.2　App のヘッドレス化の方法

App をヘッドレス化する方法として、以下の 2 つを考える。

- フロントエンドを全て除去する方法: App のフロントエンドに関するすべての部分を除去し、コアロジックのみを残す。この方法では、ユーザー入力を処理するロジックも含めてフロントエンドを除去するため、最も徹底的なヘッドレス化となる。

- フロントエンドを再構築する方法: App のフロントエンドを再構築し、GUI に依存しない形でユーザー入力を処理できるようにする。この方法では、ユーザー入力処理のロジックを保持しつつ、視覚的な表示部分のみを除去する。

これらの方法はいずれも App の GUI を除去し、バックエンドプロセスとして実行することを可能にするため、ヘッドレス化の一形態とみなすことができる。ただし、ヘッドレス化の程度やアプローチが異なる。フロントエンドは、App のユーザーが直接操作する部分を指し、GUI だけでなく、ユーザーインターフェースを通じてユーザー入力を処理するロジックも含まれる。一方で、App における UI に依存せずに動作し、App の本質的な機能を担う部分をコアロジックとする。

3.2.1 フロントエンドを全て除去

App からフロントエンドを全て除去し、App のコアロジックのみを残す構成とする。この構成を図 3.1 に示す。ユーザー操作をエミュレートする Agent は、コアロジックに対して直接メッセージを送信し、操作を実行する。しかし、この構成では、フロントエンドのすべての要素が除去されるため、フロントエンドに依存するユーザー入力処理のロジックをシミュレーションで検証できないという問題がある。そのため、フロントエンドにおけるユーザー入力を処理するロジックに潜むバグが未検出のまま残る可能性があり、実際のシステムにおいて重大な問題を引き起こすリスクが存在する。ユーザー入力処理のロジックの例として、スロットリング [24] やデバウンス [25] が挙げられる。スロットリングは、短時間に多発するイベント（例えば、スクロールやウィンドウのリサイズ）を制限し、処理負荷を軽減する手法である。これらのロジックが正しく機能しない場合、App のパフォーマンスが低下することや、ユーザーエクスペリエンスが損なわれることがある。したがって、フロントエンドに依存するユーザー入力を処理するロジックの検証が不十分であると、こうした問題が発生するリスクが高まる。

3.2.2 フロントエンドの再構築

ヘッドレス化するために、App からフロントエンドを全て除去し、App のコアロジックのみを残す構成の場合、ユーザー入力を処理するロジックに潜むバグを検出することができないという問題があった。そのため、フロントエンドを全て除去するのではなく、App のフロントエンドを再構築し、GUI に依存しない形で

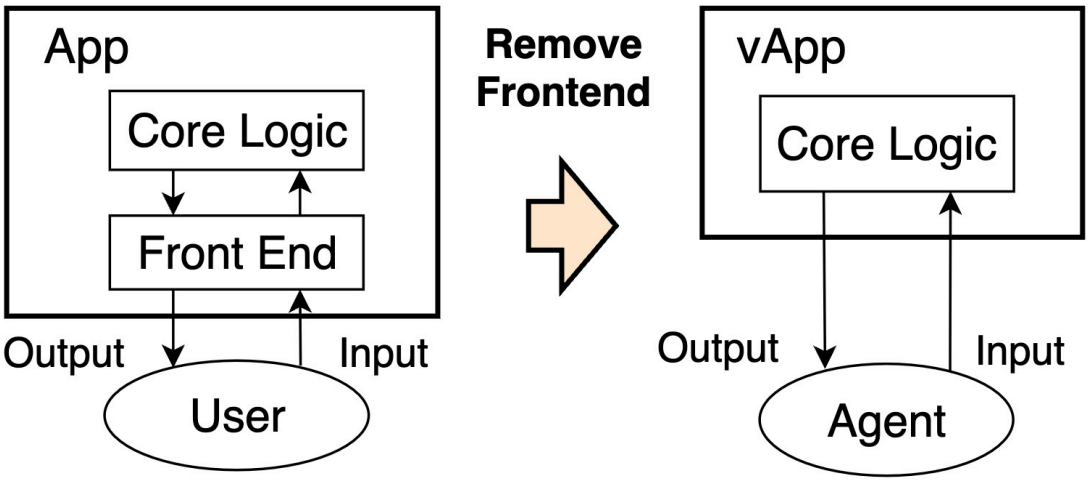

図 3.1: フロントエンドを全て除去

ユーザー入力を処理できるようにする。この方法では、ユーザー入力処理のロジックを保持しつつ、視覚的な表示部分のみを除去する。

フロントエンドの再構築を行う際、抽出する要素を一般化する方法として、フレームワークを用いる方法とデザインパターンを用いる方法を考える。フレームワークは、App の構造を整理するための基盤を提供するものであり、主に次に挙げる二つの仕組みを提供する。

一つ目は UI フレームワークの仕組みである。具体的には、以下に挙げるものである。

- 宣言的な UI の記法やコンポーネント・ウィジェットによる再利用可能な UI パーツの提供
- 状態変化に応じた UI 更新の自動化や簡略化

二つ目は、コアロジックを含むアプリの設計支援である。具体的には、以下に挙げるものである。

- 状態管理・データフロー管理を容易にするための仕組みや推薦パターン
- データフェッチ、ロジック実行と UI 描画を分離しやすい設計が可能
- ネイティブ API アクセスや外部ライブラリとの統合をサポートし、アプリ全体のコアロジックを同じ言語・ランタイムで記述可能にする

具体的な App のフレームワークとして、Xcode[12] の提供する UIKit[26] や Cocoa Touch[27]、Android Studio[13] の提供する Jetpack Compose[28] などがある。

二つ目は、デザインパターンを用いる方法である。デザインパターンは GUI を持つ App に対して用いられるものであり、UI との App のコアロジックを分離しやすく

するための再利用可能な設計指針を指す。代表的なものには、MVC（Model-View-Controller）、MVP（Model-View-Presenter）、MVVM（Model-View-ViewModel）などがあり、いずれのパターンも「関心の分離」を重視し、アプリケーション内の各コンポーネントに明確な役割分担を与えるのが特徴である。

デザインパターンはフレームワークより抽象度が高く、フレームワーク内でも採用されているため、デザインパターンを用いる方がより一般化に適していると考える。よって、フロントエンドの再構築にはデザインパターンを採用した。

MVC モデルの説明

デザインパターンの中で MVC モデルが最も使用されているため [29]、本研究では MVC モデル（ASP.NET）をデザインパターンの例として説明する。MVC モデル（ASP.NET）を図 3.2 に示す。MVC モデルは、アプリケーションを Model、View、Controller の 3 つの部分に分離する [30]。

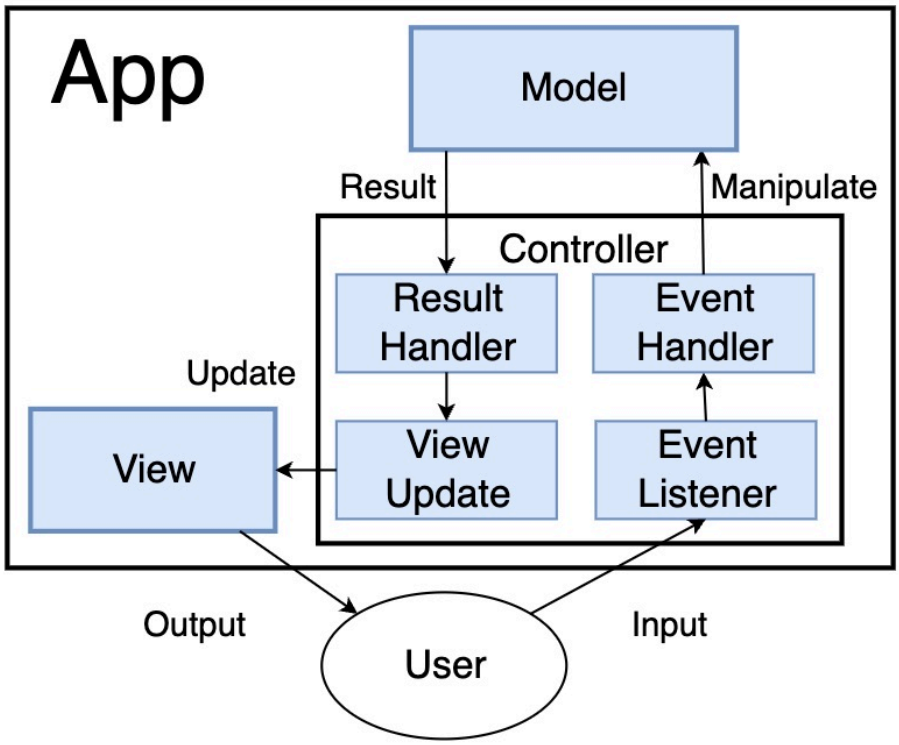

図 3.2: MVC モデル

- **Model**：アプリケーションにおけるデータ管理やコアロジックを担当し、ユーザーの操作やオンラインサービスからの実行によってデータの変更が生じた場合、その結果を Controller に返す。Model は、データ管理やビジネスロジックの実行、データベースへのアクセス、API 通信などの機能を持ち、アプリケーションの中心的な役割を果たす。

- **View：**Controller からの指示に基づいて、画面に対して表示内容のレンダリングを担当する。View は、ユーザーに表示される UI コンポーネントを管理し、Model から提供されたデータや Controller の指示に従って、ユーザーに対して視覚的なフィードバックを提供する役割を持つ。

- **Controller：**ユーザーからの入力を受信し、その入力を処理するロジックを担当する。Controller は、ユーザーのアクションを受信すると、対応する Model へのデータ操作を指示する。また、Model から返された結果を受信し、それに基づいて適切な View を選択し、View に対して表示を更新する指示を送信する。Controller には、Event Listener、Event Handler、Result Handler、View Update といった内部要素が含まれる。**Event Listener** は、ユーザーからの操作入力を検知するプログラムである。例として、ボタンのクリックやフォームの送信などである。**Event Handler** は、Event Listener が検知したイベントに対応して呼び出される。Event Handler は呼び出されると、検知したイベントに対応した、Model のコアロジックを呼び出す。加えて、スロットリング [24] やデバウンス [25] といった入力を制御するロジックなども内包する。**Result Handler** は、Model が処理を完了すると、処理結果を Model から受信する。受信した内容に基づいて、対応する View Update の処理を呼び出す。**View Update** は、Result Handler に呼び出された際に、View の更新を実行する。

