Design and Development of Responder: a Notification-Acknowledgment Android OS Phone Application

James H. Greenwell[1]
KBRwyle, Greenbelt, Maryland, 20771, United States of America, james.h.greenwell@nasa.gov

This paper discusses the design and development of the Android phone application Responder and how it assists Flight Operation Team (FOT) members in handling alerts from a Short Message Service (SMS) text message and email based notification system. The FOT needed a simplified and more powerful tool for receiving, storing, and acknowledging notifications sent from the ground systems automated notification software called Attention!®. The previous system was to rely on a text message application that comes on a phone. This way was slow, inefficient, and did a poor job of keeping FOT members informed on the state of a situation when multiple notifications are being sent out by Attention!®.

Responder is designed to provide easier viewing and acknowledging of multiple messages. It gives the user a quick look at all recently received messages in an organized and clean manner that prevents notifications from being lost during a large burst of messages. It makes it easier to find and review old messages received and Responder was designed to give the user more flexibility in setting up how and what alerts will notify them when they are received by the phone. Since the implementation of Responder the FOT members are more informed and are able to focus more on problem solving the reason a notification was received then trying to keep up with sending cumbersome acknowledgements over text message or email. Responder is only able to receive and send SMS text messages but by using email to text services provided by the cellular carrier it is able to send and receive emails.

I. Introduction

The Global Precipitation Measurement (GPM) Mission has a Mission Operations Center (MOC) that is lights out, staffed only during working hours Monday through Friday. One mechanism that allows for this is the commercial software Attention!® that works with the Goddard Mission Services Evolution Center (GMSEC) architecture to notify FOT members 24 hours a day, seven days a week about the health and safety of the various parts of both the ground system and satellite. When a problem is detected a message is sent to the GMSEC bus and picked up by Attention!®. The software is then able to notify a FOT member of the abnormal spacecraft and or ground system situation via a Short Message Service (SMS) text message and email. The messages must be acknowledged by the FOT member in a specified amount of time or the message will be escalated and sent to more FOT members. To acknowledge a message a SMS text or email must be sent back to the notification software with the corresponding message code.

The notification software is only in the MOC and provides no application to assist a FOT member with receiving and responding to notifications. This means that a FOT member must receive and reply to messages using a text messaging application on their phone. This setup works fine for single alerts, very simple use cases, and simple notification system designs but it quickly becomes inefficient and cumbersome to the user in a real world setting. It is not uncommon to receive a large burst of five to fifteen messages at a time or within quick succession and having to acknowledge each message individually is challenging using a standard text messaging application. Looking for previously received notifications by scrolling through a large number of old text messages presents a similar problem. An application was needed to provide easier responding, simplified viewing, quicker access to information, and flexible to different use cases and possible future architecture changes.

[1] Systems Engineer, GPM Flight Operations Team.

II. Design and Development Plan

The project started as a request from GPM FOT members to have an easier way to respond to notifications from Attention. It is a piece of software that was designed and developed as a side project when time would allow. Because of this, there was no formal process to planning the design and development. The informal process, looking backwards, could be considered to have consisted of requirements gathering, feasibility research, coding, testing, and refining. The software development paradigm that was used would best fall under the agile methodology. This paradigm wasn't chosen as much as it was a product of the resources available. The Agile Software Design methodology is based on the idea of an iterative and adaptive model that values a working product and a satisfied customer over complete and well documented design.

A. Requirements Gathering

The requirements gathering portion of the design was informal and ongoing throughout the project. Interviews and meetings with other members of the FOT was the main source for requirements gathering. Working with the small team meant it was easy to find concurrence on the core functions for the application. The overall requirement was to create an application that could allow the user to follow and participate in the lifecycle of a notification message. This processes also made it very clear that there were a list of nice to haves or possible future features that could be added to the application at a later date. This finding during the requirements gathering is what lead to the decision to take an iterative approach to the design. First build a beta version that fulfills the most basic of requirements and use this as a proof of concept. Next, build up a fully functioning version that meets all core requirements and then as time and other resources allow add nice to have features.

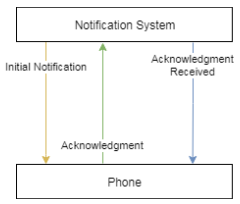


Fig. 1 Notification Message Lifecycle

B. Feasibility Research

The feasibility of the project was in question given the resources and lack of knowledge base in this area. The list of requirements had to be evaluated against what could reasonably be done in the amount of time allotted given the inexperience. A lot of research was done into the programming language, Integrated Development Environment (IDE), phone operating system, and database that could be used. Eventually the decision was made to use a phone running the latest version of the Android operating system. The application would be coded in Java using the Android Studio IDE. These choices were made because they provided what looked like the easiest way to achieve the ultimate goal of Responder.

