THE DESIGN AND IMPLEMENTATION OF MOBILE DELUGE ON ANDROID PLATFORM FOR WIRELESS SENSOR NETWORK REPROGRAMMING

A Thesis

Submitted to the Faculty

of

Purdue University

by

MD Omor Faruk

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2017

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF COMMITTEE APPROVAL

Dr. Yao Liang, Chair

Department of Computer and Information Science

Dr. Mihran Tuceryan

Department of Computer and Information Science

Dr. Snehasis Mukhopadhyay

Department of Computer and Information Science

Approved by:

Dr. Shiaofen Fang

Head of the Graduate Program

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Faruk, MD Omor M.S., Purdue University, December 2017. The Design and Implementation of Mobile Deluge on Android Platform for Wireless Sensor Network Reprogramming. Major Professor: Yao Liang.

Wireless Sensor Networks (WSN) is being used in various applications including environmental monitoring, site inspection and military. WSN is a distributed network of sensor devices that can be used to monitor temperature, humidity, light and other important metrics. The software that runs on the sensor devices define how the device should operate. In real world WSN deployment, device software update is required to maintain optimal operation. In this thesis, we propose a novel idea of updating the software of the sensor nodes using a mobile device running on Android Operating System. Our implementation builds upon Mobile Deluge with few enhancement which is a method of re-programming WSN with laptop computer. We have evaluated our application performance by lab experiments and in real world deployments of WSN and found the application stable and battery efficient.

1. INTRODUCTION

Wireless Sensor Networks (WSN) are used extensively in various applications including environmental monitoring, health care, industrial and residential application and so forth. As like any other network infrastructure one important aspect of WSN is regular maintenance that includes replacing depleting batteries, repairing damaged enclosures and changing the application running on the WSN motes. WSN application includes monitoring temperature, pressure, light, humidity and others. It is very common that a new variable needed to be added in the monitoring application or an update is needed for the running application. But re-programming motes to change the running application physically one by one is very time consuming and in some cases, impossible since motes are sometimes deployed in harsh environment which are not readily accessible. Hence the idea of remote programming comes into picture where motes can be reprogrammed over the air without physically accessing them.

Remote programming can be done is several ways one possible way is to disseminate the new application in the network of motes. But the problem of this approach is it is not applicable for heterogeneous WSN motes. The term heterogeneous here implies that different manufacturer of nodes which uses different architecture of hardware and reprogramming mechanism. Reprogramming in this method is not efficient since such a protocol was not designed to work with LPL (Low Power Listening) mode of the motes. Also, in this method the wireless channel becomes very noisy since dissemination traffic and all other traffic that may be transmitting between the motes uses the same channel.

The idea of Mobile Deluge(MB) addresses all the problems of the previously stated method. MB is designed in such a way that can work with heterogeneous motes, works efficiently by overcoming the problem with LPL and bypassing noisy channel for reprogramming. MB works with a gateway node which can store multiple image format of different types. MB then can selectively connect to different nodes as commanded by the user and commands the other node to switch their channel to a pre-defined channel and to stop LPL for efficient re-programming. Now, this gateway mote can be connected to a PC or Laptop for proper operation. But most of the time carrying a laptop with a mote connected to it is cumbersome because mote deployment can be in remote areas. In addition to that, for efficient and good result gateway mote must be within the wireless range of the target motes that needs to be programmed.

Hence a more convenient way would be using a device that would be powerful yet small in size and weight. Handheld mobile devices can be used in such cases. Along with the advent of technology these small devices possess significant computation power to do complex tasks. Mobile devices that run Android or IOS are being used in variety of interesting applications other than making and receiving calls, that includes but not limited to health care equipment monitoring, smart home solutions, portable hotspot and others. Most of the applications developed uses TCP/IP or Bluetooth technologies that are readily available in the devices.

Therefore, we present a novel idea of re-programming nodes in wireless sensor networks using a smart phone running Android Operating System(OS). Since any Android devices currently does not natively provide 802.15.4 chip we have used external TelosB mote running TinyOS as our base station to talk to the 802.15.4 network. In our implementation, the android phone works as a gateway to send and receive commands from sensor networks. In the process, we have developed a re-programming application for wireless sensor nodes that runs on Android operating system. The application program can also be modified to support other applications like 802.15.4 sniffer or wireless sensor network sink for data collection.MD uses Deluge to do the re-programming but Deluge protocol is not encrypted and data can be seen by 802.15.4 traffic sniffers.