再構築によって抽出する要素

フロントエンドの再構築の概略を図 3.3 に示す。App をヘッドレス化するために、MVC モデルにおいて Model および Controller の特定の内部要素を抽出する。以下にそれぞれの要素における再構築によって変更された点について述べる

- **Model：**変更なし。

- **View：**画面表示の必要がなくなるため、削除する。

- **Event Listener：**View がヘッドレス化によって除去されるため、ユーザーから操作を受信する事ができない。そのため、Agent から操作を受信することができるように修正する。

- **Event Handler：**変更なし。

- **Result Handler：** View Update が削除されるため、Model から受信した結果に基づいて対応する View Update を呼び出すことができない。そのため、Model から受信した結果を変換し、Agent に送信できるように修正する。

- **View Update：**View が削除されるため、削除する。

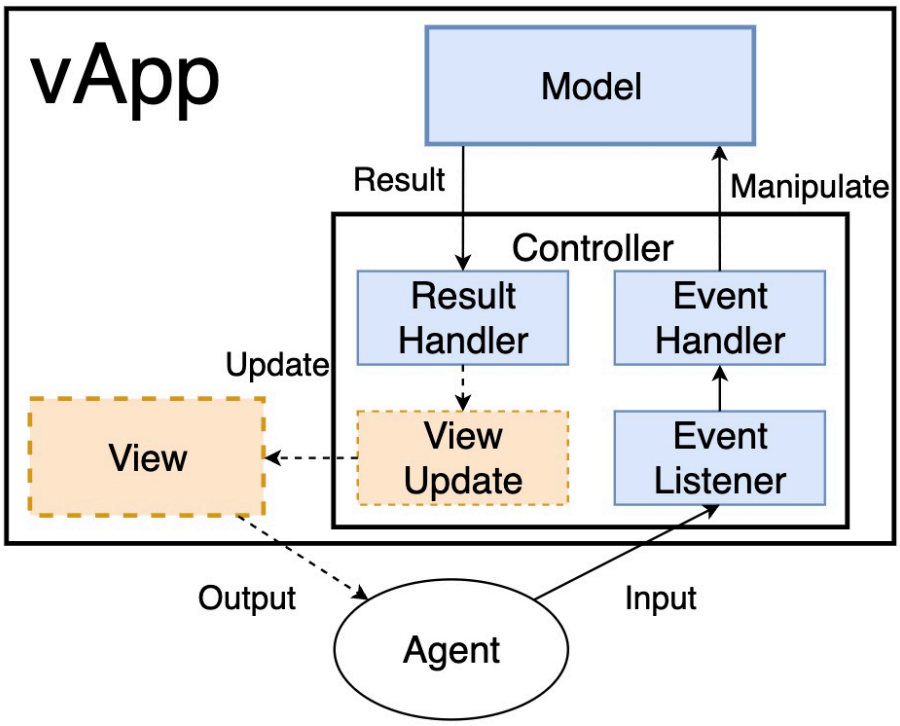

図 3.3: フロントエンドの再構築

3.3 エージェント指向シミュレータ（Agent）の導入

本章では、まず vApp の操作機構の必要性を整理し、続いて操作機構として用いるエージェント指向シミュレータ（Agent）の概要と利点について述べる。

通常の App への操作は、スクリーンに対する指でのタッチや、入力デバイスを用いてユーザーによって操作が実現される。しかし、vApp はスクリーンや入力デバイスなどのハードウェアに依存せずに実行されるため、新たに vApp に対する操作を実行する機構を確立する必要がある。さらに、この操作機構は vApp を大規模実行した際のシミュレーションにも用いるため、手動で vApp の操作を実行することは人的リソースの観点から現実的ではない。よって、vApp の操作を実行する仕組みは自動であることが不可欠となる。既存の自動操作手法の多くは、GUI 要素を認識して操作する方式であるため、ヘッドレス環境では利用できない。また、ランダムテストのように vApp に対しても適用可能な手法も存在するが、簡易的なスクリプトによる操作を実行するだけであり、ユーザーの動作を十分にシミュレートできず、実際のユーザー行動に対する忠実性が低いという課題がある。以上を踏まえ、本研究ではヘッドレス環境である vApp にも対応可能で、かつ実際のユーザーの行動をある程度忠実に再現できる操作機構として、Agent を用いることとした。

Agent とは、事前に設定されたシナリオやアルゴリズムに基づいて特定のタスクを自律的に実行するソフトウェアプログラムを指す。各 Agent は事前に設定さ

れた行動シナリオに基づき、vApp に対する操作プログラムを実行し、I/O モデルを通じて vApp へ入力を送信する。従来の自動操作は、あらかじめ設定されたテストスクリプトやランダム操作に依存することが多く、実使用シナリオを十分に再現できない場合がある。これに対し、Agent はユーザがどのような手順や頻度で App を操作するかを柔軟にモデル化できるのが大きな利点である。たとえば、災害時に利用される App のケースでは「避難行動」「SNS 投稿」「拠点情報の確認」など複数の行動パターンを組み合わせることで、実際に近い複雑な操作シナリオの再現が可能となる。また、Agent は内部状態（位置、状況、行動意図など）を持ち、それに基づいて行動を変化させることができるため、単なるランダムテストよりも現実的で多様な操作をシミュレーションに反映できる。

3.4 vApp に対する I/O モデル

大規模シミュレーションで vApp を使用する場合、各 Agent は 1 つの vApp と対になって動作する。しかし、vApp は GUI を持たないため、従来のタッチ操作や画面遷移といった入出力機構が使えない。そこで、Agent と vApp の間でメッセージ通信による操作と情報交換を行う必要があり、その具体的な設計を「I/O モデル」として定義する。

本研究では、この I/O モデルとして「vScreen Model」と「vOps Model」を提案する。これらのモデルの概要を図 3.4 に示す。どちらのモデルが適しているかは App の種類によって異なる。そのため、提案手法では両方のモデルを採用し、実装する App の特性に応じて適切なモデルを選択する方針とした。

3.4.1 vScreen Model

以下に vScreen Model の概要及び、画面の表示情報を Agent で用いる必要性について述べる。

vScreen Model の概要

vScreen Model は、画面構成と遷移を意識したモデルである。具体的には、vApp 内部に「Send Input and View Data」というコンポーネントを導入し、Agent 側には「Input List」および「View Data」を持たせる。このモデルでは、Controller の要素である Result Handler が、Model のコアロジックによって処理された結果を受信し、その内容に応じて以下の適切な処理を行う。

- **インタラクティブ要素の場合：**Model から受信した結果が、ボタン、テキスト入力、チェックボックスのようなフォーム関連の要素やインタラクティブ要素として処理されるデータの場合、新たに追加した要素である Send Input

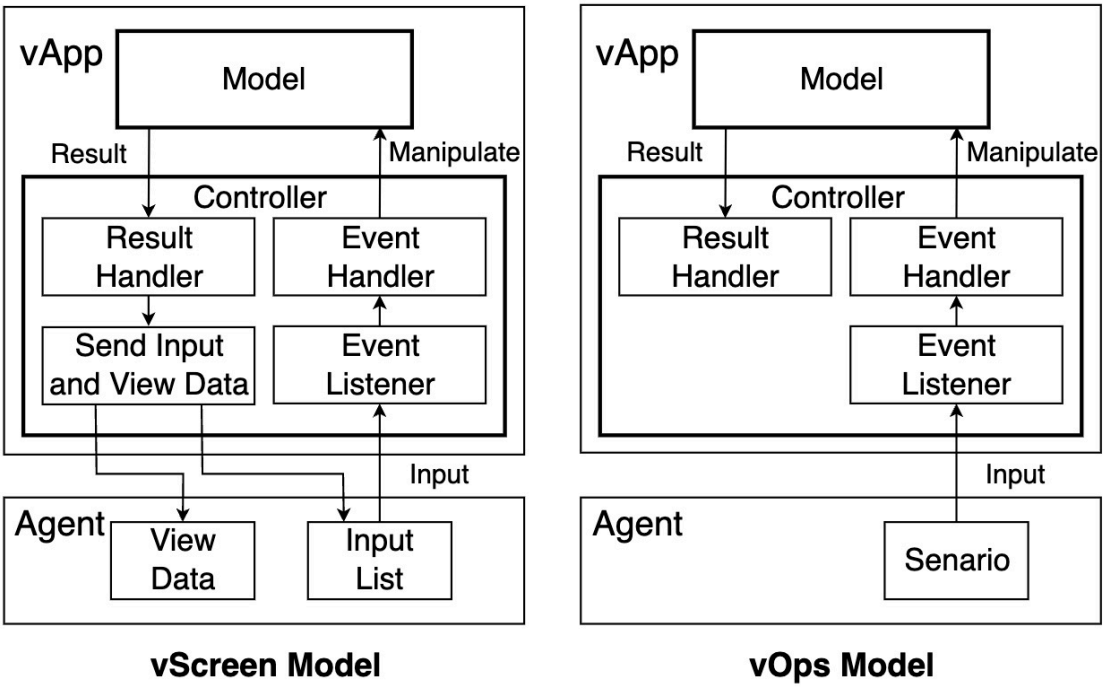

図 3.4: vScreen Model 及び vOps Model

and View Data に渡され、そこから Agent の Input List に送信される。その際 Agent は、Input List がもつリストから入力を選択し、vApp に送信する。

- **非インタラクティブ要素の場合：** Model から受信した結果が、テキスト、画像、メディアのような非インタラクティブ要素として処理されるデータであり、Agent の動作に影響を与えるデータであった場合、新たに追加した要素である Send Input and View Data に渡され、そこから Agent の View Data へ送信される。その際 Agent は、View Data を利用してユーザーのシミュレーションを実行する。

このモデルは、インタラクティブ要素及び、Agent の動作に影響を与えるテキストやイラストのような非インタラクティブ要素を Agent に対して送信することにより、次の二点を可能にする。一つ目は、vApp 内のコアロジックの処理結果に応じて Agent は操作入力を動的に変化させることが可能になることである。二つ目は、Agent の動作を vApp が生成する画面表示内容に応じて変化させることが可能である。そのため、より現実に近いシミュレーションが可能となる。

一方問題点として、表示する内容が多い App の場合は、作成する Input List の量が増加し、実装難易度が上がることに加え、シミュレーションの進行に影響を与えることが予想される。

