SimpleHealth – a Mobile Cloud Platform to Support Lightweight Mobile Health Applications for Low-end Cellphones

Peyman Emamian
Computer Science Department
North Dakota State University
Fargo, USA
peyman.emamian@ndsu.edu

Juan Li
Computer Science Department
North Dakota State University
Fargo, USA
j.li @ndsu.edu

Abstract— **Mobile medical applications are increasingly being used by patients and consumers. However, due to their complexity, these applications are normally only accessible to smartphone users. People using low-end cellphones cannot benefit from this new technology. The goal of this paper is to expand health service platform to lower-end cellphones, so that people in underdeveloped regions can benefit from it. In particular, we propose a scalable platform for lightweight health applications with novel and proactive client communication. Through the effective support of a multi-layered cloud platform, we assure the scalability, elasticity and reliability of the server side. With simple Short Messaging Service (SMS) channels, health workers and patients can access complex healthcare services with low-end cellphones. The multi-layered architecture provides separation of concerns and decoupling of communication and business logic. Furthermore, our proposed plug-in model can expand and customize functionalities. Extensive experimental results have demonstrated the effectiveness of the proposed platform.**

Keywords*— health application; short message service; cloud; mobile application*

I. Introduction

Cellphones are increasingly being used as a common platform for wide varieties of health and wellness-related applications. Nowadays, a cellphone user can download over 40,000 mobile applications that offer health, fitness, and medical applications [1]. These applications can be used for medical and healthcare education and recommendation, clinical practice and intervention, collecting and collating health and wellness data for analysis, telemedicine and remote healthcare. They will offer new opportunities to improve patient care and reduce healthcare costs.

Mobile health application is particularly important for rural regions around the world, especially in underdeveloped countries and areas. These areas normally do not have access to basic healthcare services and most of the bulk of healthcare delivery falls on local health workers who have limited skills and expertise. The last few years has witnessed unprecedented growth in the usage of cellphones in the underdeveloped world, thus linking millions of previously unconnected people. The ubiquity usage of cellphone in developing world provides new and innovative opportunities for healthcare efforts in in these regions. An exploratory study of 488 mobile phone users at Karnataka, South India shows that cellphone was acceptable in the rural Indian as a tool for receiving health information and supporting healthcare through mHealth interventions [3]. In another study performed in rural areas of Kenya [4], researchers concluded that use the short messaging service (SMS) is the most cost effective way of communication. Cellphones-based applications have demonstrated their potential for enhancing rural healthcare [2].

Despite of the existence of so many mobile health and telemedicine efforts such as Micromedex [5], Doctor on Demand [6], MyChart [7], most of the existing approaches are not directly applicable and sustainable for developing countries because of the following reasons: firstly, most of the existing applications rely on the increasing computational power of high-end smart phones which may not be economically viable for users in rural areas. Secondly, these systems require cellphones connected to the Internet which may not be available and/or affordable in rural regions. Thirdly, most of these existing applications are not aligned with the realities of rural settings in developing countries. On the other hand, there have also appeared some lightweight healthcare messaging applications (e.g., FrontlineSMS [8]) designed for low-end cellphones. These systems offer communications platforms which are affordable and highly available for users in underdeveloped regions. However, they normally lack complete functionalities and services required by a complex healthcare application.

To address the aforementioned limitations of existing mobile health approaches, we have designed and implemented *SimpleHealth*, a practical mobile cloud platform to support lightweight mobile health applications for low-end cellphones in rural areas. This platform enables health workers and patients to access complex healthcare services through simple Short Messaging Service (SMS) channels with low-end cellphones. SMS, also known as text messaging, uses standardized communication protocol to enable mobile phones to exchange messages no more than 140 characters long. SMS can send information in near-real time to thousands of people and is the most widely used data application in the world. *SimpleHealth* platform integrates cloud computing with the SMS mobile communication to overcome limitations related to SMS-based low-end mobile phone (e.g., system scalability and availability, device's computing power, storage, battery life, and bandwidth).

