RMVRVM – A Paradigm for Creating Energy Efficient User Applications Connected to Cloud through REST API

Lavneet Singh
Adjunct Faculty, Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar, Gujarat, India
lavneet_singh@daiict.ac.in

ABSTRACT

The applications that run on resource-constrained devices, especially for batteries, pose a challenge. The activities such applications do while running on such devices consume energy and drain the device's battery. Many of these applications use REST API to communicate with their backend services running outside of the devices, primarily on the cloud. The paradigms like Model View View-Model (MVVM) used on the application side require data transformations that cause applications to consume more battery. There is a need for an improved approach and a paradigm that can be used to develop green software with reduced battery consumption. This paper proposes a novel Remote-Model View Remote-View-Model (RMVRVM) paradigm. The use of RMVRVM paradigm lowers the battery consumption on devices where the application is running and hence contributes to writing green software. In addition, RMVRVM makes an application more responsive and thus a delight to use. This paradigm has been implemented in industrial case studies, and significant gains in terms of the reduced amount of data transfer, reduced battery consumption, and faster response time were observed. Experiments were also done to further validate the paradigm with encouraging results. The practitioners can apply the RMVRVM to design applications for battery-constrained devices with smaller energy footprints and better response times.

CCS CONCEPTS

• **Software and its engineering** → Software creation and management; Designing software; Software design engineering.

KEYWORDS

Sustainable Software Engineering, Green Software, Model View View-Model (MVVM), Energy Efficient Mobile Applications, Battery Saving, Non-functional Requirements

ACM Reference Format:
Lavneet Singh. 2022. RMVRVM – A Paradigm for Creating Energy Efficient User Applications Connected to Cloud through REST API. In *15th Innovations in Software Engineering Conference (ISEC 2022), February 24–26, 2022, Gandhinagar, India.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3511430.3511434

ISEC 2022, February 24–26, 2022, Gandhinagar, India

ACM ISBN 978-1-4503-9618-9/22/02…$15.00
https://doi.org/10.1145/3511430.3511434

1 INTRODUCTION

The use of mobile phones among the masses has increased tremendously during the last decade. Similarly, IoT applications running on sensors, sensor base stations, robots are increasingly becoming useful and popular for everyday use. These applications run on resource-constrained devices, especially for the battery. An application's activities while running on such devices consume energy and drain the device's battery [1]. Many of these applications use REST API [2] to communicate with their backend services running outside the devices, namely in the cloud. These API are generally either have already been designed when desktop applications were pervasive or are designed keeping in mind general data requirements of the application running on different clients. When developers use these API in applications that run on constrained devices, the applications consume more battery due to extra processing required for the data incoming through the API. This results in poor user experience not only because the battery is drained by the running application, but also due to the sluggishness that creeps into the application because of the extra amount of time it takes to process the data.

Current methods of developing such applications use paradigms like MVVM (Model View View-Model) [3],[4] to handle the data received by calls to the REST API. It separates the incoming data into data models for the application and then has converters that process these data model objects to create view models. The view models are bound to the user interface (UI) elements through mechanisms like data binding [5]. Implementing paradigms like MVVM result in application code becoming CPU intensive because of the data transformations required between data models and UI elements through the view models. Additionally, the data received by the application from API may not all be useful for it. Sometimes, it must filter the data after receiving it to put it into the form required by its data model objects. This results in the processing of data to make it into the form that the UI needs. This processing is the root cause of higher battery consumption in applications [6] developed for resource-constrained devices, like mobiles phones.

This paper proposes the RMVRVM (Remote-Model View Remote-View-Model) paradigm to be followed in developing applications for resource-constrained devices. RMVRVM paradigm moves both model and view model of the application on to server-side aka cloud application that provides the REST API that the application running on the device is using. By moving both the model and view model of the application to the cloud, the client-side application running on a resource-constrained device no longer receives unnecessary data from REST API. Hence, it is freed from the responsibility of filtering and transforming data into view models. This paradigm change transforms the application to the one that is frugal, performant,

and consumes a much lesser battery. All these benefits make the application a delight to use by its intended users. RMVRVM helps create greener software as it reduces the battery consumption of applications and hence reduces their carbon footprint.

RMVRVM paradigm was applied on two real-world applications run by a large health care insurance company, one of which is being actively used by over 0.5 million (5 Lakhs) of its customers. Also, an experimental application was developed to do study this paradigm. After applying the paradigm, the battery consumption by the application came down by up to 66%, and responsiveness of the application improved by up to more than four times.

The contributions of this paper are as follows:

- Proposes a novel paradigm that helps save battery on mobile phones and other resource-constrained devices. Additional benefits to an application like better response time are also obtained as a natural side-effect of more straightforward implementation,
- Details out the challenges that could be faced by practitioners during its implementation and proposes solutions to those challenges,
- Deals with various ways of how such applications consume REST API, including internal API and external or 3rd party API and how the proposed paradigm can handle these, and
- Provides details on experiments and two case studies that provide validation of the paradigm and act as a guide to practitioners for implementing it.

The remainder of this paper is organized as follows. Section II provides the background information on the current state of architecture of such cloud-connected applications on both the client and server-side. Section III covers the related work. Section IV introduces the paradigm and discusses the architectural changes required. Section V describes the experiments and case studies conducted in the industry to implement the paradigm. Section VI provides the conclusion and scope of work in the future.

2 BACKGROUND

This section covers the architecture of user application and its REST API, focusing on the description of application side MVVM paradigm's details.