したがって、このモデルはシミュレーションに忠実度が要求される場合や、画面遷移や表示される内容自体が限られている場合に有効なモデルであると言える。

画面の表示情報を Agent で用いる必要性

通常の App においては、ユーザーが画面に表示された情報を見て操作を判断する。たとえば、災害時の避難 App では、App が避難経路を画面に表示し、ユーザーは表示された地図やルート情報を参照して移動を決める。しかし、vApp には GUI が存在しないため、本来ユーザーに提示される情報をそのまま Agent に「見せる」ことはできない。もし vApp 側が「画面に表示する」体裁で情報を作成しても、それを Agent が視覚的に受け取る仕組みがないため、予期しない動作や不整合が生じる恐れがある。よって、Agent 側の動作のプログラムに対して、画面表示によってユーザーに伝達しようとしていた内容を送信するための仕組みが必要である。

そこで、vApp 内部で画面表示用として生成されるテキストやイラストなどの非インタラクティブ要素を活用する。本来の App であれば画面描画に利用される情報（地図上の座標や表示テキストなど）を「画面表示と同等の意味を持つデータ」として Agent に送信する。Agent はこれを直接受け取り、ユーザーが画面を見て意思決定を行うかのような動作を再現する。こうして、GUI を除去した vApp でも、ユーザーが必要とする情報を Agent へ渡せるようになる。

3.4.2 vOps Model

vOps Model は、vApp が提供するサービスを重視したモデルである。Agent は、あらかじめ持っているシナリオに基づいて、vApp に対して操作の入力及び動作を実施する。Agent は既にシナリオを持っているため、vApp からの View 更新を利用する必要がない。

このモデルは、事前に定義されたシナリオを基に操作を行うため、リアルタイムな状態管理が不要であり、実装の難易度が低いということに加え、計算負荷が低いという特徴がある。

一方問題点として、Agent が事前に定義されたシナリオに基づいて vApp への操作入力及び動作を実施するため、vApp 側の動作として入力を受信するタイミングとは異なるタイミングで入力を実行する場合がある。また、災害時の避難経路表示 App のような画面表示がユーザー動作に影響する App の場合、Agent は画面の表示内容を行動に反映することができず、忠実度の低いシミュレーションとなる。

そのため、シミュレーションの忠実度がそれほど要求されない場合や限られた計算資源の中で大規模実行をするべき場合に有効なモデルであると言える。

3.5 デバイス特有のセンサ情報の取り扱い

ここでは、本研究においてスマートデバイス特有の GPS・バッテリーなどのセンサ情報をどのように扱うかを説明する。また、vApp と Agent 間でセンサ情報を交換する 2 種類のモデル（Asynchronous / Synchronous）を示す。

3.5.1　センサ情報の取り扱いの必要性

一般的に、物理デバイスやエミュレータ上で App を実行する場合は、スマートデバイスに搭載されている GPS などのセンサ情報や、オペレーティングシステムが管理するバッテリー残量などの状態情報を App から直接取得できる。そして、App は取得した情報を内部ロジックで用いる。しかし、vApp は物理デバイスやエミュレータを利用しないため、これらのセンサ情報が物理的に存在しない。そこで提案手法では、センサ情報を新たに生成して vApp に提供する仕組みを導入する。具体的には、Agent がユーザーをシミュレートしている点に着目し、Agent が各種センサ情報を「擬似的に生成」して vApp へ送信する方針をとった。これにより、実機やエミュレータを用いずとも、vApp は必要なセンサ情報を受け取り、あたかも物理デバイス上で動作しているかのように振る舞うことができる。

3.5.2　センサ情報の交換モデル

センサ情報は vApp と Agent 間で交換されるため、これらの交換モデルは同様のモデルを使用する。これらの交換モデルとして、「Asynchronous Model」と「Synchronous Model」を提案した。提案した交換モデルの概念図を図 3.5 に示す。

次にこれらのモデルについて説明する。センサ情報は生成された後にキャッシュされる。このキャッシュするエンティティを本研究において「State」として定義する。State は vApp と Agent のどちらの側にも存在し、それぞれの側で使用される。送信される State と、受信側のキャッシュされる State を区別するために、受信側の State を「Shadow State」と定義した。以下に提案したモデルの詳細について説明する。

Asynchronous Model

Asynchronous Model は、vApp と Agent がそれぞれ独立したタイミングで State を送受信するモデルである。このモデルは、State の送受信を非同期に扱うため、vApp や Agent の内部処理状況に左右されない。このモデルにおける、State の交換手順は以下の通りである。

1. 「State」側は、生成された情報を State に格納する。
2. 「State」側は、State に格納された情報を「Shadow State」に対して送信する。
3. 「Shadow State」側は、受信した情報を自身の Shadow State にキャッシュする。
4. 「Shadow State」側は、必要に応じて Shadow State を参照して使用する。

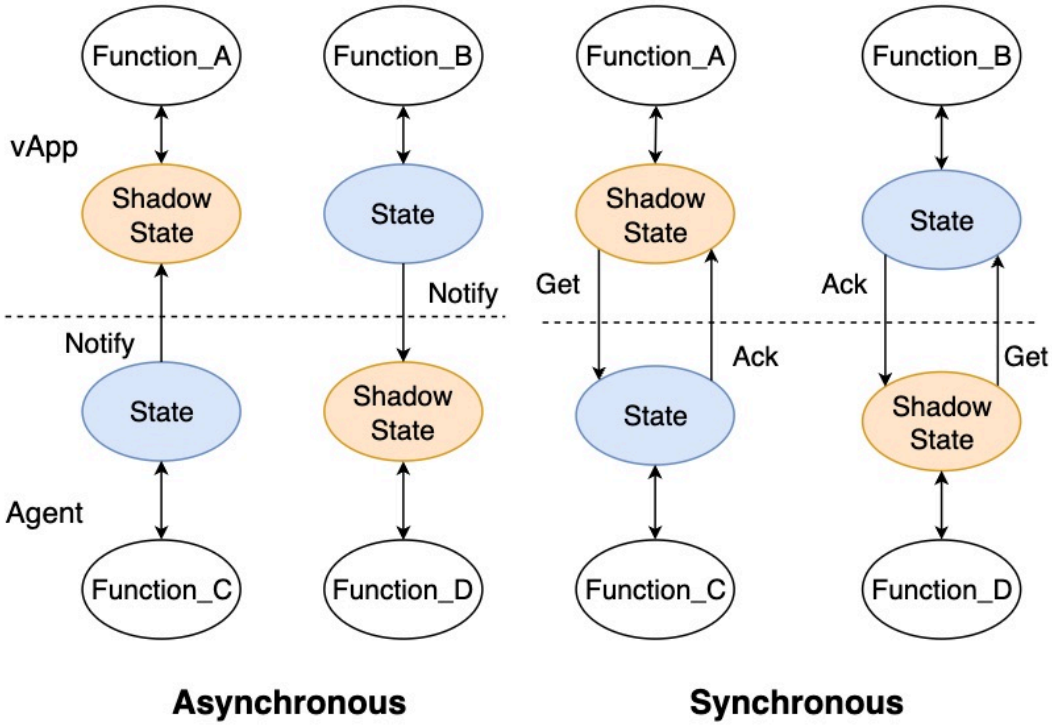

図 3.5: State の交換モデル

このモデルでは、vApp と Agent が互いに独立したタイミングで State を送受信するため、各プログラムの進行速度が異なっていても影響を受けにくく、大規模シミュレーション環境でも柔軟に動作可能である。一方で、時系列を厳密にそろえたい場合には、State が非同期に到着することから、明示的な同期手段やトレーサビリティの確保が課題となる。

Synchronous Model

Synchronous Model は、vApp と Agent が同期的に State を送受信するモデルである。このモデルは、State の送受信を同期的に扱うため、vApp や Agent の内部処理に左右される。このモデルにおける、State の交換手順は以下の通りである。

1. 「State」側は、生成された情報を State に格納する。
2. 「Shadow State」側は、State を利用するために「State」側に情報を要求する（Shadow State 側は情報を受信するまで待機）。
3. 「State」側は、State に格納されている情報を送信する。
4. 「Shadow State」側は、受信した情報を Shadow State に格納する。
5. 「Shadow State」側は、Shadow State を参照して使用する。

このモデルでは、vApp と Agent が同期的に State をやり取りし、交換が完了するまで Shadow State 側は待機する。その結果、時系列を厳密に合わせてシミュレーションを進行できる利点があるが、プログラムの進行速度に差がある場合は待機時間が発生し、全体の効率が低下する可能性がある。本研究では、後述の理由によりシミュレーションの効率を重視するため、Asynchronous Model を採用した。

第4章　例題Appへの適用

提案手法の概念実証を行うために、例題 App への適用を行った。以下に実装した App の概要及び、提案手法で設計した vApp、vApp に対する I/O モデル、デバイス特有のセンサ情報の取り扱いの実装方法について以下で述べる。

4.1　概要

本章では、まず例題 App の全体像を示し、続いて vApp の具体的な実装、vApp と Agent の I/O モデル、そしてセンサ情報の扱いについて順に述べる。実装したプログラムの詳細ソースコードは付録に掲載している。

4.1.1　例題 App の概要

本例題 App は、災害時の避難用 App である。システムは vApp、Agent、Backend Server で構成される。システムを構成するそれぞれの要素について以下で述べる。

vApp

vApp は避難上 App を提案手法で述べた設計に従ってヘッドレス化したものである。主に Agent からの操作の受信及び、Backend Server からの情報の取得をする。以下で vApp の持つ機能を大まかに述べる。

- Agent が生成する位置情報を受信し、Backend Server から位置情報に応じた地図情報と避難所リストを取得する。
- Backend Server から受信した避難所リストを Agent に送信する。
- Agent から選択した避難所を受信し、Agent の位置から Agent が選択した避難所への避難経路を生成する。
- 生成した経路を Agent に送信する。

Agent

Agent は、vApp を利用して避難する避難者をシミュレートしたものである。主に、vApp からの画面情報の取得及び、避難者のシミュレーションをする。以下で Agent の持つ機能を大まかに述べる。

- 位置情報を生成・変更し、Agent に送信する。
- vApp から避難所リストを受信し、リスト内から選択した避難所を vApp に送信する。
- vApp から受信した避難経路に応じて、自身の位置情報を変更する。

Backend Server

Backend Server は、vApp とは異なるネットワーク上に存在し、App からのリクエストに対してデータを返信するシステムである。vApp から位置情報を受信し、その位置情報に応じた地図情報と避難所のリストを返信する機能を持つ。したがって、Backend Server では地図情報と避難所のリストをデータベースも管理している。