Android Studio had numerous examples and tutorials that were extremely helpful in learning the basics of coding an application for the Android OS. The IDE provided a very easy interface for creating applications without needing to be a full time Java and Android App developer. This meant that the framework for coding the application was mostly done and customizing different classes and objects was the majority of the work. It also provided an easy to use workspace for creating the user interface. The templates and basic configurations that came with Android Studio meant design and implementation was simplified. There was no need to write xml pages from scratch. The immense knowledge base and examples that could be found online also aided in the decisions for Responder.

A major benefit to choosing Android over other options is the ability to load applications not in an official App store. This is an ideal situation as there is no intention of getting the application into an official App store such as the Apple App Store or Google Play Store. The process to load a personal application onto an Android phone is very simple and provided a quick way to implement Responder as each version is completed and ready for operations.

C. Coding

Planning for the coding portion was not very organized nor documented and this is the part that fell into the adaptive part of the agile methodology. The overall plan was to use examples and tutorials from the feasibility research portion and then adapt them to fit what Responder was supposed to do. Coding was done by a single person which simplified the integration processes.

The coding process involved breaking down the program into different use cases or stories and then building custom classes and code to accomplish each of these cases. To prevent duplicating code the use cases were broken down into functional group and objects. Braking these down allowed for the building of custom classes that could work across use cases. Due to inexperience, there was a lot of trial and error to accomplish this task and a lot of testing, refining, and recoding that was done to get a final product. As is the case with almost any project, getting small increments to work alone was possible but integrating these together is where most of the time was spent.

D. Testing

As with the rest of the development sections, there was no formal test plan laid out at the beginning of the project. The general idea was to test as it was built. This meant testing each separate use case or iteration to confirm that their objects had been met and any requirements that were tied to them were also met. This build, test, build cycle allowed for a lot of flexibility in the coding process as well as finding small errors before they were integrated into the larger project. This saved a lot of debugging and error hunting time.

System testing of Responder was also done once enough of the components were functional for this kind of testing. Once again by not waiting until Responder was complete before testing a lot of headache was saved. It was easier to debug and make changes to core functions before they became too cumbersome to change. This allowed for a more seamless integration into the MOC architecture once Responder was ready for final testing before implementation.

E. Refining

For this paper, refining is referring to the cyclical way these prior steps were applied. It is the process of integrating different core functions and use cases, testing as one builds, and then repeating until a final product is ready. So this isn't so much a separate process from the others but the idea that all previous steps are being repeated and refined as testing and integration is happening.

III. Notification Architectures

When planning the notification scheme the team has to answer several questions about how the notification system will be designed. Should everyone on the team be alerted to every message or should there be a primary point of contact? How will these messages be sent out? Should the system rely on email, SMS messages, some other internet based communication, or a combination of both? The next set of questions deal with how to implement this design. One of the major things to consider is where the majority of the configurations will be done. How much configuration will be done by the notification software inside the MOC? This is what will be called server side configuration. And how much should be done on the phone side or what could be thought of as the client side.

A. Server Side Configuration

GPM is currently setup to have two phones that receive messages from the notification system. There is a primary phone and a backup phone and these phones are carried at all times by the current primary and backup on-call FOT members. The reasoning behind this is should some situation arise where the primary on-call person is unable to receive messages the backup on-call will eventually receive them. Once the backup on-call receives the message they will now have the responsibility of alerting the primary on-call and to handle any problems until the primary on-call team member is able to be reached.

GPM's basic notification architecture was designed prior to having Responder help manage messages from the notification system. Without Responder, the phone did not have the ability to filter messages or change the phones notification settings, based on different message states and