The architecture [7] that is followed in general for creating applications running on constrained devices like mobile phones is shown in Figure 1. There can be different applications on the client-side – mobile applications, applications running on IoT devices or their base stations, and inside of robots, desktop, and progressive web apps (PWA), among others. These applications obtain the data required at the client-side by making REST API calls to the services that provide such data. Most such applications will have their counterpart REST API services available on the cloud. To provide data to client applications, the architecture of REST API generally consists of different layers and components. Upon receiving a request, the rest API layer can do many tasks like authenticating the request, decryption of the request data, and query strings. The services layer receives the request from the API layer and proceeds with executing steps to get the information asked for in the request by the client-side. These steps generally involve querying the database through the database layer to obtain information and then process

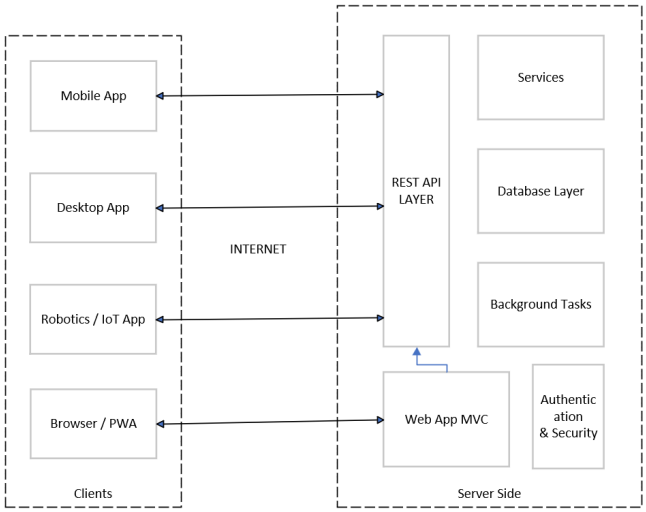

Figure 1: REST API Application Architecture

it. It could also involve invoking 3rd party API to obtain additional data and then sending it back to the client through the API layer as a REST API response to the incoming request.

On the client-side, paradigms like MVVM are used to receive, filter, and transform the data incoming in response to the REST API call. As shown in Figure 2, the MVVM like patterns are implemented by various objects in different layers of the user application. Figure 2 is a representation design of a typical mobile application for users. The application's UI consists of different pages that display relevant information to the user. Typically, each UI page is bound to its view model internally. The data binding of UI elements to the view model has several advantages. It allows an automatic update of the state of view model objects of a UI page based on the inputs provided by the user. It does not require getting data out of the state of each UI element that is affected by the user's input; the MVVM paradigm automates it. In two-way data binding, the state of a UI page is automatically updated in the event of a change in the state of its view model object. Thus, view models and their data binding to respective UI elements is a very effective software design for user-centric applications.

The view model objects are tied to the data model through converters. These model-to-view model converters translate data present in data model objects to view models and vice versa. One data model object often caters to more than one view model because the same data set could be presented differently to the user. Each form will have a separate view model, but they all will have the same underlying source of data, i.e., model. If a state is changed in data model objects because of the updates due to incoming data from API, patterns like publish-subscribe are used to trigger updates to the upper layers of UI application. The updates to the data model due to inputs from users are done by regular flow because upper layer objects have direct reference to lower layer objects. The JSON format of data is pervasive in REST API. The JSON format may need to be converted into data model objects through JSON to model converters.

It is important to note that if the incoming response has no excess data, the need for JSON to model converter could be eliminated because JSON itself is a valid object state in JavaScript which is the primary language used on the client-side for processing information. The service or API access layer prepares the correct URL for the REST API, including the query parameters and request body, and initiates the REST API call.

2.1 Problems and Issues with State-of-the-art Architecture

The state-of-the-art architecture of user applications that run on resource-constrained devices like mobile phones and are connected to the cloud via REST API is not power efficient. The primary reason is that the architecture has been applied from the conventional applications world, like desktop applications connected to the cloud. There are the following issues with the current architecture.

2.1.1 Excess data received in response to REST API calls. The API response often sends much more data than is required at the application end. This could happen for various reasons like if API is designed ahead of the application, the API is created by a different team, or there is a communication gap between the application and API team, the application is consuming an API that was already existing, API is catering to multiple clients with different UI, etc. This excess data consumes unnecessary network bandwidth and affects the battery because data transmission consumes energy. Additionally, filtering the data received consumes the battery further because the excess data needs to be weeded out before the data is put into data model objects for further usage in the application.

2.1.2 Unused data lying in data model objects. The view models often need only a fraction of data available in data objects in the application's data model. Furthermore, this difference in requirements of what data a view model needs and what is available in its data model leads to having converters. Usually, each view model has its converter and is usually created as a separate abstraction or embedded in the view model itself. The conversion of data models into view models consumes CPU cycles, slowing down the application and consuming energy. For two-way data flow, i.e., from the model to the view model and then to the view and vice versa, the energy consumption increases even further because model objects state needs to be maintained for consistency of the application.

These problems in current user application architecture result in more battery consumption and sluggish application behavior that could result in user dissatisfaction to the extent that the user may uninstall the application from the device.

3 RELATED WORK

As the work in this paper relates to reducing the energy consumption by application, information about how various hardware components of a phone stack up in typical usage of the battery was studied [1], where work by Carrol et al. confirms that CPU is the 2nd most energy-consuming hardware in a phone after GSM. Pramanik et al. [5], in their State-of-the-Art review of Smartphone battery and energy usage report that on Android devices, up to 24% of the battery is consumed by applications during computation. A battery optimization technique for health activity monitoring applications

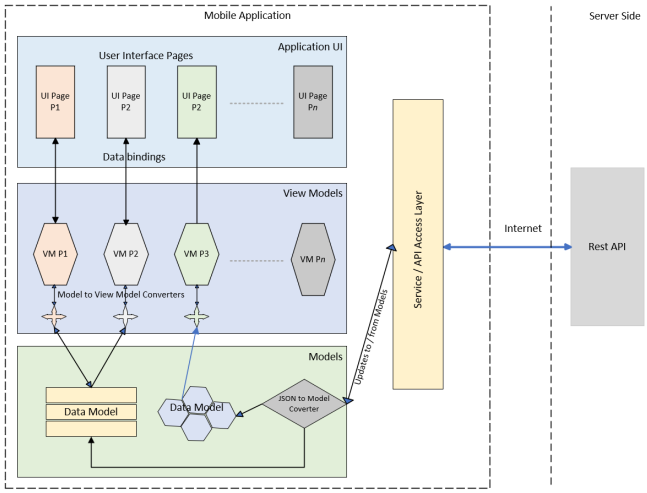

Figure 2: Client-side application design

based on smartphone sensors was done by Eastwood et al. [10]. They modified the application to detect intervals where it could go into a 'sleep' state to consume lesser power. This is equivalent to reducing the CPU usage in applications built following the traditional MVVM paradigm. Un-necessary data reception is identified as a problem by Kim [11], and a solution based on battery level sent to server-side is proposed. The analysis of power consumption by applications using 4G network like LTE is done by Gupta [12], where power consumption patterns of popular applications are studied. Application State Proxy (ASP) design is proposed by Bolla [13], which focuses on stopping the applications running in the background to conserve energy.