4.1.2 実装概要

実装するシステムの概念図を図 4.1 に示す。Backend Server などの提案手法で設計した内容の範疇から外れるものは概念図に含まないものとした。実装する vApp は、スマートフォン向けのネイティブアプリケーションであり、MVC モデル [30] のデザインパターンを採用するものとする。MVC モデルに沿ったプログラムをするためのフレームワークは用いず、提案手法で述べた通りに自身で MVC モデルの要素に当てはめて実装した。

すべての実装には JavaScript[31] を用いる。理由は二つある。一つ目は、Apache Cordova[32] を用いることで、実装した vApp に GUI を付与して App とし、スマートフォン用のネイティブアプリケーションに変換することが可能だからである。二つ目は、サポートプラットフォームとして、Android、iOS などのクロスプラットフォームに対応しているからである。JavaScript の実行環境には node.js[33] を用いた。

4.2 vApp

本節では、vApp の具体的な実装方法について説明する。その一環として、実装した vApp に GUI を付与するプログラムを追加し、App とした場合の例を示す。

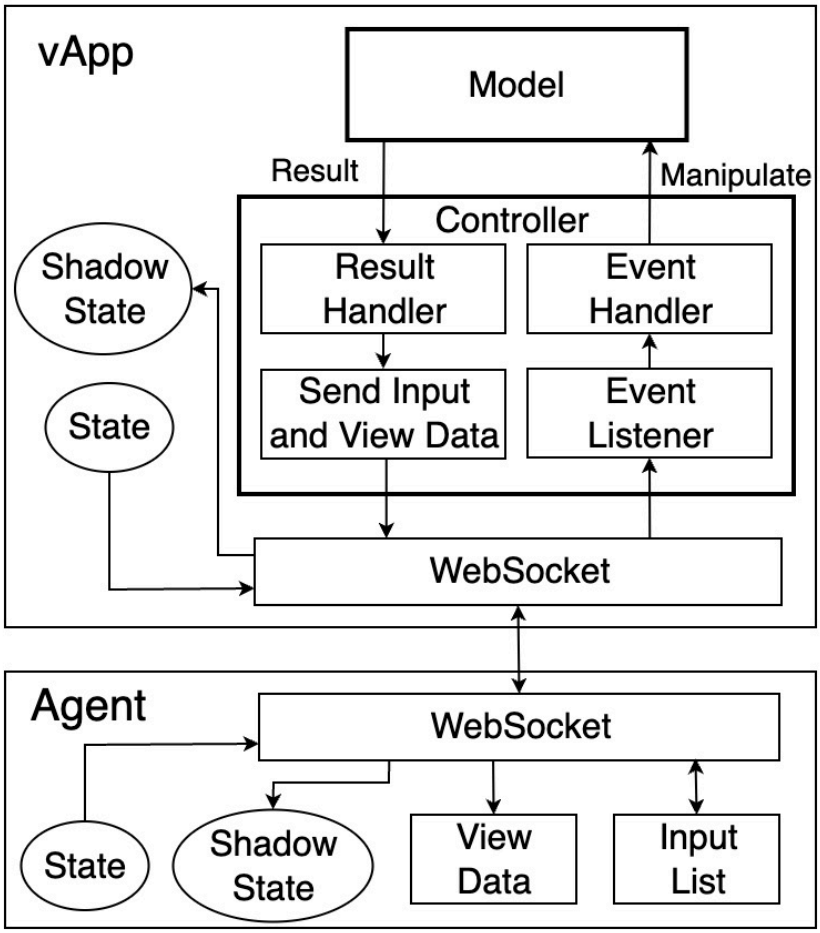

図 4.1: 例題 App のアーキテクチャ

これにより、App と vApp のプログラムの構造の違いを明確にする。App のソースコードの一部を図 4.2、4.3 に示し、vApp のソースコードの一部は、図 4.4、4.5 に示した。なお、これらのリストのソースコードは、本節の説明において本質的でない部分を大幅に省略している。

図 4.2、4.3 では、(1)〜(6) の各要素が MVC モデルに対応しており、それぞれ順に View、Model、Event Listener（Controller）、Event Handler（Controller）、Result Handler（Controller）、View Update（Controller）である。図 4.2、4.3 における各要素の機能を以下で述べる。

- **View：**本来の機能は、HTML で画面表示の内容を持ち、id タグを用いて UI コンポーネントの管理するというものである。図 4.2、4.3 の場合を説明する。Controller クラス内の updateRoute() によって、View クラス内で管理している UI コンポーネントである、id=”route”の値が変更された場合に、それを HTML に反映させるという機能を持つ。

- **Model：**本来の機能は、コアロジックを担当し、Event Handler から呼び出さ

れ、コアロジックの結果を Result Handler に送信するというものである。図 4.2、4.3 の場合を説明する。Controller クラス内の eventHandler() から Model クラス内の generateRoute() が呼び出され、内部においてロジックが発動する。そこで生成された結果をコールバック関数を用いて、Controller クラス内の resultHandler() を呼び出すという機能を持つ。

- **Event Listener**：本来の機能は、ユーザーから操作を受信し、操作の内容を Event Handler に送信するというものである。図 4.2、4.3 の場合を説明する。View クラスにおいて定義されている button がユーザーによって click が実行されるとそれを受信し、Controller クラスの eventHandler() を呼び出すという機能を持つ。

- **Event Handler**：本来の機能は、Event Listener から呼び出され、受信した Event の種類に応じて Model のロジックを呼び出すというものである。図 4.2、4.3 の場合を説明する。Controller クラス内の `this.view.shelter1.addEventListener()` から eventHandler() が呼び出され、eventType が clickShelter1、clickShelter2、clickShelter3 のいずれかであれば Model クラス内の generateRoute() を呼び出し、eventType が clickEvacComp であれば Model クラスの evecComp() を呼び出すという機能を持つ。

- **Result Handler**：本来の機能は、Model のコアロジックから呼び出され、結果の種類に応じて View Updata を呼び出すというものである。図 4.2、4.3 の場合を説明する。Model クラス内の generateRoute() からコールバック関数を通じて呼びだされ、result の種類が updateRoute の場合に Controller クラスの updateRoute() を呼び出し、result の種類が updateShelter の場合、updateShelter() を呼び出すという機能をもつ。

- **View Update**：本来の機能は、Controller の Result Handler から呼び出され、変更されたデータに基づいて View のもつ UI コンポーネントを変更するというものである。図 4.2、4.3 の場合を説明する。Controller クラス内の resultHandler() から updateRoute() が呼び出され、View クラスにおける this.route という UI コンポーネントの値を変更するという機能を持つ。

一方、図 4.4、4.5 では、vApp を実装するためにフロントエンドの再構築を実施し、図 4.2、4.3 と比較して削除・変更されている要素がある。(1)〜(4) はそれぞれ順に Model、Event Listener（Controller）、Event Handler（Controller）、Result Handler（Controller）となっている。ただし、各要素の機能には変化はないため、図 4.4、4.5 における変更点のみ以下で説明する。

- **View**：削除。

- **Model**：変更なし。

- **Event Listener：**「App」では、View クラスにおいて定義されている button がユーザーによって click が実行されるとそれを受信し、Controller クラスの eventHandler() が呼び出されていた。「vApp」では、View クラスが存在しないため、WebSocket[34] によって Agent から操作を受信し、Controller クラスの eventHandler() を呼び出すように変更した。

- **Event Handler：**変更なし。

- **Result Handler：**「App」では、Model クラス内の gentrateRoute() からコールバック関数を通じて呼びだされ、result の種類が updateRoute の場合に Controller クラスの updateRoute() を呼び出し、result の種類が updateShelter の場合、updateShelter() を呼び出されていた。「vApp」では、View Update が存在しないため、Agent に対して WebScoket によって結果を送信するように変更した。vApp における Result Handler の実装に関しては、4.3 章で詳細に述べる。

- **View Update：**削除。

4.3 vApp に対する I/O モデル (vScreen Model)

例題 App を用いた、vApp に対する I/O モデルの実装例を説明する。提案手法において、vApp に対する I/O モデルとして「vScreen Model」と「vOps Model」を提案した。vOps Model は非常にシンプルなモデルであり、例題 App として説明に用いる必要がないと判断し、例題 App には vScreen Model を実装した。vScreen Model は Controller の Result Handler が、Model のコアロジックによって処理された結果を受信し、その内容に応じて処理を行う。結果の内容がボタン、入力フォームなどのインタラクティブ要素の場合、Input List に変換し、新たに追加した要素である Send Input and View Data に渡され、この要素によって Agent の Input List に送信される。また、結果の内容がテキストやイラストなどの非インタラクティブ要素であり、Agent の動作に影響を与えるデータであった場合、Send Input and View Data 渡され、Agent の View Data に送信される。

図 4.6 に vApp における vScreen Model に関するソースコードを載せた。図 4.6 内の (1) の Result Handler では、Model のコアロジックによって処理された結果を受信し、内容に応じて Send Input and View Data を呼び出している。図 4.6 内の (2) の Send Input and View Data では、WebSocket[34] を用いて Agent に対して Input List 及び、View Data を送信している。sendSheltersToAgent() が Input List を送信し、sendRouteToAgent() が View Data を送信している。

図 4.7 に Agent における vScreen Model に関するソースコードを載せた。図 4.7 内の (1) の onSheltersData() において、Input List として避難所のリストの受信を

```
-----------------------(1:View)-------------------------
<!DOCTYPE html>
<body>
 <button id="shelter1">shelter1</button>
 <button id="shelter2">shelter2</button>
 <button id="shelter3">shelter3</button>
 <button id="evacComp">Complete Evacuation</button>
 <div id="route"> Route Information</div>
 <script src="app.js"></script>
</body>
</html>
class View {
  constructor() {
    this.shelter1 = document.getElementById('shelter1');
    this.shelter2 = document.getElementById('shelter2');
    this.shelter3 = document.getElementById('shelter3');
    this.route = document.getElementById('route');}
}
-----------------------------------------------------------
-----------------------(2:Model)-------------------------
class Model {
  constructor() {this.onResultHandlerCallback = null;}
  generateRoute() {
    msg = {type: "updateRoute",payload: this.route};
    this.onResultHandlerCallback(JSON.stringify(msg));}
  evacComp() {}
}
-----------------------------------------------------------
```