2. BACKGROUND

2.1 802.15.4 protocol

802.15.4 is defined by 802.15 working group of IEEE. The standard was defined in 2003 [1] 802.15.4 is defined for Low Rate Wireless Personal Area Networks (LR-WPANs). In the definition, lower physical(PHY) and datalink layer(MAC) are standardized and upper layers are open to implementation for specific application. WPANs operates on low power and data rate with maximum of 250Kbits/s. Unlike other wireless networks like WiFi or cellular it does not require infrastructure to operate. Various centralized and decentralized protocol exists for 802.15.4 [2]. The protocol uses CSMA/CA for collision avoidance and supports secure communications. In centralized communication algorithms, every device receives a guaranteed time slots while in decentralized algorithm nodes synchronize their transmission by distributed algorithms. Several applications of 802.15.4 are remote control, wireless sensor networks, industrial control and home automation.

2.2 TinyOS

TinyOS is designed for low power embedded devices. It is used in Wireless Sensor Networks (WSN), Personal Area Networks (PAN), Industrial and even space applications. The OS was originally developed at UC Berkley as part of the DARPA NEST program. TinyOS was released as open source software and soon gained popularity among large community of developers. TinyOS is written in NesC, a dialect of C programming language. NesC supports the component architecture of TinyOS. NesC being a dialect of C is closer to hardware and thereby optimizes code for devices with very small memory. TinyOS uses component architecture for its software. Components are connected between them through interfaces. Also, TinyOS has only one call stack. Hence, any operations that requires more than hundreds of microseconds are asynchronous i.e. non-blocking and function returns as an event to the caller. TinyOS also provides a concept called Task. Task is basically a long running operation that are posted to scheduler and are executed in a first come first serve basis only if there are enough resources available. Once the operation is done caller will receive an event. TinyOS also supports threading with TOSThreads library. Moreover, TinyOS has large community of developers, academic researchers and industry users who supports TinyOS by adding bug fixes as well as new features

2.3 NesC

NesC as discussed earlier is a dialect of C and was developed at UC Berklay to support TinyOS program architecture and execution model. NesC has some distinctive features that made it appealing to work with low memory devices. NesC gives the opportunity to structure the code in a different way than compilation. It is done by first constructing some components that does a specific task then one or more component is constructed based on other tasks. Then, all these components can interact with each other via an interface. In NesC term this interaction is called wiring. During the compilation process wired components generate whole program. Component can behave in different ways in terms of interface. A component can provide or use an interface. The component that provide the interface has the responsibility to implement the functionality for the other component who is using it. Interfaces are bi-directional in NesC, i.e. interfaces define a set of functionalities (Commands) to be implemented by the provider of the interface as well as set of events that must be implemented by the user of the interface. In a typical application, Commands travel downwards i.e. closer to hardware whereas the events move upward (from hardware to application layer). Finally, to optimize the program NesC uses whole program compilation and static binding of components via their interfaces.

2.4 Android

Android Inc. was initially formed in 2003 to develop smart mobile devices. Android Inc. was initially struggling to make an impact in the market. Then in 2005, Google acquired Android Inc. Google first developed a mobile platform powered by Linux Kernel [3, 4]. First commercially available mobile phone running on Android OS was released on October 22, 2008 [5]. Starting from 2008 google is improving the OS, fixing bugs and releasing regular updates. But most of the time updates are not pushed to devices immediately because of the delay of different carrier or vendor who has modified android OS. Also, manufacturers always concentrate on releasing new devices and thus updates for old devices remain a lower priority. This problem causes a security vulnerability for millions of devices running older Android OS. But important security updates are always released by Google as soon they are detected.