The rest of the paper is organized as follows. Section II surveys the related work. Section III describes the details of the proposed platform and its enabling technologies. Section IV presents the implementation. Evaluation of the proposed mechanisms are presented in Section V. Concluding remarks are provided in Sections VI.

II. Related Work

Many efforts have been devoted to design light-weight mobile healthcare application. For example, the ALIVE project [9, 10] has shown that simple email reminders can have a significant impact on improving diet and physical activity of individuals. Many healthcare providers have used text messaging to send reminders, recommendations, and education materials to patients. For example, Delaware Physicians Care, Inc. (DPCI), has used text messaging to remind patients with diabetes about their scheduled blood test appointments [11]. In another project, they uses text messaging to remind pregnant moms of their prenatal and postnatal appointments as well as to provide them with educational information. Another health-related text messaging tool is presented in [12] that has been designed to help educating the youth in the San Francisco area about sexual health. Users can send a simple text message to get information about what to do after unprotected sex or they can get guidelines and information about sexually transmitted infections, including HIV.

ELMR (Efficient Light-weight Mobile Records) system [13] offers a lightweight database access protocol for accessing and updating health records from remote cell phones. The proposed database access protocol for health care applications is optimized and simplified to be applied under extreme bandwidth constrained SMS service. The system has been used in healthcare delivery in AIDS care centers in Ghana and South Africa where health workers need to frequently access health databases using low end devices [13].

A sensor-based heart monitoring system was proposed in [14]. In the case of irregularity in user's heart rate, an SMS is sent to the user and/or the user's doctor or relatives. Similarly, [15] proposes a remote health monitoring system that monitors several vital signs of the patient (such as oxygen percentage in blood, heart rate, and temperature). If any of these parameters are not in the predefined range, an SMS will be sent to the user's doctor/emergency number. Text messaging has shown to be a successful health intervention enabler for smoking cessation programs [16, 17], depression treatment [18], obesity prevention, alcohol recovery [19], and asthma treatment and education [20] among others.

SMS-based m-health systems are also used to assist the work on health providers. For example, the work proposed in [21] can help health workers in rural areas of India communicate more effectively with doctors. The health workers collect the symptoms of the patients; then they use their cell phones to send the symptoms to a remote server where they are stored; The doctors can asynchronously access the server, review each patient's record, ask more questions about the patient, and finally do the diagnosis and initiate the treatment. The server then sends the prescription back to the health worker via SMS.

Mobile applications that are more sophisticated than a simple text messaging-based system normally require more complicated hardware and software components. For example, UbiFit Garden [22] is a health-related behavioral change system that uses on-body sensors and a mobile application user interface to encourage regular physical activity. The computer system has a client-server architecture. The XML requests are generated by the Android client and sent to the server via network. On the other side, when the data is received via HTTP request, the server creates a reply in XML format and sends it back to the client via HTTP response.

Although there are various light-weight mobile health applications and different kinds of implementations, a general-purpose platform/framework that can provide scalable and efficient services is still missing. The goal of this paper is to solve this problem.

III. System Design

The major goal of the proposed framework is to make the healthcare service platform scalable, easily expandable, and lightweight at the client side. This framework would enable complex healthcare functionality on low end devices that are incapable of interacting with rich application interfaces. For example, not all devices can communicate with Web applications or applications that need to be installed on the device. Moreover, not all devices have enough amount of memory or processing power to support browsers or user programs. Furthermore, a client that is producing data could be an entity other than human, for example a sensor or a Web service that is generating data in the system. Our framework utilizes cloud computing to implement complex health applications in the cloud. All of the major processing work must be done in a backend cloud so that the service can be easily updated and scaled. Then the client application can be simple and lightweight. With this approach, we can benefit from complicated health applications on low-end devices. Thanks to the ubiquity of SMS service for mobile phones, we choose SMS as the communication medium, although the system is also capable of communication using other approaches.