図 4.2: App のソースコード (1:View 及び Model)

```
class Controller {
  constructor(model) {
    this.model = model;
    this.clickShelter1 = "clickShelter1";
    // Shelter2 and Shelter3 programs omitted.
    this.clickEvacComp = "clickEvacComp";
    model.onResultHandlerCallback = (result) => this.
       resultHandler(result);
------------------(3:Event Listener)--------------------
    this.view.shelter1.addEventListener('click', (
       shelter1) => this.eventHandler(shelter1));
    // Shelter2 and Shelter3 programs omitted.
    this.view.evacComp.addEventListener('click', (
       evacComp) => this.eventHandler(evacComp));}
------------------------------------------------------------
------------------(4:Event Handler)---------------------
  eventHandler(eventType) {
    switch (eventType) {
      case "shelter1" || "shelter2" || "shelter3":
        this.model.generateRoute();
      case "evacComp":
        this.mode.evecComp();}}
------------------------------------------------------------
------------------(5:Result Handler)--------------------
  resultHandler(result) {
    data = JSON.parse(result);
    switch (data.type) {
      case "updateRoute":
        updateRoute(data.payload);
      case "updateShelter":
        updateShelter();}}
------------------------------------------------------------
------------------(6:View Update)-----------------------
  updateRoute(r){this.view.route.innerText=`${r}`}
  updateShelter(data){}
------------------------------------------------------------
}
```

図 4.3: App のソースコード (2:Controller)

している。(2) では、避難所のリストから避難所の選択をし、vApp に対し送信をしている。(3) の onRouteData において、View Data として経路情報を受信している。(4) では、経路情報を用いて Agent が避難の動作をし、位置情報を変更している。

4.4 センサ情報の交換モデル（Asynchronous Model）

例題 App を用いた、デバイス特有のセンサ情報の取り扱いの実装例を説明する。提案手法において、センサ情報の交換モデルとして、「Asynchronous Model」と「Synchronous Model」を提案した。例題 App には Asynchronous Model を実装した。Asynchronous Model は、Agent と vApp 間でセンサ情報を非同期的に交換するモデルである。センサ情報は State として生成された後にキャッシュされ、相手側に非同期で送信される。相手側で受信されたセンサ情報は Shadow State としてキャッシュされ、必要に応じて使用される。図 4.8 に Agent における Asynchronous Model に関するソースコードを載せた。図 4.8 内の (1) の generateLocation() においてセンサ情報として GPS をシミュレートして、擬似的に位置情報を生成し、sendAgentLocation() を呼び出している。(2) の sendAgentLocation() において vApp に対してセンサ情報を WebSocket を用いて送信している。図 4.9 に vApp における Asynchronous Model に関するソースコードを載せた。図 4.9 内の (1) の eventHandler() において、Agent からセンサ情報を受信し、updateAgentLocation() を呼び出している。(2) の updateAgentLocation() において受信した位置情報をキャッシュとして保存している。

```
-----------------------(1:Model)-----------------------
class Model {
  constructor() {this.onResultHandlerCallback = null;}
  generateRoute(start, end) {
    this.onResultHandlerCallback(this.route);}
  evacComp(){
    this.onResultHandlerCallback();}
}
-------------------------------------------------------
```

図 4.4: vApp のソースコード (1:Model)

```
class Controller {
  constructor(model) {
    this.model = model;
    model.onResultHandlerCallback = (result) => this.
       resultHandler(result);
------------------(2:Event Listener)--------------------
    this.wss = new WebSocket.Server({ port: 3000 }, ()
       => {});
    this.agentSocket = null;
    this.wss.on("connection", (ws) => {
      this.agentSocket = ws;
      ws.on("message", (msg) => {
        this.eventHandler(msg);});
    })}
-----------------------------------------------------------
------------------(3:Event Handler)--------------------
  eventHandler(eventType) {
    switch (eventType) {
      case "selectedShelter":
        this.model.generateRoute();
      case "evacComp":
        this.model.evaComp();}}
-----------------------------------------------------------
------------------(4:Result Handler)--------------------
  resultHandler(result) {
    data = JSON.parse(result);
    switch (data.type) {
      case "sendSheltersToAgent":
        sendSheltersToAgent();
      case "sendRouteToAgent":
        sendRouteToAgent(data.payload);}}}
-----------------------------------------------------------
}
```

図 4.5: vApp のソースコード (2:Controller)

```
class Controller {
------------------(1:Result Handler)--------------------
  resultHandler(result) {
    data = JSON.parse(result);
    switch (data.type) {
      case "sendSheltersToAgent":
        sendSheltersToAgent();
        break;
      case "sendRouteToAgent":
        sendRouteToAgent(data.payload);
        break;
      default:
    }
  }
-----------------------------------------------------------
-------------(2:Send Input and View Data)---------------
  sendSheltersToAgent() {
    const msg = {
      type: "sheltersData",
      payload: this.model.shelters
    };
    this.agentSocket.send(JSON.stringify(msg));
  }

  sendRouteToAgent(route) {
    const msg = {
      type: "routeData",
      payload: route
    };
    this.agentSocket.send(JSON.stringify(msg));
  }
}
-----------------------------------------------------------
```

図 4.6: vApp における vScreen Model の実装

```
class Agent{
-----------------(1:Receive Input List)-----------------
  onSheltersData(shelters) {
    const idx = Math.floor(Math.random() * shelters.
       length);
    const chosen = shelters[idx];
    this.selectedShelter = { lat: chosen.lat, lng:
       chosen.lng };
    this.sendSelectedShelter();
  }
-----------------------------------------------------------
---------------------(2:Send Input)---------------------
  sendSelectedShelter() {
    const msg = {
      type: "selectedShelter",
      payload: this.selectedShelter
    };
    this.socket.send(JSON.stringify(msg));
  }
-----------------------------------------------------------
------------------(3:Receive View Data)-----------------
  onRouteData(route) {
    this.shadowRoute = route;
    this.stepCount = 0
    this.followRoute();
  }
-----------------------------------------------------------
-------------------(4:Use View Data)--------------------
  followRoute() {
    this.moveIntervalId = setInterval(() => {
        this.stepCount++;
    }, 5000);
  }
}
-----------------------------------------------------------
```

図 4.7: Agent における vScreen Model の実装

```
class Agent{
--------------------(1:Generate Sensor Info)-------------
  generateLocation() {
    const lat = 35.68 + (Math.random() - 0.5) * 0.01;
    const lng = 139.767 + (Math.random() - 0.5) * 0.01;
    this.agentLocation = { lat, lng };
    this.sendAgentLocation();
  }
-----------------------------------------------------------
--------------------(2:Send Sensor Info)-----------------
  sendAgentLocation() {
    const msg = {
      type: "agentLocation",
      payload: this.agentLocation
    };
    this.socket.send(JSON.stringify(msg));
  }
-----------------------------------------------------------
}
```

図 4.8: Agent における Asynchronous Model の実装

```
class Model {
-------------------(2:Cache Sensor Info)----------------
  async updateAgentLocation(location) {
    this.shadowAgentLocation = location;
    }
  }
-----------------------------------------------------------
}
class Controller {
-------------------(1:Receive Sensor Info)--------------
  async eventHandler(event) {
    data = JSON.parse(event);
    switch (data.type) {
      case "agentLocation":
        // の位置をに更新AgentModel
        this.model.updateAgentLocation(data.payload);
        break;
      default:
    }
  }
-----------------------------------------------------------
}
```

図 4.9: vApp における Asynchronous Model の実装

第5章　各手法の比較による評価

App を大規模展開した際のシミュレーションを行うため、各手法の比較評価を実施した。この評価により、各手法のパフォーマンスを複数の観点から比較し、長所と短所を明らかにする。これにより、最適な手法を選択するための指針を得ることを可能とする。

5.1　比較手法

比較手法は、表 5.1 に示した 5 つの手法を用いる。実行環境、アプリケーション形態、操作形態によって手法の分類を行った。実行環境には、物理デバイス、エミュレータ、汎用サーバがある。物理デバイスの機能要件は App の動作が可能であることであり、ローエンドモデルの Android スマートフォンとする。エミュレータは Cuttlefish[2] を用いる。汎用サーバは、費用や所要時間の計算が容易であることから、クラウドサービスを利用することとした。

アプリケーション形態は、App、App のコアロジックのみ、vApp の 3 種類である。操作形態は、人による操作と自動操作がある。自動操作では、通常の App の場合は Appium[18] を用い、コアロジックのみ及び vApp では独自の自動操作方法を用いることとした。

表 5.1: 比較手法の一覧

手法	実行環境	アプリケーション形態	操作形態
1	物理デバイス	App	手動
2	物理デバイス	App	自動
3	エミュレータ	App	自動
4	汎用サーバ	コアロジックのみ	自動
5	汎用サーバ	vApp	自動

5.2 評価項目

評価項目は以下とし、各評価項目は五段階で評価を行う。数値に基づく評価が可能な項目については、実験や計算によって得られた数値を用いて定量的な評価を行う。

- **計算リソース:**100 万インスタンスを実行する際に必要な計算リソースを評価。
- **所要時間:** 100 万インスタンスを実行するための準備から実施までに要する時間を評価。
- **実行費用:** 100 万インスタンスを実行する際に必要となるデバイスの購入費などの合計金額を評価。
- **自動化の容易さ:** App を自動操作するためのツール開発・設定の容易さ及び、自動化するための環境の設定の容易さを評価。
- **デバッグの容易さ:** デバッグに用いるログの内容・取得方法・管理方法によって容易さを評価。
- **規模性:** App のインスタンス数を増加させる際の容易さを評価（実行後の運用は考慮しない）。
- **忠実性:** 物理デバイス上で App を手動で操作する利用者が、急激に増加する状況をどれだけ忠実にシミュレーション可能であるかを評価。
- **実装オーバーヘッド:** 完全な App にするために必要な追加実装の規模を評価。

5.3 結果

各評価項目に対する各手法の評価を表 5.2 に示す。

5.3.1 計算リソース

各手法の単位計算リソースにおける最大実行インスタンス数及び、100 万インスタンスの実行に必要な計算リソースの個数を表 5.3 に示す。手法 1、2 の 100 万インスタンスの実行に必要な計算リソースの個数は、物理デバイスを単位計算リソースとし、手法 3、4、5 は表 5.4 に示す実験環境を単位計算リソースとし、算出した。ただし、手法 4、5 の計算リソースの差は誤差であるため、手法 4 の計算リソースは、手法 5 で算出したものと等しいものとした。