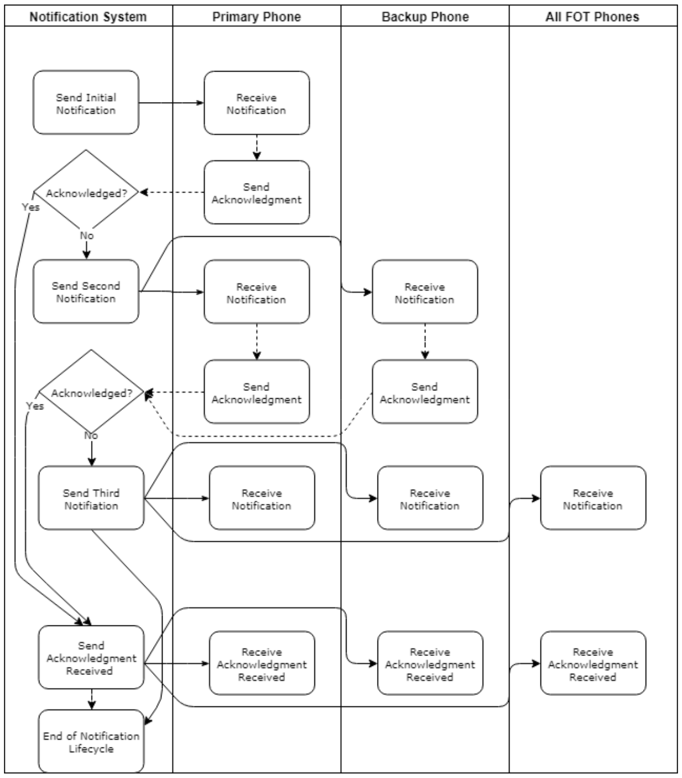


Fig. 2 Notification Escalation Example

conditions. And so configuring for the notification architecture could only be done on the notification system side, the server side, as can be seen in Fig. 2.

This configuration leads to a complex set of rules that have to be put in place on the server side of the architecture. These rules must decide who gets what notification and when based on the type of message as well as what stage in the escalation tree is the message is located. Setting up these rules takes time and because this system is inside the MOC architecture it is strictly controlled by a configuration management process. Making changes to these rules to implement a different notification architecture becomes a major effort. The entire automation system must be thoroughly tested before changes can be approved and implemented. This can take a lot of time and resources for what may be something as simple as changing the on-call phone number or where someone on the escalation tree is notified.

There is another major flaw in the design as shown in Fig 2. The primary phone is the only phone that will receive all messages and thus other team members may not get a full picture when a message falls through. It is very possible that the backup on-call or other FOT members will only see a small subset of burst of alert messages and will have to devote time into investigating the problem.

B. Client Side Configuration

Responder now gives designers a different option for where some configuration can occur. With its ability to keep track of a message's place in the escalation tree as well as differentiate between notifications and acknowledgement received messages, Responder can remove some of the complexity from the notification server in the MOC. The simplest approach can be seen in Fig. 3 where the notification system sends every notification and acknowledgment received message to every phone that has Responder. Once this simple design has been implemented there will be little need to make modifications to it in the future. The users get to decide on when and how the phone will alert them of these messages.

For example, if the FOT wanted transfer some of the configuration of their current scheme to rely more on Responders ability they could use the simplified server configuration in Fig. 3 and then configure Responder as shown in Table 1. This would give the same end result as the current scheme but would eliminate some of the faults discussed before. Now when the backup on-call gets alerted by Responder they will know where that message is on the escalation tree and once they open Responder will be able to see all messages and their current state.

Table 1 Example Responder Configuration

	Notifications		
Client	Alerts	Ack Received	Notify After
Primary	On	On	1
Backup	On	Off	2
Everyone Else	On	Off	3

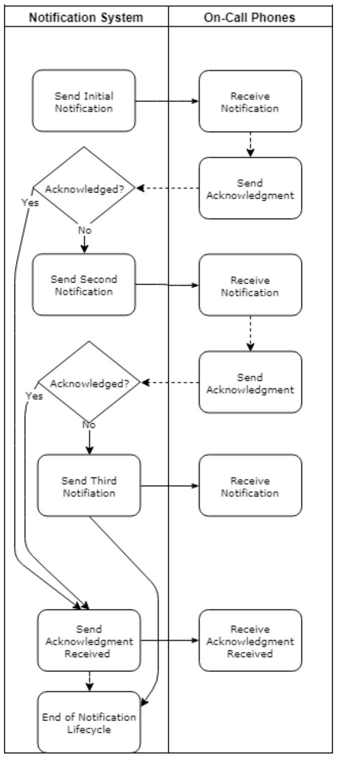


Fig. 3 Simplified Alert Escalation Diagram

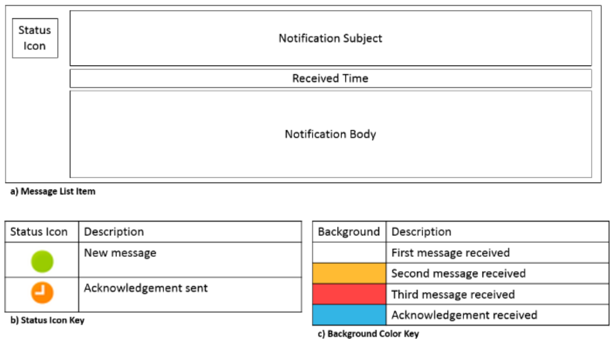
There are a number of different configurations that could give several different results and all are changeable on the fly. This is very useful if the primary on-call and the backup on-call need to switch for a few hours or days. There will be no need to physically change phones or find some other FOT member to take the phone. This kind of flexibility provides many positive side effects for the entire team. The main drawback would be if there is a miscommunication and created a situation where no team members' phone is configured as prime.