A. Overview

Fig. 1 shows the architecture of the proposed mobile cloud platform. As shown in the figure, we adopt a layered architecture to break the system into modules that communicate with each other while having minimum coherence with each other. This architecture results in separation of concerns and better scalability of the whole system. Communication between components of different layer is performed by sending and receiving messages; therefore, scaling up or down one component is hidden from the other components and does not affect any other module in the system. Moreover, the system supports plug-ins which assures that any additional feature can be added as a separate plug-in. The plug-in-able approach in the cloud makes it easy to add new features or functionalities to the program without having to change the core functionalities. This also makes the introduction and propagation of errors harder and

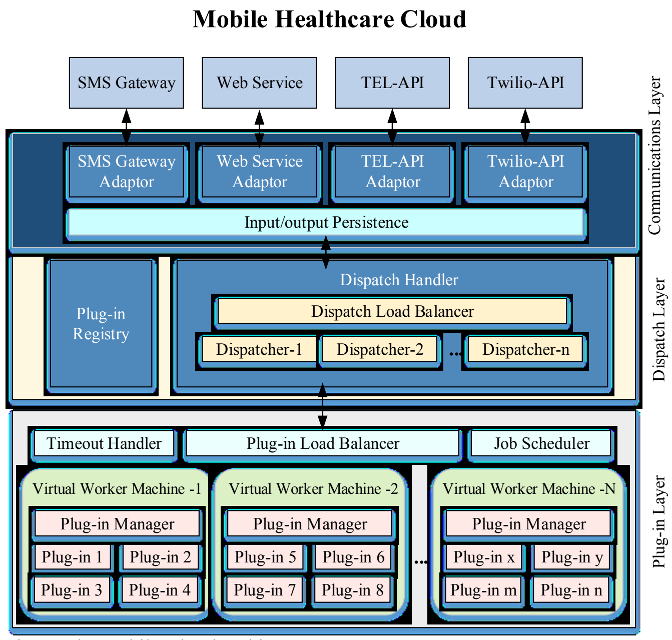

Fig. 1. The mobile Cloud architecture

less frequent throughout the system and therefore the system core functionalities would not be misused.

Although our current implementation is to empower the healthcare applications by exploiting SMS communication, our communication layer can extend the input to any means of communication, such as Twitter or Web services. As long as there is an adapter unit inside the communication layer that can convert the input to an understandable form by the system, any device with any communication format can communicate with the system. In the following section, we resent the system components in detail.

B. Communication Layer

As the top layer, communication layer is responsible for all inbound/outbound communications. It includes an SMS gateway, and can also include web service, or any gateways for other external communications. Regardless of the source, any input to the system would be converted to a unified format understandable by all of the other components/layers of the system. The input message contains the body of the message and all of the meta information. In the same respect, adapters convert responses from the lower level to the format understandable by the external communication device. Every external communication media has its own adapter unit in the communication layer to do the conversion. This approach unifies the interactions with the lower levels and provides a layer of abstraction. Therefore, the lower layers will not be affected by the changes in the communication layer.

Communication layer is also responsible for handling the sessions and cookies if available. Some external communication media have virtual sessions that provide more information about the communication. One of the important components of the communication layer is the *persistence unit*. It records any communication to/from the system and stores converted input message from the adapter along with the original communication data. The communication history that is recorded by the persistence unit is used for consistency, fault tolerance, accounting, security and statistical analysis.

C. Dispatch Layer

In a nutshell, dispatch layer is responsible for transferring input message to the appropriate plug-in. Also, it manages how plug-ins respond to the input message (e.g. timeout for generating a response.) All these are done through the dispatch handler. The dispatch layer also has a registry of plug-ins that contains the matching criteria for each plug-in. The source of the message and its content are the general criteria to decide which plug-in to handle the message. The dispatch handler consists of multiple dispatcher units. Each unit has a queue of input messages to process and pass to the plug-in layer. Dispatch load balancer manages all the dispatchers and distributes the input messages to them, and if necessary, it creates new instances of dispatchers to handle more messages or it discards the idle instances. The dispatch load balancer scales up or down the number of dispatchers in use based on the system load so the dispatch handler can respond to the input messages in the queue as fast as possible and can prevent possible queue overflows or delays in responses.