手法 3、5 は 100 万インスタンスの実行に必要な計算リソースの個数を算出するために、単位計算リソースにおける最大実行インスタンス数を実験結果を基に算

表 5.2: 各手法の評価結果

評価項目	手法 1	手法 2	手法 3	手法 4	手法 5
計算リソース	1(1M)	1(1M)	2(50,000)	5(6,250)	5(6,250)
所要時間 [ヶ月]	1(19)	1(17)	4(5)	5(2.5)	5(2.5)
実行費用 [$]	1(1.95B)	1(1.56B)	2(28.08M)	5(1.17M)	5(1.17M)
自動化の容易さ	1	2	3	5	5
デバッグの容易さ	2	3	5	4	4
規模性	1	1	3	5	5
忠実性	5	3	3	1	3
実装オーバーヘッド	5	5	5	1	3

表 5.3: 各手法の 100 万インスタンス実行に必要な計算リソース数

手法	最大インスタンス/単位リソース	100 万インスタンスのリソース
1, 2	1	1,000,000
3	20	50,000
4, 5	160	6,250

出した。実験環境では、4 つの CPU を使用しているため、CPU 使用率の上限は 400%であり、CPU 使用率の測定は以下の方法で実施した。

- CPU 使用率の取得コマンド：ps コマンド
- CPU 使用率の計算方法：各測定時点での全インスタンスの CPU 使用率の合計を取り、平均値を算出
- CPU 使用率の測定間隔：10 秒ごとに測定
- 測定期間：実行開始から 600 秒間（合計 60 回測定）

以下に手法 3、5 の実験方法及び、単位計算リソースにおける最大実行インスタンス数の算出方法を示す。

手法 3

単位計算リソースにおける最大実行インスタンス数は、以下に示す式で算出した。

$$\text{最大実行インスタンス数 (20)} = \left\lfloor \frac{CPU \text{ 使用率の上限 (186\%)}}{\text{単位インスタンスの } CPU \text{ 使用率 (8.92\%)}} \right\rfloor$$

表 5.4: 実験環境の詳細

項目	値
CPU モデル	Intel(R) Xeon(R) E5-2650
物理 CPU 数	4
物理コア数	4
論理スレッド数	4
クロック周波数	2.00 GHz
メモリ容量	8 GB
ストレージ容量	38 GB

cpu 使用率の上限は、後述する手法 5 の実験で求めた最大実行インスタンス数の cpu 使用率を使用した。単位インスタンスの CPU 使用率は以下の三つの要素の合計から算出した。

- エミュレータの CPU 使用率：6.71[%]
- vApp の CPU 使用率：1.84[%]
- GUI のオーバーヘッド：$1.84 \times 0.2 \fallingdotseq 0.37$[%]

エミュレータの CPU 使用率は、表 5.4 に示した実験環境において Cuttlefish[2] を実行した際の CPU 使用率の結果を用いた。vApp の CPU 使用率は手法 5 の実験結果を用いた。GUI のオーバーヘッドは、vApp の CPU 使用率の 20%とし、vApp の CPU 使用率に加算した。

手法 5

単位計算リソースにおける最大実行インスタンス数は、vApp のインスタンス数を 1, 10, 50, 100, 125, 150... と増加させ、CPU 使用率が飽和したインスタンス数を用いた。実験結果を図 5.1 に示す。実験結果から、165 インスタンスで CPU 使用率が減少しているため、最大実行インスタンス数は 160 とした。

5.3.2 所要時間

思考実験から算出した、各手法の所要時間及び、100 万インスタンスを実行するまでの手順を表 5.5 に示す。

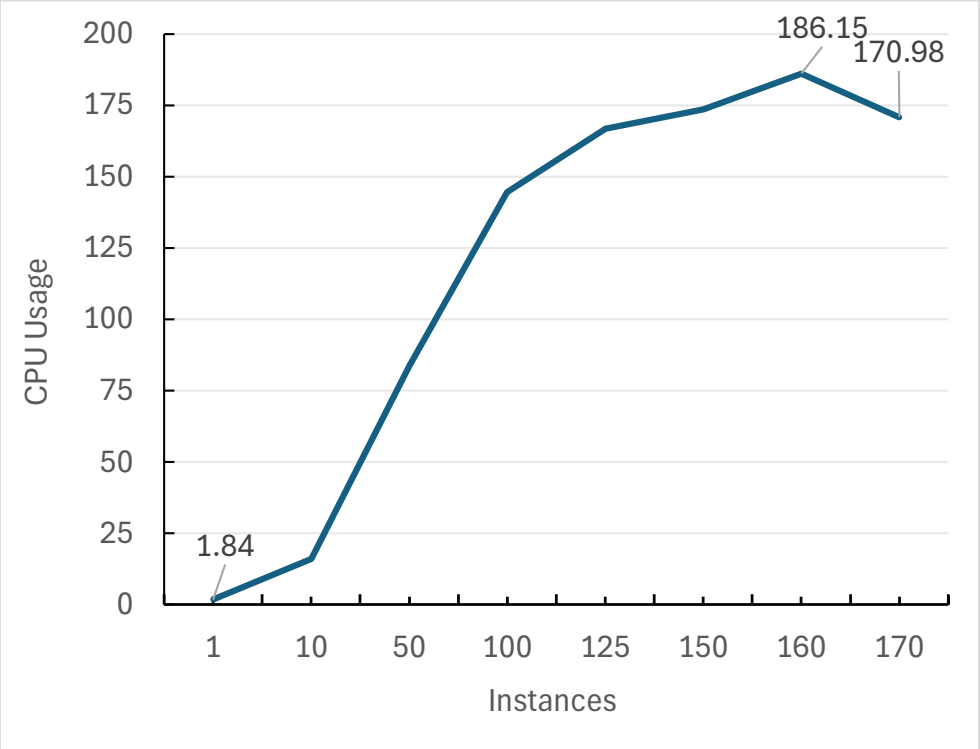

図 5.1: 提案手法の実行結果

手法 1

手法 1 は、分散したユーザーにスマートフォンを配布し、各ユーザーに App のインストールと実行を行ってもらい、大規模実行を実施する場合を想定した。スマートフォンの調達に 10ヶ月を要する。これは 100 万台のデバイスを製造・検品・初期設定するためであり、大量生産に伴うリードタイムや物流計画が関係している。参加者募集には 5ヶ月を要し、広告配信や応募受付などの活動を行う。スマートフォンの発送には 2ヶ月を要し、配送準備と管理を行う。App のインストールには 1ヶ月を要し、各参加者がデバイスを起動してオンラインストアから App をインストールする。App の実行は 1 日で完了し、実行タイミングを同期して行う。最後に、スマートフォンの回収には 1ヶ月を要し、返送スケジュールの調整やデバイスの回収を行う。

手法 2

手法 2 は、スマートデバイスを一箇所に集約し、App の大規模実行を行う場合を想定する。スマートフォンの調達には手法 1 と同様に 10ヶ月を要する。インフラ環境の構築には 5ヶ月を要し、デバイス管理用の物理インフラを準備する。自動化ツールの開発には 1ヶ月を要し、App の操作やデバイス設定を自動化するプログラムを作成する。テストと検証には 1ヶ月を要し、動作確認と性能試験を行う。

表 5.5: 各手法の所要時間と手順

手法	所要時間	手順
1	19ヶ月	デバイス調達 (10ヶ月)、参加者募集 (5ヶ月)、配送 (2ヶ月)、設置 (1ヶ月)、回収 (1ヶ月)
2	17ヶ月	デバイス調達 (10ヶ月)、インフラ構築 (5ヶ月)、ツール開発 (1ヶ月)、テスト (1ヶ月)
3	5ヶ月	クラウド調達 (1 週)、環境設定 (1 週)、ツール開発 (1.5ヶ月)、パッケージ作成 (1 週)、テスト (2ヶ月)
4	2.5ヶ月	クラウド調達 (1 週)、環境設定 (2 週)、ツール開発 (1ヶ月)、パッケージ作成 (1 週)、テスト (2 週)

手法 3

手法 3 は、コスト計算の容易さと短い調達期間を考慮し、クラウド上の計算リソースを用いて大規模実行を行う場合を想定する。クラウドリソースの調達には 1 週間を要する。クラウド環境の構築には 4 週間を費やし、ネットワーク設定やエミュレータの配置、自動デプロイの構築を行う。計算リソースの評価より、50,000 個のインスタンスを使用するため、環境構築とテストに時間を要する。自動化ツールの開発には 1.5ヶ月を要し、エミュレータ上で App を自動操作するプログラムや、App とエミュレータを自動展開・実行するプログラムを開発する。App のパッケージ化には 1 週間を要し、コンテナでパッケージ化し、大規模デプロイを容易にする。テストとパフォーマンスチューニングには 1ヶ月を費やし、大規模なエミュレータ環境での動作確認と最適化を行う。

手法 4,5

手法 4 は、手法 3 と同様にクラウド上の計算リソースを使用するが、使用するインスタンス数は 6,250 と手法 3 の約 8 分の 1 である。クラウドリソースの調達は手法 3 と同様に 1 週間を要する。環境構築には 2 週間を要し、計算リソースが少ないため環境構築とテストの期間も短縮される。自動化ツールの開発には 1ヶ月を要し、コアロジックを自動操作するプログラムを作成する。App のパッケージ化には 1 週間を要し、コンテナでパッケージ化する。テストと検証には 2 週間を費やし、スケーラビリティとパフォーマンステストを実施する。App の大規模実行は 1 日で完了する。手法 5 は手法 4 と同様のプロセスであり、所要時間も同じである。

5.3.3 実行費用

思考実験から算出した、各手法の 100 万インスタンスの実行に必要な合計コストと主な費用項目を表 5.6 に示す。為替レートは 1 ドル＝ 155.62 円（2024 年 11 月 20 日時点）とした。

表 5.6: 100 万インスタンスの実行費用

手法	コスト項目	内訳	合計 [$]
1	デバイス費用	\$153 × 100 万台	1,530 M
	物流費用	\$0.3 × 100 万台	0.3 M
	広告費用	\$65 K × 5ヶ月	0.325 M
	報酬費用	\$30 × 100 万人	300 M
	パッケージ費用	\$1.5 × 100 万台	1.5 M
	配送費用	\$5 × 100 万台	5 M
	回収パッケージ費用	\$1.5 × 100 万台	1.5 M
	回収配送費用	\$5 × 100 万台	5 M
合計			1,950 M
2	デバイス費用	手法 1 と同じ	1,530 M
	設備レンタル費用	\$7/m^2/月 × 0.05 m^2 × 100 万 × 5	1.75 M
	ラック費用	\$700 × (100 万 / 500 台)	1.4 M
合計			1,560 M
3	クラウド使用料	\$0.26/時間 × 2,160 時間 × 50,000	28.08 M
合計			28.08 M
4, 5	クラウド使用料	\$0.26/時間 × 720 時間 × 6,250	1.17 M
合計			1.17 M