E. Chen et al. [14] propose offloading an Android application to a virtual machine with an IaaS (Infrastructure-as-a-Service) solution and does not tie up with mobile applications architecture. F. Xia et al. [15] propose a Hadoop-based offloading mechanism for mobile applications with the primary target of reducing the execution time of the computation task. The decision based on which task should be offloaded or not still needs to be made on the device, which could still be significant because it requires estimation technique implementation. K. Akherfi et al. [16] advocate partitioning the application at compile time to identify components that can be offloaded. This approach assumes that the offloading tasks executing components are known beforehand. C. Ragona et al. [17] deal with wearable devices offloading tasks to mobiles, considering these as mobile cloud components.

4 PROPOSAL

This section proposes the RMVRVM paradigm and discusses how it addresses the issues identified with state-of-the-art architecture in Section II. After that, the challenges that might be faced by practitioners in implementing the RMVRVM paradigm are discussed, and possible solutions are proposed.

4.1 The RMVRVM Paradigm

This paper proposes a solution to the problems mentioned above in a new paradigm RMVRVM (Remote-Model View Remote-View-Model). The paradigm derives its name from the traditional MVVM paradigm, but it is entirely different from MVVM when seen from an architectural point of view. The proposed paradigm moves both the model and view model of the user applications from the client-side to the server-side. This move has far-reaching consequences and alters the application architecture on both the client and server-side in a significant way.

The architecture that results by following the RMVRVM paradigm is shown in Figure 3. The architecture moves both the model and view model of the application user interface to the server-side, modifies the REST API appropriately to make the response objects directly usable by the client-side. This architecture puts the onus of dealing with complexity on the server-side and endeavors to make the application or client-side design as straightforward as possible. This approach eliminates excess and unused data and allows users' device applications to be more frugal, responsive, and delightful to use.

4.1.1 How to identify excess and unused data on the client-side? The excess and unused data is the one that is received by the client-side when REST API is invoked, but the client-side does not use the data received in response or must filter the data further to make it worthwhile for the end-user. In other words, any data that would make no difference to the client-side if it (the data) stops coming to the client is excess and unused data. A key goal of RMVRVM architecture is to eliminate the excess and unused data on the client-side.

The detail of the architecture is now discussed.

4.1.2 Server-side design. The REST API request contains the information about the UI page whose data is to be fetch from the server-side. The server-side receives the REST API request through the REST API layer. The data in the request include its query parameters, is decrypted if required in the REST API layer. There are two types of REST requests that the server-side needs to handle primarily. One, GET requests are the API calls from the client-side to receive data from the server-side. Two, POST requests are the API calls for sending information from the client-side to the server-side for serialization into data store like databases or runtime state of server-side objects like caches. The other types of requests are PUT, for updating the state of an existing object, and DELETE, for deleting an object.

The GET request is the most prominent because of its more frequent use in obtaining information from the server-side by the users' application. In the discussion that follows, the role of components is described primarily for handling the GET request for clarity. The actual implementation will add the necessary code to handle all four types of REST API calls.

Query Processor. Since the request is for data about a specific page on the UI side, the server-side code must identify the view-model corresponding to the UI page. Query Processor does this task. The Query Processor could have components responsible for analyzing the query, involving parsing the query parameters and constructing appropriate abstraction defined in the query processor from the parsing result. This object created after query parsing is used to identify the view model ready to receive the data from the model.

View Model. The view model created by the query processor component is now responsible for fetching data from its model. The data model objects are generally singleton, and they keep updating their state from data sources such as databases by connecting to them through the database layer interface. Every view model object obtains a reference to the data model attached. It creates its data model to view the model converter. Then, it invokes the data model object, passing it to the converter and any other query parameters necessary for obtaining the data. A critical artifact from the view model to its data model is the delegate that the data model will invoke using the publish-subscribe design pattern.

Data Model. The data model object is generally a singleton and exposes a method for view models to obtain its singular reference. It also provides a method for view models to register their delegates for the publish-subscribe mechanism. The relationship between the data model and its view model is one-to-many; that is, a view model is attached to only one data model, but a data model could be attached to one or more than one view models. Upon receiving a request for data from one of its view models, the data model object triggers a query to the data layer using this layer's API. The returned information is converted to the data model's state. Once its state is ready, the data model object notifies all view models that registered their delegate about updates to its state.

Model to View Model Converter. The primary role converters play is to transform the data from the data model to its view model. A converter is entirely owned by its view model. The data model could use the converter if the view model passes it to the data model as a delegate object. When the data model invokes the converter delegate, it transforms the data supplied into an appropriate state suitable for the view model. Alternatively, a view model could directly interact with the data model and use its converter internally to transform data it received from the data model into its state. The case studies implemented the first method of using converters in a delegate of view model to the data model and converting the state upon invocation by the data model.

Response Preparation. View model receives the data from the data model object through the publish-subscribe mechanism. It is recommended that the view model does not further manipulate data. The manipulation of data is the job of its converter, and doing so by the view model will violate the Single Responsibility principle. Next, the response preparation is done after the view model provides its state to the Response Preparation component. This component is responsible for preparing the REST API response data object and providing it to the API layer, which sends the response back after packaging it to the relevant response structure of the REST request.