D. Plug-in Layer

A plug-in represents a specific functionality in the system (e.g., the “drug information plug-in” provides information about a drug). The plug-in layer makes sure that each plug-in responds to the input within the time limit. In addition, this layer is responsible for assuring that errors are handled properly and in case of a physical or logical failure, the plug-in action is passed to another instance of that particular plug-in.

Main processing of plug-ins is done inside a worker machine. A worker machine could be a virtual or physical machine that runs multiple instances of one or more plug-ins. A plug-in manager inside a worker machine manages the number of instances and communicates with the load balancer to instantiate the plug-ins that are needed in the system or too kill idle plug-in instances. The plug-in load balancer makes sure that there are enough instances of each plug-in available to respond to the requests from the other layers. It also routes the plug-in actions to the appropriate worker machine. The plug-in layer might contain other components to provide additional functionalities.

The job scheduler provides chronological functionalities to the system. Recurrent or scheduled jobs are handled by this unit. For example, drug intake reminders are scheduled by a plug-in request. The timeout handler unit is responsible for handling any timeout event generated by the plug-ins and it triggers the necessary actions when timeouts occur. It works closely with the load balancer such that it helps to identify the plug-ins that are faulty or not responsive. In that case, the load balancer resends the plug-in actions to another plug-in. If a component in the other layers needs to communicate with the plug-in layer, it must use the load balancer, so that the consistency of the communication between the layers is maintained and the load balancer can provide the best performance for the system.

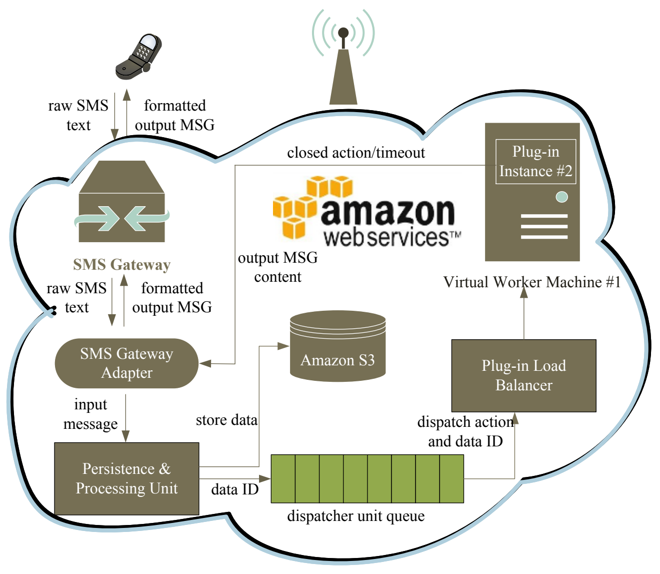

Fig. 2. Job flow in the Amazon AWS cloud

IV. System Implmenation

A prototype of *SimpleHealth* mobile service is implemented and deployed on the Amazon Web Services. The implemented prototype contains the base framework for storing messages and interacting with inputs and also includes one module that provides information about drugs. The features of the system can easily be expanded by adding new modules that interact differently with the user using the same framework.

Amazon Web Services platform is chosen as the cloud platform to provide utilities like queuing and scaling options. As for the implementation framework, we used Java Spring to implement the components of the system thanks to its ease of use, clear documentation and rich libraries integrated with the framework. In addition, using Spring makes dependency injection easier by opting in for Inversion of Control (IoC) pattern and auto-wiring reusable sub-components into each of the components.