手法 1

所要時間の評価と同様に、分散したユーザーにスマートフォンを配布し、各ユーザーに App のインストールと実行を行ってもらい、大規模実行を実施する場合を想定した。使用する物理デバイスは、ローエンドスマートフォンで販売台数の多い Xiaomi 社の Redmi 12C[35] とした。実験には単一のアプリが実行できれば十分であるため、この機種を選択した。価格は 23,800 円である（2024 年 11 月 4 日時点）。

手法 2

所要時間の評価と同様に、スマートフォンを一箇所に集約し、App の大規模実行を行う場合を想定した。デバイスの費用に加え、デバイスを設置するためのラッ

クの費用、スマートフォンを一箇所に集約して実行するための設備のレンタル費用を追加した。レンタルの期間は所要時間の評価で算出した5ヶ月を用いた。

手法 3,4,5

所要時間の評価と同様にクラウド上の計算リソースを用いて大規模実行を行う場合を想定した。利用するクラウドプロバイダーは AWS[36] とし、インスタンスタイプは C4.xlarge（オンデマンド料金：0.26$/h、東京リージョン）を用いる。これは、計算リソースの評価で使用した CPU と性能及び数が同等であるためである。各手法のインスタンス数は、計算リソースの評価で算出したものを用いた。利用時間は、所要時間の評価で算出した環境構築とテストに要する期間の合計を用いた。手法 4 と手法 5 は所要時間が等しいため、実行費用も同様に等しくした。

5.3.4 自動化の容易さ

App を自動操作するためのツール開発・設定の容易さ及び、自動化するための環境の設定の容易さを評価した。

実行環境ごとの自動化の容易さを比較する。物理デバイスを用いる場合、物理デバイス専用の管理スクリプトや接続制御が必要となり、自動化の難易度が上がる。エミュレータを用いる場合、物理的制約が少なく、管理スクリプトや接続制御の開発が容易であるため物理デバイスを用いるよりも自動化が容易となる。

操作形態ごとの自動化の容易さを比較する。手動操作する場合は自動化が非常に困難であり、自動化の容易さは極めて低い。物理デバイスを操作するロボットなども考えられるが、導入コストが非常に高額となる上に台数を揃えることが困難であることから、現実的ではない。自動操作は Appium などのツールを用いることが可能になるため、手動操作と比較して自動化は容易である。

アプリケーション形態ごとの自動化の容易さを比較する。コアロジックのみおよび vApp は、GUI を排除することで自動操作をするプログラムが画面遷移やユーザー操作の追跡を必要がなくなるため、自動操作プログラムがシンプルなものとなる。これにより、自動化が容易となる。

5.3.5 デバッグの容易さ

デバッグに用いるログの内容・取得方法・管理方法によって容易さを評価した。実行環境ごとのデバッグの容易さを比較する。物理デバイスは端末ごとにログを取得する必要があり、大規模展開する場合では、非常にログの管理が煩雑になる。それにともなってデバッグも困難になる。エミュレータには標準のデバッグ機能が搭載され、エミュレータごとのログの取得・管理が容易でありログの内容も明確となるため、デバッグが容易である。汎用サーバでは同じ OS 上に複数のアプリ

ケーションを展開するため、ログの取得と管理を一元化することが可能であるため、デバッグが容易である。

アプリケーション形態ごとのデバッグの容易さを比較する。コアロジックのみ及び、vApp は GUI を含まないプログラムであるため、プログラムの複雑性が低く、ログの内容がシンプルであるためデバッグが容易となる。さらに、コアロジックのみは、vApp と比較してユーザー入力処理を含まないため、よりログの内容がシンプルになる。

操作形態ごとのデバッグの容易さを比較する。手動操作では、人が操作するため不確定要素も多く、ログの内容が明確ではなくなるためデバッグが困難になる。自動操作では、実行された操作を記録および追跡できるため、問題発生時にどの操作が影響したかを特定しやすく、効率的なデバッグが可能となる。

5.3.6 規模性

App のインスタンス数を増加させる際の容易さを評価した。

各実行環境における規模性を比較する。物理デバイスでは、App のインスタンスごとに専用のデバイスが必要となるため、最もスケーラビリティが低い。一方、汎用サーバは複数の App インスタンスを同時に実行できるため、より高いスケーラビリティを持つ。エミュレータも汎用サーバ上で複数のインスタンスを同時に動作させることが可能だが、1 つのエミュレータインスタンスにつき 1 つの App しか実行できない。そのため、大規模な運用には多くの計算リソースを要する。

アプリケーション形態ごとの規模性を比較する。コアロジックのみ及び vApp は、App から GUI を排除したことで軽量となったため、App よりも規模性が高い。

操作形態ごとの規模性を比較する。手動による操作では物理デバイスの実行数の人員が必要となるため、規模性は非常に低くなる。

5.3.7 忠実性

物理デバイス上で App を手動で操作する利用者が、急激に増加する状況をどれだけ忠実にシミュレーション可能であるかを評価した。実行環境ごとの忠実性を比較する。物理デバイスを用いる場合に最も忠実性が高くなる。エミュレータは OS やハードウェアをエミュレートしているため、汎用サーバよりも忠実性を確保可能である。汎用サーバはサーバ用の OS 上でセンサは独自の機構でエミュレートするため、忠実度は低くなる。

アプリケーション形態ごとの忠実性を比較する。コアロジックのみは、忠実性が低い。これは、GUI ロジックが動作しないため、現実の動作環境とは異なる挙動がシミュレーション結果に影響を与えるためである。vApp はコアロジックのみに入力処理のロジックを追加しているため、コアロジックのみより忠実性が高くなる。

操作形態ごとの忠実性を比較する。人が操作する手法の方が現実の利用者行動に近い負荷を再現できるため、忠実性が高くなる。これは、人手による操作が実際の利用者の動作をより忠実に再現するためである。

5.3.8 実装オーバーヘッド

完全な App にするために必要な追加実装の規模を評価した。アプリケーション形態ごとの実装オーバーヘッドを比較する。App は実装オーバーヘッドがないものとする。コアロジックのみはフロントエンドをすべて除去しているため、実装のオーバーヘッドが非常に大きくなる。vApp はフロントエンドにおけるユーザー入力を処理するロジックがすでに実装されているため、GUI のみを実装するのみで良い。

5.4 考察

評価結果から、提案手法（手法 5）は他の手法と比較して多くの面で優位性を示した。以下の点について考察する。まず、計算リソースと規模性について、提案手法は汎用サーバ上で軽量な vApp を実行することで、100 万インスタンスの大規模シミュレーションを現実的なコストと時間で実現可能であることが示された。手法 1、2 では、物理デバイスの調達や管理に莫大なコストと時間が必要であり、手法 3 でもオーバーヘッドが大きく、リソース効率が低い。

実行費用と所要時間の観点では、提案手法はクラウド上の計算リソースを効率的に利用することで、他の手法と比較して大幅なコスト削減と時間短縮が可能であることを示した。特に、手法 3 と比較して約 1/24 のコスト効率化が図られており、現実的なシミュレーションの実施が可能となっている。

自動化の容易さに関しては、GUI を排除することで自動操作をするプログラムが画面遷移やユーザー操作の追跡を必要がなくなるため、自動操作プログラムがシンプルなものとなる。これにより、Appium などの App の自動操作ツールと比較して、自動操作ツールの開発や設定が容易であることを示した。

しかし、忠実性の面では、提案手法が人が操作する手法や物理デバイス・エミュレータを用いた手法に劣る可能性が示唆された。これは、提案手法ではユーザーの操作パターンや、デバイス固有の挙動を完全にシミュレートすることが難しいからである。これらの要因がシミュレーション結果に影響を及ぼす可能性がある。

また、実装オーバーヘッドに関しては、vApp から完全な App へ変換する際に必要となる GUI の実装が課題として挙げられる。提案手法では、フロントエンドの再構築によりユーザー入力処理のロジックを保持しているものの、最終的な App として完成させるためには追加の開発工数が必要である。

第6章　おわりに

6.1　今後の展望

今後の展望としては、以下の点が挙げられる。

6.1.1　App から vApp の自動生成機構の実現

App のフレームワークを用いて App から vApp の自動生成機構の実現である。フレームワークは App の構造を整理するための基盤を提供するものであり、UI フレームワークやコアロジックを含むアプリの設計支援の仕組みを提供するものである。

フレームワークにはデザインパターンが用いられ、デザインパターンはフレームワークよりも抽象度が高い。そのため、各フレームワークはどのデザインパターンでも用いることが可能となっている。

したがって、フレームワーク内におけるデザインパターンの各要素を特定し、その要素を vApp の設計通りに削除・修正する仕組みによって、vApp を自動生成することが可能であると考える。この仕組みにより、状況に応じて App と vApp を切り替えることが可能となり、両方 App の検証をシームレスにすることが可能となる。この仕組みの実現のために、フレームワークにおけるデザインパターンの要素の識別方法を今後検討する必要があると考える。

6.1.2　Agent の高度化

提案手法の忠実性を向上させるには、App を操作するユーザーの行動パターンをより現実に近づけて再現する手法の検討が必要である。そのため、Agent の動作を決定する際に、センサーでは直接検知できない人間の心理的要素（例: 緊張や焦燥の度合い）や、外部環境の影響（例: 周囲の混雑状況、環境条件、災害時の特殊な状況）を変数として取り入れる手法を確立する必要があると考える。このような情報は、実際のユーザーが App を操作する際、自然に意思決定に影響を与えている。したがって、これらの情報を Agent が処理できるようにすることで、シミュレーションの忠実性をさらに向上させ、現実の使用状況に近い挙動を再現できると考える。

6.1.3　大規模シミュレーションの実施

本研究では数百万インスタンスの App の大規模展開時のシミュレーションを可能にするための手法を提案したが、実際に提案手法を用いてこのような規模でのシミュレーションの実施をすることができなかった。したがって、提案手法を用いて実際に数百万インスタンス規模のシミュレーションを行い、その有効性と実用性をさらに検証する必要があると考える。

6.1.4　vApp の適用可能領域の明確化

提案手法では、App をヘッドレス化することにより大規模実行を可能にした。これはサーバー負荷が大きくなるような App や画面レンダリングよりもコアロジックの検証が重要な App には適用が有効であると考える。しかし、高度な UI/UX が主要な価値を担う App には適用が難しいのではないかと考える。このような App をヘッドレス化によって GUI を排除すると、App の本質的な機能を検証することが不可能になるからである。このように提案手法が適用可能な App の種類を明確化する必要があると考える。