The sequence diagram in Figure 4 is a simplified version of the actual sequence of events in the application mentioned under industrial case study I. The key takeaway from this diagram is that only one of the three view model objects is created depending upon what parameter has been sent in the query by the user application. The Query Processor parses the query string to find the query

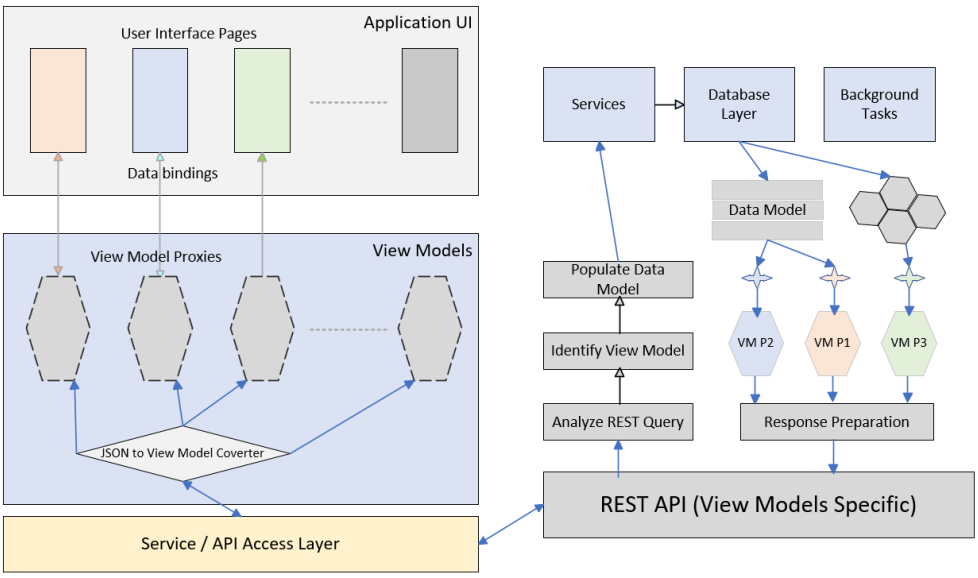

Figure 3: RMVRVM Architecture

parameter used by the UserController to create the correct view model object that will get data.

4.1.3 Client-side design. The client-side design is shown in the left half of Figure 3, which depicts the RMVRVM architecture. The Application UI layer has the user interface pages used by users to view the information and interact with the application.

View Model Proxies. Each UI page has a view model proxy. The difference between a view model proxy and a view model proper is that the proxy does not have any behavior. It is just a representation of the state of the view model created during this REST API on the server-side for its UI page. Using a simple proxy view model helps in data binding. It receives its state directly from the JSON to View Model converter. The view model proxy objects are singletons and are created on-demand, i.e., when the user interacts with its UI page for the first time. Once created, the view model proxy lives through the application's life.

JSON to View Model Converter. The client-side has a JSON to View Model converter object that transforms the incoming response in JSON to the structure of the view model proxy. The response object already has a field that identifies which view model proxy needs to be updated. The converter reads that field and serializes the JSON into the view model proxy object, a singleton. If the paradigm is followed correctly, the conversion of JSON to view model is only a notional step with minuscule overhead because the response object contains JSON, a replica of the view model state that fits perfectly when serialized to the proxy. So, no transformations or conversions are involved, and only copying is done. This copying could be further simplified using the client-side application program's language constructs, like moving the JSON object to view model proxy rather than copying. As the view model proxy is data-bound to its UI page, the UI framework automatically updates the view (UI page), thus reflecting updated information to the user.

The client-side design has become much simpler as compared to the MVVM paradigm. The key benefit of using the RMVRVM paradigm is a simple application side design. The client-side design has become more straightforward as compared to the traditional MVVM paradigm because,

- The whole layer of data models is eliminated. This is possible because the response object maps to a view model and brings a state that directly fits with the view model proxy. Thus, the data model is not needed anymore.
- There are no converters required for transforming data models to view models and vice-versa. The CPU cycles that get consumed in these transformations in MVVM are eliminated, saving battery and time
- There is no excess data received on the client-side, and therefore no filtering is required of the data received in the REST API request.

This simplification on the client-side leads to significant gains for the user. The battery consumption is reduced, the application's response time is reduced, and therefore application becomes faster and frugal and a delight to use while consuming lesser battery power on the user's device. This is the critical outcome of using this paradigm.

4.2 Server-Side Energy Consumption – Is there a net reduction in energy footprint?

Because the RMVRVM paradigm moves the complexity to the server-side, we need to answer this question from a sustainability perspective – Using the RMVRVM paradigm, is there a net reduction of energy consumption or not? The author did not conduct the study or experiment to measure the change in energy consumption on the server-side; however, the following observations can be made to answer this question.