Each component of the system that interacts with the Internet or other components of the system through HTTP communication is using the MVC (Model-View-Controller) architectural pattern [23]. In the context of Amazon Web Services, we are dividing the application into two sub-systems. The first sub-system is a web layer that interacts with inputs of the system and receives messages from outside. The second sub-system is the worker layer that processes the inputs and takes appropriate measures and actions such as sending a response or updating data in the system. The connection between these two sub-systems is Amazon's Simple Queue Service (SQS) that ensures reliable communication between these two components.

Fig.2 illustrates the information flow of the system. When the communication layer receives a SMS message from a SMS gateway, it passes it to the SMS adapter. The adapter converts the raw input to input message understandable by the system. Then the persistence unit stores the input message in the database and passes it to the dispatch layer. The plug-in handler adds the input message to a queue in one of the dispatcher units and based on the plug-in registry criteria, the dispatcher unit decides what plug-in(s) must handle this input message. The dispatcher unit creates a dispatch action and passes it to the plug-in load balancer in the plug-in layer. The plug-in load balancer passes the dispatch action to one of the plug-in instances inside a worker machine (or instantiates one if none exists). The plug-in processes a dispatch action. The processing may include looking up data, storing data, checking for a threshold, deciding on the next action, triggering an event, etc. Afterwards, the plug-in closes the action (before timeout) and return the action to the dispatch layer. The response handler in the dispatch layer receives a closed (or timed out) dispatch action and checks if the actions is correctly completed. If necessary, the response handler passes appropriate messages to the communication layer to be sent out to the device. Finally, the adapter creates the appropriate message for the gateway or web service to be sent.

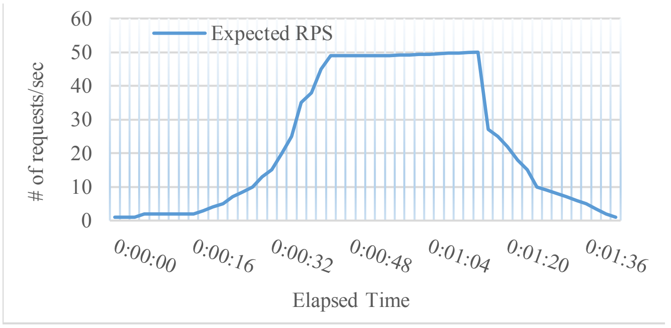

Fig. 3. Load Test - Predefine Requests for Load Testing

V. Evluation

We have performed extensive experiments to evaluate the performance of the proposed system.

In order to evaluate the scalability-related performance of the implemented prototype we apply load testing techniques to generate loads similar to the real environment to observe the behavior of the system under different situations. In particular, we use JMeter [24] to generate random targeted traffic comparable to actual traffic for similar applications. The input data is generated randomly from a set of actual requests that are answerable by the application. We used our drug lookup plugin to answer questions about drugs. Although requests were sent to the system randomly, all of the questions are valid and have a valid answer in our drug information database. We simulate request spikes for the application at certain periods of time. We also repeat the exact test based on the number of requests at each point of time in the test. JMeter and related plugins provide the environment to create an exact number of Requests Per Second (RPS) for our tests. We can create a model of desired load at each time period of the test. As a result, we are able to ensure the consistency and repeatability in our test, and exactly measure the behavior of the system at an exact load. Moreover, we can repeat the test in the same exact condition.