IV. User Interface Design

Responder's main purpose is to provide a simple and effective means of interacting with notification messages. With that in mind, the user interface was designed to focus on that simplicity. It was desired to minimize the number of interactions in order to respond to a message as well as maximize the amount of information in a user friendly way. To do this the application is broken down into two main screens; the message list screen and the detailed message screen. A third screen was added and that is the settings or preference screen. These preferences add a lot of flexibility to software to meet different notification designs without having to change the code and recompile the application.

A. Message List Screen

The Message List screen is the main view, the first one that is seen when the application is opened. It gives the user a quick overview of the most recent messages received as well as their current state. This view shows a scrollable list of objects called message list items. These items are generated from the current database of messages and will update as changes are made. The idea of the design was to give the user as much of a big picture of the state of both the information the notifications was delivering as well as the state of the notification itself. Figure 4 below shows the barebones designed of a message list item.



Status Icon	Description
🟢	New message
🕘	Acknowledgement sent

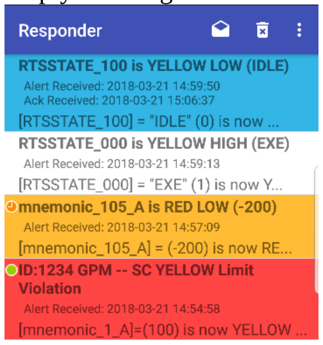
Background	Description
	First message received
🟧	Second message received
🟥	Third message received
🟦	Acknowledgement received

Fig. 4 Message List Item Design

As can be seen in Fig. 4 there are multiple text boxes that display the subject of the message, the received time of the most recent message, and if possible main details from body of the notification. Due to the variations in the type of messages and the lack of a message design standard, the algorithm that looks for important information is not always able to find key information and so as much of the body will be shown in the body text box as the screen will allow.

This message list item also has multiple ways to display information about the status of the notification by visual cues. Located on the left side of each list item view is a status icon for that particular message. This helps to show if the message is new/unread or an acknowledgement has been sent. The last bit of information is the color the list item. As seen in Fig. 4c a different color background represents the number of times that a notification with that identification number has been received or if the message has been successfully acknowledged. This is important to the FOT because for each unanswered notification that same notification will be sent out following an escalation tree. So now the FOT can know where on the escalation tree this particular notification is at.

Figure 5 shows an example of the full view when the message list screen is full of items. From this screen multiple actions can be taken on the list item messages. Clicking on a single message will launch the second view, the detailed message view. A long press allows the user to select multiple messages and perform one of a few actions to all of the selected messages. This list is also responsive and so it will update as changes occur and will build as new messages come in. It is possible to see all messages by simply scrolling down the screen with a finger swipe.



RTSSTATE_100 is YELLOW LOW (IDLE)
Alert Received: 2018-03-21 14:59:50
Ack Received: 2018-03-21 15:06:37
[RTSSTATE_100] = "IDLE" (0) is now ...

RTSSTATE_000 is YELLOW HIGH (EXE)
Alert Received: 2018-03-21 14:59:13
[RTSSTATE_000] = "EXE" (1) is now Y...

mnemonic_105_A is RED LOW (-200)
Alert Received: 2018-03-21 14:57:09
[mnemonic_105_A] = (-200) is now RE...

ID:1234 GPM -- SC YELLOW Limit Violation
Alert Received: 2018-03-21 14:54:58
[mnemonic_1_A]=(100) is now YELLOW ...

Fig. 5 Message List Screen Example

B. Detail Message Screen

This is the view that allows the user to see everything that is known about a particular message. The full subject and body of the notification can be seen as well as the sender of the message. The number of times this message has been received as well as the corresponding receive time of each message is added to the bottom of the message body before it is displayed. Figure 6 shows the layout of this screen, including the buttons to take action on the message.

So not only does this view give the user more detail about the content and state of the message it allows them to acknowledge, delete, mark as unread, or archive the message. These actions are completed by pressing the corresponding button next to the subject text box. The toolbar that contains these buttons is adaptive and responds to the size of the phone screen. Any buttons that can't be displayed are put into a dropdown menu that can be seen by pressing the overflow menu button, the one that looks like three dots stacked on top of each other.