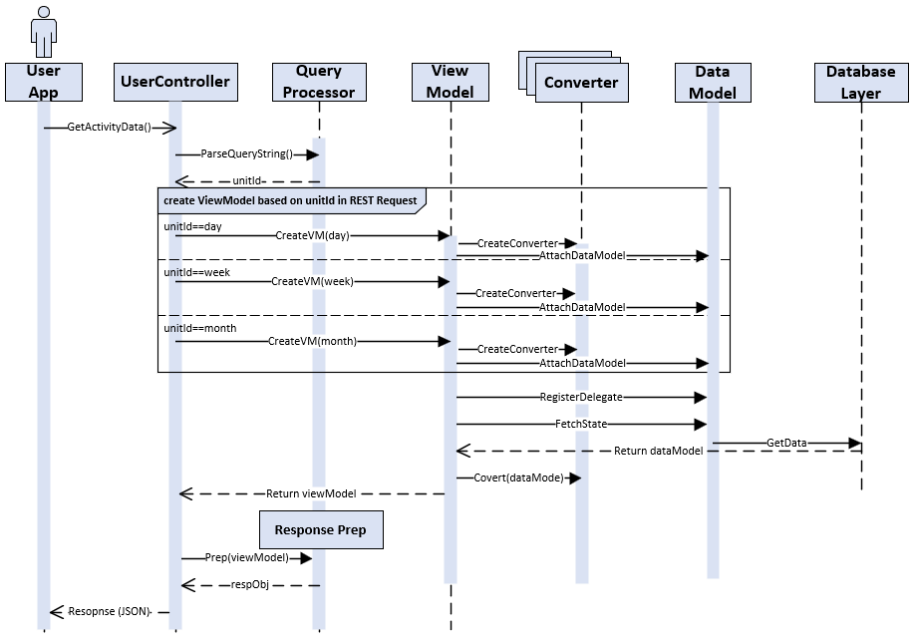

Figure 4: RMVRVM Execution on Server Side

- RMVRVM reduces the API load, i.e., the volume of data that needs to travel from server to client-side is reduced. Hence a net reduction of energy consumed over the network can be expected.
- In section 5, RMVRVM is shown to reduce the carbon footprint on the client side. The server side is usually a more controlled environment, and the server providers employ green energy practices at a data-center level to reduce the overall carbon footprint of the data center. This could reduce the degree to which more energy is required because the computations are moved to the server-side by RMVRVM.

4.3 Challenges to the adoption of RMVRVM

The proposed RMVRVM paradigm affects the application architecture in a significant way. Because the architecture moves the view model and data model of the application from client to server-side, it affects both sides of the application architecture. In this sub-section various situations, where adopting the RMVRVM pattern could be challenging, are explored and guidelines and design are provided with more elaboration.

4.3.1 Challenge 1: The REST API that the application uses is a 3rd party API not under application author's direct control. Many applications use 3rd party REST API to get information for their users. The 3rd party REST API is not under the author's direct control of the application. So, the question is – how the RMVRVM can be adopted because no change is possible on the REST API side.

Discussion & Solutions

A more common scenario is that the application uses both 3rd party and internal REST API. In this case, the solution is to move the calls to 3rd party API to the server-side and expose the same through the internal API. The user application calling 3rd party API directly is never a good idea when the application has its own API. The server-side should do the 3rd party API calling and provide the result to the application. Once this modification is done, the RMVRVM paradigm can be followed.

In a rare scenario where the application running on the user device uses only 3rd party API, the RMVRVM is unsuitable. A Proof-of-Concept (PoC) is recommended to be done in this case to create an internal API on the server-side that wraps the 3rd party API, implements the API required by the user side conforming to RMVRVM paradigm like removing unnecessary data from the 3rd party API responses, and then measure the gain in terms of battery usage, the response time of application and amount of data handled by the application. In most cases, it is expected that the PoC will help decide to adopt the RMVRVM paradigm because the application will become better for its users on all performance and efficiency parameters.

An example taken from case study II is practical to illustrate the real-world handling of this solution. The application involved showing the location of an agent to the master user. The application running on the agent's phone would send its location to the server-side, and the server will store this information. The application on the master phone will call REST API to get the location from the server-side, use Google's Location API to get the address of the location coordinates received from the API, process and transform the received data because the response of Google's Location API is too verbose and finally show to master user the address of the agent's location. Here Google's Location API is 3rd party API. When the adoption of RMVRVM was done for this scenario, the responsibility of calling Google's Location API was moved to the server-side, so was transforming the response to view model of

the UI page. Hence, the application was saved from doing all this processing, and it simply got the address in exact JSON that could directly map to its view model proxy and got displayed on the UI page through data binding.

4.3.2 Challenge 2: A mobile application has many UI pages that dynamically refresh. How is it possible to move every view model to the server-side? This is a common scenario among large applications. For example, mobile applications that handle e-commerce. For such applications, the number of pages will be significant. Individual pages might be showing a list of products that get refreshed dynamically. How can RMVRVM be applied in this case for existing applications?

Discussion & Solutions

In the case where applications have many UI pages, it is recommended that the view models be categorized in 3 lists ordered – by usage, a degree of data transformation, and excess data received using the REST API. Then the view models that top the list should be picked up and moved to the server-side. Once the work is complete, this list could be used again to pick the following top view models. In the agile project management model, the team can do this under NFR (Non-functional requirements) work. Note that the adoption of the RMVRVM paradigm does not impact the functional requirements of an application. RMVRVM is an implementation design that positively affects NFRs. Hence, the agile process can allocate fixed hours to this NFR work and gradually move at least the most impactful view models to the server-side, if not all.

4.3.3 Challenge 3: The RMVRVM moves the complexity to the server-side. How can this complexity be managed? The benefits of RMVRVM come at a cost that increases complexity on the server-side. But it is argued that server-side complexity is more manageable than application-side complexity. The changes are easier to do on the server-side because server-side codes' ownership is with the team or organization. It is easier to deploy updates to server-side code than to upgrade an application running on user's devices.

Many approaches can be adapted to handle the complexity of RMVRVM on the server-side. One solution is to use a separate REST API for each view model separated using the URL routing paths mechanism. This approach will create many abstractions, each responsible for one view model. The expected behaviors and properties among these abstractions can further be abstracted into higher levels making a hierarchy, which enables re-use. An API aggregator layer can use a façade design pattern to project a single API layer to connect to for any application using the API.

5 RMVRVM - EXPERIMENTS AND INDUSTRIAL CASE STUDIES

In this section, the experiments conducted to validate the efficacy of the RMVRVM paradigm are described. Two industrial case studies, where the RMVRVM paradigm was applied, are also discussed.

5.1 Experiments

The experiments were conducted by creating a mobile application that simulated workload on a mobile device. This application can be configured to run using RMVRVM or otherwise (i.e., MVVM). The application was built using the Xamarin framework using C# and runs on Android devices. The server-side required for the RMVRVM paradigm was implemented using ASP.NET Core 3.1 Web API application with REST API to start and stop the experiment, get task execution status, and upload the report.

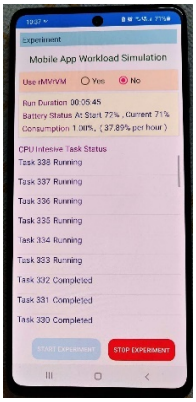

Figure 5: Application User Interface

5.1.1 Source Code. The source code for the mobile and web API application, the experiment reports, and the Android package (APK) is available on the GitHub repository [18] with public access.