Fig. 3 shows the scheduled number of requests at each point of time (Requests Per Second or RPS) used for load testing. This model is generated using the JMeter plugin to shape and generate random requests. It uses a data set of actual drug names to test the system in random order. We simulate an environment that mimic the traffic of a real text-based messaging systems. An

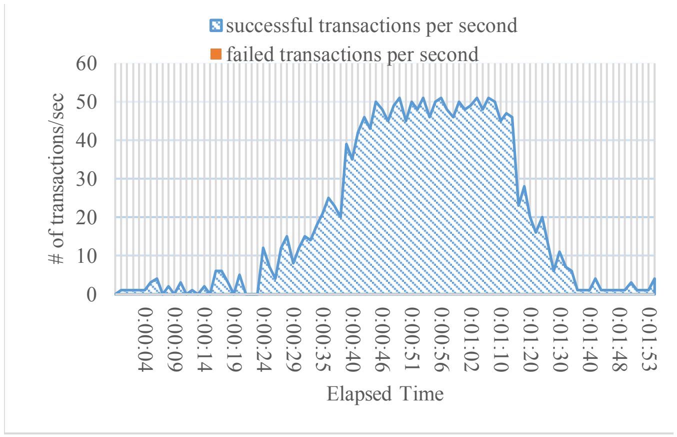

Fig. 4. Load Test – Actual transactions

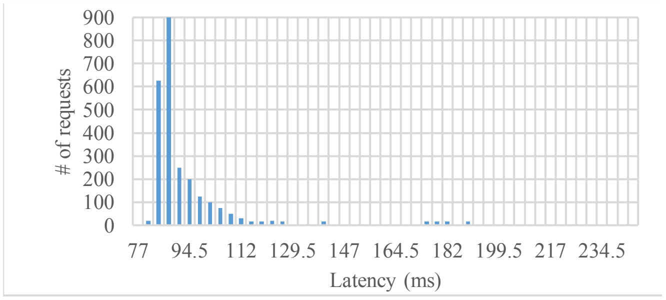

Fig. 5. Load Test - Response Latency

outside factor such as outbreaks of infectious diseases triggers a sudden growth of inputs in the system. It is usually in a short time period but the throughput spike is in orders of magnitude and could potentially put the system in the denial of service (DoS) state. As shown in the figure, we increase the input rapidly and then keep the input load on the stress level for some time. Both of these phenomena have an impact on how the application handles the load. We expect: first, the application should be able to cope with the increasing traffic and the increasing rate of the traffic increase. Second, the application should also be able to maintain the stress level for longer period of time as a sign of handling and reusing resources used in the previous requests/responses. Finally, the system should recover when the load is decreased.

Fig. 4 shows the input traffic that was actually used in the testing based on the real-time measurements. The figure shows both successful and failed transactions per second. We can see that none of the requests have failed and the application was able to successfully handle all of the requests.

It is important that our application responds to the requests under a certain time threshold regardless of the traffic throughput or the number of running instances, thus guaranteeing a uniform and consistent user experience for the customers of the application. For this project, ideally the majority of the response times should be under 200 milliseconds to provide a smooth and fast user experience. Given the nature of the communication for text messaging, response time up to a few seconds is acceptable.

Fig.5 presents the results of our load test in the response latency. The figure shows the number of requests that have a specific latency (in milliseconds). Ideally, we want lower latency for each request and having the majority of the requests (higher number on Y axis) to have a small latency (closer to the left on X axis). Although there are a few requests with higher latency, but the majority of the requests fall under 120 milliseconds. The result of our measured test shows that 98.10% of the tested requests (total of 2372 requests) had a latency less than or equal to 120 milliseconds. Furthermore, 99.74% of the requests were responded under 200 milliseconds. The request load used for this test in Fig. 5 includes all of the request and response times used for load testing of the application. As can be seen from the figure that when the application was under stress of high traffic during a short period of time, it was still able to maintain the ideal response time.