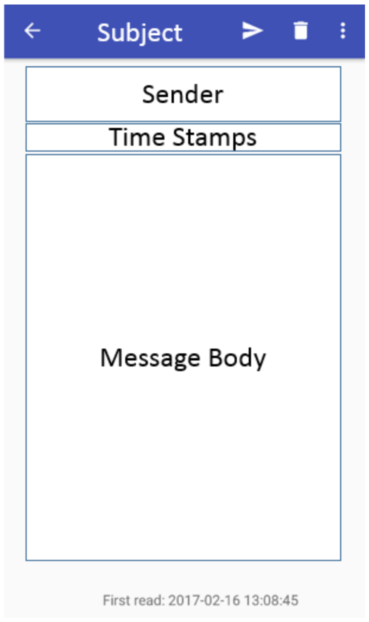


Fig. 6 Detailed Message Layout

C. Settings Screen

The preference or settings screen is where the initial setup and customization occurs. Some of the settings allow for several different notification architectures. It also allows for personalization so that notifications and sounds can be set to different team members preferences. Currently there are no personal accounts that make it simple to change these based on what account is logs into Responder. When a phone is being shared the individuals must manually make any changes. The screen is broken into three sections: General, Notifications, and Information.

The Information section gives details about the current version and cell phone carrier. This is to help with version control and baselining the app across multiple phones. The cell carrier is very important because the way the SMS and email conversions happen is done differently by different carriers. This was discovered after initial testing as the initial testing was done on a different phone carrier than the operational phone. Once these differences were discovered the functions for sending and receiving SMS messages from email had to be updated to work on multiple carriers. Currently Responder has only been tested on these two carriers.

The General section contains options and settings for filtering SMS messages, logging messages, and controlling an auto delete function. The way Responder knows which SMS messages are for it is by doing a simple compare of the sender to the accepted emails and phone numbers set in this section. It is important to note that this is just a way to filter messages so Responder won't attempt to ingest SMS messages that are from other sources. It is not meant as a security feature to authenticate the messages are real or from who they say they are.

The auto delete function was not in the original design and came about due to a limitation of GPM's implementation of Attention!®. The identification numbers used to both track and acknowledge notifications are a limited set and get reused often for different notifications. Responder uses this number to group messages together and works on the assumption that any notification with the same identification number is the same notification as any message it has in its database. So to prevent any possibility of receiving two messages with the same number an auto delete function was built that can be turned on to clear out the database of any message that are over a specified age.

For the first iteration of Responder the Notification section was used to simply choose the notification sound for all messages it received. Since that time it has grown to provide settings that allow the user to decide when and how Responder alerts them to new messages and acknowledgments. The user can choose different notification settings for received alerts and acknowledgment messages. Lastly, they can choose on which step in the escalation tree the phone will notify them. The idea behind these additional notification settings was to allow more flexibility for different notification architectures.

V. Conclusion

Responder has come a long way since the idea was first discussed. Over its iterations it has gone from an application that sent a simple reply text message to more comprehensive and complete application. With the ability to manually configure notification settings and filter criteria it is now a heavily relied on software that the FOT does not want to lose.

Going forward there are a number of different directions the development could go. Ideas about ways to improve the current application and add new features seems to be endless. One of the main focuses, when time will allow, is going to be on making Responder even more adaptable and configurable. Adding users and groups that could be preconfigured for different scenarios. This would make switching on-call phones easier. A user could have different sounds loaded or maybe even display settings such as text size.

The other main focus is going to be giving the user access to as much related information they would need about a message without needing to go outside of Responder or to be one click away. For example, GPM's notifications usually include error codes or command sequence codes. These codes can refer to any number of things and it unlikely that any team member has all them memorized. Having a feature that scans incoming messages looking for these codes and then provides a link to information about that code would be incredibly useful. This could also work with simple tracking of mnemonics and stats about how many times it has had different warning states and when they happened. Having a database with all this information that is searchable would provide a lot of assistance to the user.

Responder is currently in its third iteration with no immediate plans to begin on this new work. As with this entire design and development process, going forward relies on finding spare time to work on it. To this point Responder has been built by a single person working when the schedule allows for it. This makes it difficult to summarize some aspects of the process such as timeline, cost, and time spent working. The effort overall was at minimal cost and time spent on it as it was done as a side project, never allowing the development to get in the way of the developers main duties. The first two iterations were done over a couple of months and changes for the third iteration were done over a couple of weeks. Talking to the FOT has shown that this has all been worth it. They all agree that Responder makes their jobs easier and has been very useful.