The simulation of workload on CPU was performed using tan(arctan(tan(arctan. . . load function executed iteratively. This method has been used in the CPU soak test in the PDP-11 microprocessor [8]. The maximum possible iterations were set to C# / .NET framework's Int32.MaxValue [9] which is 2,147,483,647. A reverse-intensity value is an input from the application to calibrate it, and the iterations are reduced to the division of Int32.MaxValue by the reverse-intensity value. This calibration helps run loads of different intensities on the CPU to simulate different situations of applications consuming battery on the device.

5.1.2 Using the Application. After the application is launched, the reverse-intensity value defaults to 10,000. The experiment can be started by clicking on the Start Experiment button on the device. The application executes the tasks in multiple threads, one task per thread spawned every second. Each task executes the load function for iterations set as described in the previous para. This task is executed on the device if the Use RMVRVM option is set to No; otherwise, that task is executed on the server-side code running on the cloud. As the tasks are executed, the application displays the status of tasks. For ease of implementation, the application shows the status of the last 50 tasks only. When the application is running, it shows the duration of the run, battery status at start and current, and consumption of battery which is the difference of start and currently available battery percentage values (Figure 5). It also displays the rate of consumption battery per hour. To stop the application, the Stop Experiment button can be clicked. This stops the experiment and uploads the data to the cloud backend, converting it into CSV format for data analysis. The reports are stored in the cloud and the same were used for data analysis.

Table 1: Configuration of Devices Used in Experiment

Phone	Battery	CPU / RAM
OnePlus A6000	3300 mAh	Snapdragon 845 / 8 GB
Samsung Galaxy F62	7000 mAh	2.6 GHz / 6 GB
Asus 6Z	5000 mAh	2.8 GHz / 6 GB

Figure 6: Battery Consumption Experiment Results

5.1.3 Experiment Setup and Run. The experiment was run on three different Android phones. Following are the configurations of these phones.

The reverse-intensity level for OnePlus and Asus phones was set to default 10,000. For the Samsung Galaxy device, the reverse-intensity level was set to 6000 due to its larger battery. Before starting the experiment, the phone's brightness level was fixed to a low level, and Adaptive Brightness was turned off. The lock screen was also turned off, so was the screen saver. All other running applications were closed. The experiment was performed multiple times on each phone, keeping the RMVRVM setting to Yes or No alternatively.

The battery consumption in percentage is captured by the application every time the battery drops, along with the experiment's duration. Hence, the application aggregates a collection of tuples of <time duration, batter percentage change> during the experiment run. After the user stops the experiment, the application sends this collection of tuples to the cloud through REST API. The cloud backend web API application receives the collection and converts it into CSV format and saves it in Azure Blob storage for later use. These CSV report files were then downloaded, and the data analysis was performed.

5.1.4 Experiment Result, Analysis, and Discussion. The experiment results obtained from the reports were analyzed, and the resulting plots are shown in Figure 6 for each phone. The result of the experiments validates the efficacy of the RMVRVM paradigm. The consumption of the battery is substantially reduced when RMVRVM is used as compared to when it is not used. In OnePlus, using the RMVRVM paradigm, the application consumed 66% lesser battery than when the RMVRVM paradigm was not used. In Samsung, the result was similar, where using RMVRVM reduced battery consumption by 66%. Running the same experiment on Asus with a reverse-intensity set at a default value of 10,000, the application consumed 50% lesser battery when using the RMVRVM paradigm. The slope of the battery consumption graph is the rate of battery consumed per unit time. It can be observed from the graphs that the rate of consumption of the battery is linear, which is expected

because of experiment loads the CPU in a linear fashion using a loop. The rate of battery consumption in the RMVRVM paradigm is 0.11 percent per minute compared to 0.4 percent per minute in the case of the OnePlus A6000 device. The battery consumption rate for MVVM is, on average, 3.5 times that of the RMVRVM paradigm.

The RMVRVM paradigm has a penalty on battery because it must get data from the cloud about the result of the tasks executed. This is not the case for standard or MVVM because the tasks are executed locally on the device. In the experimental setup, the request to the cloud is made every second in RMVRVM to keep parity with the non- RMVRVM option, where a new task gets created every second that runs the load function. Despite this penalty, there is a reduction of 66% battery consumption in the case of RMVRVM.

5.2 Industrial Case Studies

The RMVRVM paradigm has been applied in two industrial products. These two applications of the RMVRVM paradigm are discussed here as industrial case studies.

5.2.1 Case Study I – ActivityDayz Application. The ActivityDayz application is a mobile application that is part of a business in the Health Insurance sector owned by a large industry group in India engaged in multiple businesses. The application runs on both Android and iOS devices.

The current user base of the application is about 0.5 million (5 Lakhs) active customers. The application is available only for customers that take health insurance from the business house. This application takes input from various activity monitoring devices like smart watches and fitness devices like Fitbit, GoogleFit, SamsungHealth, Apple watch that the user might use to monitor health activities like walking, jogging, exercising, etc. The application sends two types of fitness activity data used by this system – the number of steps taken and calories burnt per day by the user. Then, the backend system calculates the number of active days for a user, and health credits are given to the users. A user can then use these health credits to pay for various things like a health insurance premium.

The application that runs on users' phones has many pages showing users their health activity-related information. For this case study, we'll take an example of a view that could show activity data daily, weekly, and monthly basis based on user's input. The original implementation is as follows. The backend system had a single API for getting activity data of the user – GetActivityData(). When the phone application calls this API, it will return health activity data to the application from the start date mentioned in the API until the end date mentioned in the API request. The application receives this data into its model objects, then filters them according to the current view and presents it to the user. For example, suppose the user wanted to see the last month's activities in a weekly split. In that case, the application will take the input from the user, which is the week-wise view in this case, call the GetActivitData() API, receive data, and then filters and calculates weekly activity data. The application receives data from API per-day basis, converts it into per week basis, and then fills up the week view model object. The user is finally presented the data as the weekly view page is data-bound to the weekly view-model object that the app just updated.