Given the platform that we are using for handling the incoming traffic, we handle each request in a separate thread, hence increasing traffic triggers more threads in the application to respond to the requests. We are using multiple layers of components in our architecture. This gives us the flexibility to scale any of those components individually. The gateway layer has its own load balancer and it can scale up/out as much as needed. This accommodates for the incoming SMS traffic without worrying about the processing time of plugins or any other concerns on the other layers. On the other hand, the plugin layer uses a load balancer as well as queues. It can scale based on the number of items that are waiting in the queue to be processed and independent of the gateway layer. For example, we use just two instances respond to the gateway layer requests because its process is easy and it is reliant on the network speed. On the other hand, we need more processing power on the plugin layer. Naturally, we may have ten instances to process the items in the queue and to answer to the users. Using different layers of scalability, we prevent the errors rom one layer to be extended to another layer. For example, if a plugin uses a third-party service to do certain processing, in case of failure or slowness on that service, we only need to add more instances to the plugins layer without affecting the gateway layer. Usually the size of the queue will trigger these actions. As a result, users will not notice the delay in the response time and the system provides a consistent user experience.

During our experiments, the scaling policy that we put in place increased the number of instances and added two more servers to handle the peak load of the inputs. It also terminated the instances and reduced to only one server when the load returned to minimum. Our scaling policy is based on parameters of the system running the instances. We used CPU utilization and response delay of instances provided by AWS instances to actively determine whether new instances are needed or we should terminate idle instances. An alternative approach would be to measure the input traffic of the instances. Since SMS messages are relatively the same size, an increase in the input traffic would correlate to the number of inputs, hence the requirement for instances.

Although the number of servers used in our proof of concept application is relatively low, it is comparable to real world SMS applications. Moreover, the combination of the scaling policy and the load balancers in each layer theoretically would have the same consistent behavior if the number of the nodes were higher. The Amazon Web Services components such as load balancers and SQS (Simple Queue Service) guarantee the same behavior.

Elasticity shows the elastic behavior of the system. When the traffic is more than the amount that could be handled by one instance, the application should scale out and initiate new instances to properly handle the traffic. On the other hand, when the traffic is less than the processing power of the system, it should scale in to preserve resources and cut the cost. Therefore, only sufficient processing power is used at any input level while the efficiency and responsiveness of the application is guaranteed to be at the desired threshold as shown in Fig. 5.

We are using Amazon's internal signals such as CPU usage and response delays to decide whether we should scale out and create more instances or we should scale in and shutdown the unused instances. During low traffic, specifically after a peak traffic, we shut down the unused or lightly-used instances to save resources. The scaling policy in this situation is based on the amount of time an instance is idle, or alternatively the amount of traffic that the instance receives in a certain period of time. For example, if an instance has received less than 10 requests per minute in the last 5 minutes, it is a candidate for shutdown. After an instance-shutdown there is a grace period that scale in policy is put to hold (e.g. 5 minutes.) The grace period is necessary because it can prevent shutting down too many instances all together. This would be a regular problem for equally load balanced instances, since all of the instances get roughly the same number of requests at any time.

VI. Conclustions

This paper presents a cloud-based platform to enable lightweight mobile health application focusing on light communication and scalable server side processing. The proposed approach is in particular advantageous for low-end devices with basic capabilities such as text messaging. However, the proposed framework is generic and expandable to support other technologies. The cloud-based framework offers great scalability and flexibility in that the system can efficiently grow in terms of the number of users and features of the system. Moreover, the cloud computing platform enhances availability and fault tolerance which are vital for a robust mobile health application. Furthermore, the proposed multi-layered architecture makes the design modular and less error-prone. The separation of concerns results in enhanced system scalability.

References

[1] Krebs, Paul, and Dustin T. Duncan. "Health app use among US mobile phone owners: a national survey." JMIR mHealth and uHealth 3.4 (2015): e101.

[2] Kumar, Arvind, Amey Purandare, Jay Chen, Arthur Meacham, and Lakshminarayanan Subramanian. "ELMR: lightweight mobile health records." In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 1035-1038. ACM, 2009.

[3] DeSouza, Sherwin I., M. R. Rashmi, Agalya P. Vasanthi, Suchitha Maria Joseph, and Rashmi Rodrigues. "Mobile phones: The next step towards healthcare delivery in rural India?." PloS one 9, no. 8 (2014): e104895.

[4] Eriksson, Evanjeline. "A case study about cell phone use by people in rural Kenya." (2008).