6.2　まとめ

本研究では、スマートデバイス向けアプリケーション（App）の大規模展開時のシミュレーションを容易にするため、ヘッドレス化による App の構成方法を提案した。従来の物理デバイスやエミュレータを用いた手法では、ハードウェアへの依存やオーバーヘッドの問題から、数万から数百万規模の App の同時実行が困難であった。提案手法では、GUI を持たないバックエンドプロセスとして App を実行可能とすることで、汎用サーバ上での大規模かつ効率的なシミュレーションを実現した。

また、App のヘッドレス化に際しては、GUI を持つ App のデザインパターンを活用し、フロントエンドの再構築を行った。これにより、ユーザー入力を処理するロジックを保持しつつ、GUI に依存しない形での App の動作を可能とした。さらに、エージェント指向シミュレータ（Agent）との連携により、ユーザーの操作や環境情報を適切にシミュレートし、現実に近い条件下での評価を可能とした。

評価結果から、提案手法（手法 5）は他の手法と比較して、計算リソースの効率性、所要時間の短縮、実行費用の削減、規模性の向上といった多くの面で優れていることが示された。一方で、忠実性の面では、提案手法はユーザーの操作パターンや、デバイス固有の挙動を完全にシミュレートすることが難しいことから、人が操作する手法や物理デバイス・エミュレータを用いた手法に劣る可能性が示唆された。また、実装オーバーヘッドに関しても、vApp から完全な App へ変換する際に必要となる GUI の実装が課題としてある。

本研究の成果は、ミッションクリティカルな App の大規模展開時における動作検証や負荷試験を効率的に実施するための新たなアプローチを提供するものであり、今後のスマートデバイスアプリケーションの開発・運用における信頼性向上に寄与することが期待される。

謝辞

本研究を行うにあたり, 多くの方から多大なご助言やご助力いただきました。それらの方々のご協力がなければ、本研究は成り立ちませんでした。ここに深く感謝し、心から厚く御礼申し上げます。

本研究を進めるにあたり、主指導教員である篠田 陽一 教授には適切な助言とご指導を賜りました。また、研究に対する指導だけでなく、物事の捉え方や、人に物事を説明する方法など多くのことを学ばせていただきました。国際会議への論文の寄稿、国外での研究発表、学外の研究者との関わりなど、多くの機会を提供してくださいました。深く感謝いたします。

副指導教員の宇多 仁 准教授には、研究室ゼミだけでなく、日々の生活や自主学習で多くの助言をいただきました。深く感謝いたします。また、インターンシップ指導教員である丹 康雄 教授には、中間発表や修士論文審査会において、専門的な立場から的確なご意見をいただきました。深く感謝いたします。

本研究室修了生の片岡 拓海 氏、本研究室の橋 亮真 氏、高田 敦生 氏、吉川 健太 氏、金田 昂大 氏、Yao Yudie 氏、中村 一貴 氏、西村 優典 氏、Lu Xingchen 氏には研究に関する様々な議論や研究生活を送る上での多大なご助力をいただきました。深く感謝いたします。

最後に、学生生活を支えていただいた家族へ心から感謝いたします。修士論文の提出に至るまで多くの方からご支援をいただきました。ありがとうございました。

参考文献

[1] 内閣府政策統括官（防災担当）,“南海トラフ巨大地震の被害想定について.” Accessed: Sep. 06, 2024. [Online]. Available: https://www.bousai.go.jp/jishin/nankai/taisaku_wg/pdf/1_sanko2.pdf.

[2] Android Open Source Project,“Cuttlefish virtual Android devices,” Android Open Source Project. Accessed: Sep. 06, 2024. [Online]. Available: https://source.android.com/docs/devices/cuttlefish.

[3] Amazon Web Services, Inc.,“Automated Testing Tools - AWS Device Farm -,” Amazon Web Services, Inc. Accessed: Aug. 25, 2024. [Online]. Available: https://aws.amazon.com/device-farm/.

[4] Google,“Firebase Test Lab — Test in the lab, not on your users.” Accessed: Aug. 25, 2024. [Online]. Available: https://firebase.google.com/products/test-lab.

[5]“Mobile App Distribution.” Accessed: Jan. 09, 2025. [Online]. Available: https://saucelabs.com/.

[6]“Most Reliable App Cross Browser Testing Platform,” BrowserStack. Accessed: Jan. 09, 2025. [Online]. Available: https://browserstack.com/.

[7]“Anbox,” GitHub. Accessed: Aug. 25, 2024. [Online]. Available: https://github.com/anbox.

[8] now.gg, inc,“BlueStacks: Play Games on PC Mac, Android Emulator and Cloud Gaming Platform,” Bluestacks - The Best Android Emulator on PC as Rated by You. Accessed: Sep. 06, 2024. [Online]. Available: https://www.bluestacks.com.

[9] Genymobile SAS,“Genymotion - Android Emulator in the Cloud and for PC Mac,” Genymotion. Accessed: Aug. 25, 2024. [Online]. Available: https://www.genymotion.com/.

[10] “MEmu - The Best Android Emulator for PC - Free Download,” MEmu Android Emulator. Accessed: Aug. 25, 2024. [Online]. Available: https://www.memuplay.com/.

[11] “Noxplayer – Fastest and Smoothest Android Emulator for PC Mac – Free and Safe.” Accessed: Aug. 25, 2024. [Online]. Available: https://www.bignox.com/.

[12] Apple Inc, “Xcode and Simulators,” Apple Developer Documentation. Accessed: Sep. 06, 2024. [Online]. Available: https://developer.apple.com/documentation/safari-developer-tools/installing-xcode-and-simulators.

[13] Google, “Run apps on the Android Emulator — Android Studio,” Android Developers. Accessed: Sep. 06, 2024. [Online]. Available: https://developer.android.com/studio/run/emulator.

[14] “KVM.” Accessed: Jan. 29, 2025. [Online]. Available: https://linux-kvm.org/page/Main_Page.

[15] “QEMU.” Accessed: Jan. 29, 2025. [Online]. Available: https://www.qemu.org/.

[16] H. Lin et al., “Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility,” in Proceedings of the 29th Annual International Conference on Mobile Computing and Networking, Madrid Spain: ACM, Oct. 2023, pp. 1–17. doi: https://doi.org/10.1145/3570361.3613259.

[17] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, “Graphical user interface (GUI) testing: Systematic mapping and repository,” Information and Software Technology, vol. 55, no. 10, pp. 1679–1694, Oct. 2013, doi: https://doi.org/10.1016/j.infsof.2013.03.004.

[18] OpenJS Foundation, “Welcome - Appium Documentation.” Accessed: Aug. 30, 2024. [Online]. Available: https://appium.io/docs/en/latest/.

[19] SeleniumHQ, “Selenium,” Selenium. Accessed: Aug. 30, 2024. [Online]. Available: https://www.selenium.dev/.

[20] J. Eskonen, J. Kahles, and J. Reijonen, “Automating GUI Testing with Image-Based Deep Reinforcement Learning,” in 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), Aug. 2020, pp. 160–167. doi: https://doi.org/10.1109/ACSOS49614.2020.00038.

[21] "RaiMan's SikuliX," SikuliX by RaiMan. Accessed: Jan. 28, 2025. [Online]. Available: http://sikulix.com/index.html.

[22] T. Wetzlmaier, R. Ramler, and W. Putschögl, "A Framework for Monkey GUI Testing," in 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Apr. 2016, pp. 416–423. doi: https://doi.org/10.1109/ICST.2016.51.

[23] "monkeyrunner — Android Studio," Android Developers. Accessed: Jan. 28, 2025. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner.

[24] mozilla, "Throttle - MDN Web Docs Glossary: Definitions of Web-related terms — MDN." Accessed: Sep. 02, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Throttle.

[25] mozilla, "Debounce - MDN Web Docs Glossary: Definitions of Web-related terms — MDN." Accessed: Sep. 02, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Debounce.

[26] "UIKit," Apple Developer Documentation. Accessed: Jan. 29, 2025. [Online]. Available: https://developer.apple.com/documentation/uikit/.

[27] "Cocoa (Touch)." Accessed: Jan. 29, 2025. [Online]. Available: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html.

[28] "Develop UI for Android — Jetpack Compose," Android Developers. Accessed: Jan. 29, 2025. [Online]. Available: https://developer.android.com/develop/ui.

[29] A. Daoudi, G. ElBoussaidi, N. Moha, and S. Kpodjedo, "An exploratory study of MVC-based architectural patterns in Android apps," in Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, in SAC '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 1711–1720. doi: https://doi.org/10.1145/3297280.3297447.

[30] ardalis, "Overview of ASP.NET Core MVC." Accessed: Sep. 17, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-8.0.

[31] mozilla, "JavaScript — MDN." Accessed: Aug. 29, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[32] “Apache Cordova.” Accessed: Jan. 29, 2025. [Online]. Available: https://cordova.apache.org/.

[33] “Node.js — どこでも JavaScript を使おう.” Accessed: Feb. 01, 2025. [Online]. Available: https://nodejs.org/ja

[34] IETF, “Network Standard Data Specification syntax,” Internet Engineering Task Force, Request for Comments RFC 645, Jun. 1974. doi: https://doi.org/10.17487/RFC0645.

[35] “Redmi 12C - Xiaomi Japan — Mi.com.” Accessed: Jan. 29, 2025. [Online]. Available: https://www.mi.com/jp/product/redmi-12c/.

[36] “アマゾン ウェブ サービス（AWS クラウド）- ホーム,” Amazon Web Services, Inc. Accessed: Jan. 29, 2025. [Online]. Available: https://aws.amazon.com/jp/.

本研究に関する対外発表

- 中川 颯馬, 大規模シミュレーションのためのスマートデバイスアプリケーションの構成法, WIDE Project ポスターセッ ション, May, 2024.
- Soma Nakagawa, Yoichi Shinoda. “A Structure for Smart Device Applications for Large Scale Simulation,” *IA2024 - Workshop on Internet Architecture and Applications 2024*, Taipei, Taiwan, 2024. (査読なし, 発表済み)
- Soma Nakagawa, Satoshi Uda, Yoichi Shinoda. “A Structure for Smart Device Applications for Large Scale Simulation,” *Proceedings of the 39th International Conference on Advanced Information Networking and Applications (AINA 2025)*, 2025. (査読あり, 発表予定)

付録

- **例題 App のソースコード:**https://github.com/malcoscos/SomApp/releases/tag/master-thesis