Both problems discussed in section III of this paper were present in this application. One, the GetActivityData() API was returning data that far exceeded the actual requirement of the application at the instance – because the data was always in per day format. The excess data required filtering and conversion. Hence, the application was sluggish when responding to the user's request to view health activities data per day, week, or monthly basis. The second problem of unused data lying in the model objects also manifests in model objects having to maintain health activity data daily regardless of the user's requirement. Hence most of the data were unused in the model objects and must be discarded.

The application was modified on the mobile and server-side, applying the RMVRVM paradigm. A 'unit' query parameter was introduced in the GetActivityData() API with three possible values: 'd', 'w', 'm' for per-day, per-week, and per-month, respectively. The response of the API was modified to match the exact view-model object structure after discussion with the mobile application team. When the application calls this API, it will pass the 'unit' query parameter in the request URL, setting its value to 'd', 'w', or 'm'. This choice is based on the user's input. In the per-week requirement, for example, we are discussing here, the value passed to the 'unit' parameter will be 'w'. When the server-side receives this request, it has all the information to create the same view model object that the application needs to show the health activity data to the user in per-week view. Instead of sending per-day data, the server's side processes this per-day data to convert into per-week data, create per-week view model objects and fills them up with this data, serializes the object state to JSON and sends this as a response to the API call.

The implementation became much simpler on the application side – it just needs to de-serialize the JSON response received directly to view-model proxy objects present on the application side. These view model objects are created quickly due to the direct mapping of their properties with the fields in the JSON. Hence no conversion, filtering, or transformation of data is needed. The application's UI shows the health activity data weekly when the JSON is de-serialized into view model objects.

With the RMVRVM paradigm applied, neither of the problems discussed in section III are there on the application side. Therefore, the application becomes much more responsive and a delight to use by the customer. The API returns no excess data – every byte returned in the response is useful for the application immediately.

The result of this change on both mobile application and backend implementation, including the change in the GetActivityData() API's request format, was phenomenal. The per-day/week/month pages became quicker to load than earlier. Here is a quote from an email sent by the quality engineering team member who tested the product's build released after implementing this change.

"With the d parameter, the response timing is much improved (quick) as compared to the build without d parameter."

During the User Acceptance Testing (UAT) phase of the application, the in-house users of the application were asked to share the change in response time of the application in the views where the RMVRVM paradigm was applied. Figure 7 shows that after implementing the RMVRVM paradigm, the response time reduced substantially for per day, week, and month views. The primary reason for the reduction in response time is the transfer of both

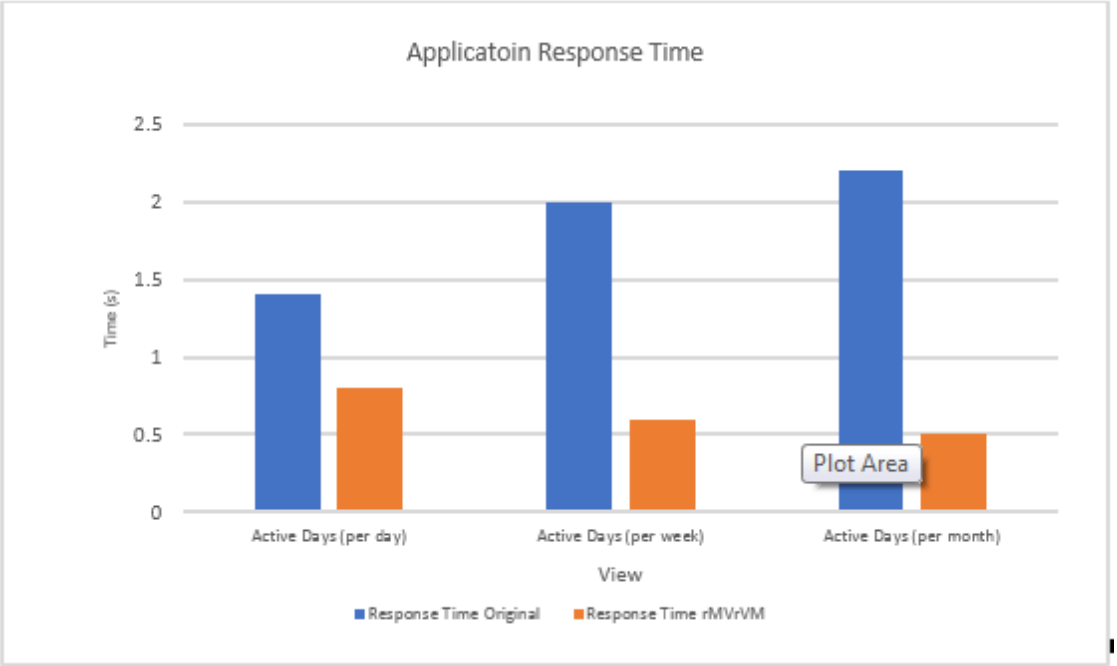

Figure 7: Response Time in ActiveDayz Application

model and view models of the pages (or views) to the server-side. The application is spared from doing any calculation and hence the response time is many orders better after RMVRVM based implementation. The number of API calls when moving from RMVRVM increased from one to three. This increase cannot be generalized and will depend on the scenario and would range from being the same as before to a higher number. However, it can be argued that the overall API load in the RMVRVM pattern will be lesser as the API will be refactored to return only required data and nothing else versus the old model where API returned redundant data.

5.2.2 Case Study II – Eagle Eye - Agent Tracking Mobile App. This case study is about the Eagle Eye application used by the same industrial house as the first case study. This application is used to track agents on the field engaged in various customer relationship activities like door-step insurance service, collection of insurance premium from home, etc. The location of field agents can be tracked using their Android phones. The application has two modes of operation – a controller and an agent. The device which will track other phones is called the controller. The application in controller mode can track its agents in different modes. The applications' agent tracking feature has two views, a) List of agents that are being tracked. Each agent's last known location address and the time are shown in this view. This is for enabling glance by the controller user. The controller user can tap on one of the agents in this list, and a detailed view opens where a map view and details of each location visited by the agent on a particular date can be viewed.