[5] https://www.micromedexsolutions.com/

[6] www.doctorondemand.com/

[7] https://mychart.centracare.com/

[8] http://www.frontlinesms.com/

[9] Block, Gladys, Barbara Sternfeld, Clifford Block, Torin Block, Jean Norris, Donald Hopkins, Charles Quesenberry, Gail Husson, and Heather Clancy. "Development of Alive!(A Lifestyle Intervention Via Email), and its effect on health-related quality of life, presenteeism, and other behavioral outcomes: randomized controlled trial." Journal of medical Internet research 10, no. 4 (2008): e43.

[10] Sternfeld, Barbara, Clifford Block, Charles P. Quesenberry, Torin J. Block, Gail Husson, Jean C. Norris, Melissa Nelson, and Gladys Block. "Improving diet and physical activity with ALIVE: a worksite randomized trial." American journal of preventive medicine 36, no. 6 (2009): 475-483.

[11] Text messaging-a new way for delaware physicians care to help its members. http://www.businesswire.com, June 2008.

[12] Erin Allday. Health department answers questions via text messages. http://www.sfgate.com/health, April 2006.

[13] Kumar, Arvind, Amey Purandare, Jay Chen, Arthur Meacham, and Lakshminarayanan Subramanian. "ELMR: lightweight mobile health records." In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 1035-1038. ACM, 2009.

[14] Altini, Marco, Julien Penders, and Herman Roebbers. "An Android-based body area network gateway for mobile health applications." In Wireless Health 2010, pp. 188-189. ACM, 2010.

[15] Wyne, Mudasser F., Vamsi K. Vitla, Praneethkar R. Raougari, and Abdul G. Syed. "Remote patient monitoring using GSM and GPS technologies." Journal of computing sciences in colleges 24, no. 4 (2009): 189-195.

[16] Bramley, Dale, Tania Riddell, Robyn Whittaker, Tim Corbett, R-B. Lin, Mary Wills, Mark Jones, and Anthony Rodgers. "Smoking cessation using mobile phone text messaging is as effective in Maori as non-Maori." (2005).

[17] Obermayer, Jami L., William T. Riley, Ofer Asif, and Jersino Jean-Mary. "College smoking-cessation using cell phone text messaging." Journal of American College Health 53, no. 2 (2004): 71-78.

[18] Joyce, David, and Stephan Weibelzahl. "Text-messaging as a means to lowering barriers to help seeking in students with depression." In Proceedings of IADIS International Conference e-Society, Dublin, Ireland, pp. 211-214. 2006.

[19] Krishna, Santosh, Suzanne Austin Boren, and E. Andrew Balas. "Healthcare via cell phones: a systematic review." Telemedicine and e-Health 15, no. 3 (2009): 231-240.

[20] Yun, Tae-Jung, Hee Young Jeong, Tanisha D. Hill, Burt Lesnick, Randall Brown, Gregory D. Abowd, and Rosa I. Arriaga. "Using SMS to provide continuous assessment and improve health outcomes for children with asthma." In Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium, pp. 621-630. ACM, 2012.

[21] Mukherjee, Chinmoy, Komal Gupta, Rajarathnam Nallusamy, and Sumit Kalra. "A system to provide primary healthcare services to rural India more efficiently and transparently." In Proceedings of the 1st International Conference on Wireless Technologies for Humanitarian Relief, pp. 379-384. ACM, 2011.

[22] Consolvo, Sunny, David W. McDonald, Tammy Toscos, Mike Y. Chen, Jon Froehlich, Beverly Harrison, Predrag Klasnja et al. "Activity sensing in the wild: a field trial of ubifit garden." In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1797-1806. ACM, 2008.

[23] Model-view-controller (mvc) software design pattern definition. https://en.wikipedia.org/ wiki/Model-view-controller, April 2016.

[24] Halili, Emily H. Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites. Packt Publishing Ltd, 2008.