The chief complaint about this application was from controller users about switching between different agent views. The application took unacceptably long to switch between list view and agent details view. Upon investigation, it was found that the problem is with the amount of data the application needs to handle with each switch of the view. The server-side API, GetAgentLocationHistory() would return location details data for last one month. The location was sent in latitude-longitude coordinates from the server-side. The application with each switch would use Google Location API for finding the address at the latitude and longitude combination. The second problem was the first view only had to show the last known location of the agent, but the application was still dealing with 30 days of location data for one agent. When the number of agents was large (50 or more), the application was very sluggish in responding to users' actions.

RMVRVM paradigm was implemented to fix both issues – remove the sluggishness of the application and reduce the data processing required on the client-side. RMVRVM was adopted by applying the application changes on both the client/device and server-side.

The Google Location services API call was moved to the server-side. When an agent phone sends its location to the server-side, the server code will use Google API to get the location's address and store it in the database. The response of GetAgentLocationHistory() API was modified to include the descriptive address so that the application can use it. Another API was introduced exclusively for the main page view, which lists the agents and their last known locations and timestamps. A new API, GetAgentStatus(), was introduced, which returns a JSON that matches the main page's view

model exactly in response. This included a separate object for the last known location of the agent. By data binding, this JSON to the agent list on the main page view, the application's response time improved by many orders, and switching between views became quicker. The agent's details page, where the agent's location during a day is displayed on the map, was also modified. The calculation of converting a location to a map's pin data structure was moved to the server-side. Now the application would create the pin collection from the response directly and pass it to the map control on the device, which will draw these pins. Also, for the agent location history view, pagination was implemented in GetAgentLocationHistory() by providing in response data of only the last three days instead of 30 days that was the case initially.

These changes lead to immediate and substantial improvements in the applications' response time, as reported by the UAT team. The switching between the main page view and the agent's detail's view was much quicker to the delight of controller users. This was the natural outcome of the RMVRVM paradigm, making the client-side much simpler. It was also noticed that with pagination, the application's response time was better even though the data would travel through the wise more often when the pagination gets involved, for example, when the user wanted to see the last seven days of an agent's location history. This seemingly counterintuitive result is because in the original case, even though data about all 30 days was present on the device, the application had to do filtering and conversion of this data as per the current view mode's requirements. This is not the case with RMVRVM because the rVM (i.e., remote View Model) is a replica of what is needed on the client-side. Hence, despite more frequent travel on the wire, the pagination-based solution made the application more responsive because the application had to no calculations, filtering, or conversions.

6 CONCLUSION & FUTURE WORK

This paper proposes a novel RMVRVM paradigm to replace the traditional MVVM paradigm for energy-constrained devices like smartphones. RMVRVM paradigm helps write green software. The experiments conducted on three Android devices of different configurations validate the idea of RMVRVM. More importantly, the RMVRVM paradigm was adopted in two industrial applications, resulting in substantial battery usage and response time reduction. Thus, the applicability of RMVRVM in real-world situations is demonstrated.

The capability of the experimental application to calibrate the workload on the device's CPU provides means to simulate other application scenarios such as what level of workload versus data transfer will bring benefit. Such a study is planned. Also, the battery consumption by the RMVRVM paradigm due to more data travel involved needs more study to specify clear guidelines on the usage of RMVRVM pattern, which will help it get adopted by the IT industry.

REFERENCES

[1] A. Carroll, G.Heiser, "An Analysis of Power Consumption in a Smartphone", NICTA and University of New South Wales.
[2] Rest API, https://www.ibm.com/cloud/learn/rest-apis [Last accessed: 27-12-2021]
[3] MVVM Design Pattern, https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android
[4] WPF Apps with MVVM, https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern [Last accessed: 27-12-2021]
[5] Data binding in Andriod https://developer.android.com/topic/libraries/data-binding
[6] P.K.D. Pramanik, N. Sinhababu, B. Mukherjee, S. Padmanaban, *et al.* "Power Consumption Analysis, Measurement, Management, and Issues: A State-of-the-Art Review of Smartphone Battery and Energy Usage", IEEE Access Vol. 7, 2019.
[7] Architect Modern Web Applications with ASP.NET Core and Azure https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/?WT.mc_id=dotnet-35129-website [Last accessed: 27-12-2021]
[8] CPU intensive calculation examples, https://stackoverflow.com/questions/3693197/cpu-intensive-calculation-examples [Last accessed: 27-12-2021]
[9] Int32.MaxValue field https://docs.microsoft.com/en-us/dotnet/api/system.int32.maxvalue? view=net-5.0 [Last accessed: 27-12-2021]
[10] N. Alshurafa, J. Eastwood, *et al.*, "*Improving Compliance in Remote Healthcare Systems Through Smartphone Battery Optimization*", IEEE Journal of Biomedical and Health Informatics, Vol. 19, Issue 1, Jan. 2015.
[11] M.W. Kim, D.G. Yun, *et al.* "*Battery lifetime extension method using selective data reception on a smartphone*", The International Conference on Information Network, 2012.
[12] Gupta, "Analyzing mobile applications and power consumption on a smartphone over LTE network", 2011.
[13] R. Bolla, R. Khan, *et al.* "Improving Smartphones Battery Life by Reducing Energy Waste of Background Applications", 2013.
[14] E. Chen, S. Ogata, *et al.*, "Offloading Android applications to the cloud without customizing Android", IEEE Xplore, 2012.
[15] F. Xia, F. Ding, *et al.*, "Phone2Cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing", Springer, 2013.
[16] K. Akherfi, M. Grendt, "Mobile cloud computing for computation offloading: Issues and challenges", Science Direct, 2016.
[17] C. Ragona F. Grenalli, *et al.* "Energy-Efficient Computation Offloading for Wearable Devices and Smartphones in Mobile Cloud Computing", IEEE Xplore 2016
[18] https://github.com/LavneetSingh/RMVRVM