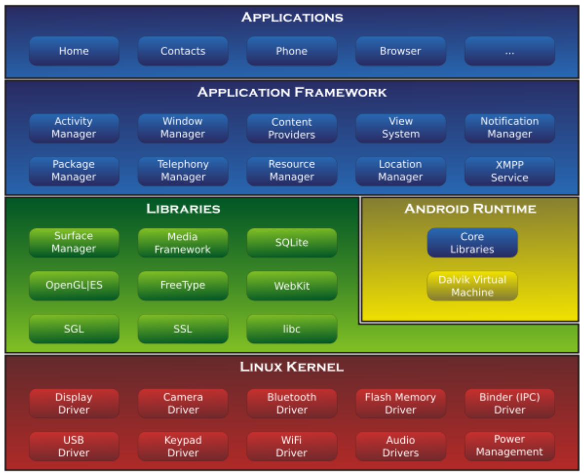


Fig. 2.1.: Android Architecture

Android runs Linux kernel under the hood but many features and modification are done in favor of devices that have lower memory and power. Since phones are typically run on batteries, it is important that resources are managed efficiently to make the best of battery power. Android included Binder, ashmem, pmem, wakelocks, loggers and out of memory handling [6] in the kernel of Linux. These modifications have made android unique. As Google also claims android is not Linux since it does not include the GNU C library instead it uses Bionic and lacks other components that are typically found in Linux kernels.

Android architecture can be represented in a layered architecture. In the architecture, there are core OS libraries, most of which are written in C programming language. These core libraries are part of Android Runtime. Android Runtime also includes Dalvik Virtual Machine (DVM) in which apps are run. DVM is different from Java Virtual Machine (JVM). DVM uses trace-based-just-in-time (JIT) compilation to run Dalvik dex code. DVM was a feature of Android up until version 5.0 which later replaced by Android Run Time (ART). Besides core libraries there are also other OEM libraries. This additional layer of library is there to support device driver that are closed source. Next layer upward is the application framework. This is where most development work is done by the application developers. This framework implements various software design patterns. The framework manages all the object creation and complex interaction between objects under the hood.

Finally, on top of application framework various application are built. Some of these applications are android specific like Phone, Contacts and Browser application and developed by google. User applications also reside in this layer. Android has a distinct feature that forces application developers to run lengthy commands on other thread than the user interface thread also known as UI thread. If UI thread is kept busy for certain amount of time, then ANR (Application Not Responding) dialog is shown to the user and the application terminates. Hence, android provides various framework for managing the interaction between UI thread and user created thread. One such frame work is called HaMeR. HaMeR stands for Hander, Message and Runnable framework. The main UI thread receives a call back from other thread once required work is done in the worker thread. UI thread then can only update the User Interface (UI) components that the user interacts with.

2.5 Android Services

Android activities are designed to be short lived and not suitable for running long running processes. Long running applications can be network access, database query or storage access. Android activity gets destroyed for various user interaction. For example, if user changes the orientation of the phone the Activity Manager destroys and re-creates the activity. In the process activitys onDestroy and onCreate method is called. A service is independent of the activity life cycle and can also run in a completely different process. Hence the service can run long running operation without any user interaction. This feature is crucial for blocking network or other I/O operation because if the state of the activity a data transmission or IO operation can be interrupted in the middle without any recovery mechanism. In addition to that, blocking operation on main thread can also cause ANR (Application not responding) error. Service when run in different process runs independently of the application process. Therefore, Android OS tries to keep the service alive even when the system is low on resources. However, Android provides a way to restore the application state after configuration change. Android system provides minimal restoration facility via a Bundle data structure. Bundle data structure can retain key value pairs which can be stored on the call back of onSaveInstanceState method. OnSaveInstance method gets called just before the activity gets destroyed giving the developer the opportunity to retain any state needed. The states that are being saved then can be restored on the onCreate or onRestoreInstanceState call back method. But this method of storing state does not work for heavy objects like a bitmap image. Thus, it is not appropriate to retain state in this way for long running applications that contains large amount of data. Also, if it is required to store large amount of data in configuration change user will get slower response in the application due to large amount of pre-processing before re-starting the activity. To overcome the issue, Android do provide another mechanism to save states in case of configuration change via Fragment class. Fragment Manger provides a mechanism to store large objects and restore them on configuration change. Yet one more way to handle the configuration change is to manually handle the state information and update them. This method requires writing more code logic because automatic resource management provided by Android is not available in manual mode. Even though the manual method provides a better performance to the user by not restarting the activity but manually configuring all the resources for different device orientation and configuration makes this process rather complicated. Moreover, when the back button is pressed in Android the activity gets destroyed but this behavior is not suitable for some operation like downloading a file from the Internet. To complete the whole download, the operation needs to be running even if the activity is shutdown. Generally, each application is designed by one or more activities. Users frequently gets back and forth between activities. Hence, if a computation remains dependent on an activity the whole application will not scale. Navigating between activities within same application may pause other inactive activities which can also be problematic for long running computation. MVP (Model View Presenter) pattern can be used to alleviate the issues discussed above.

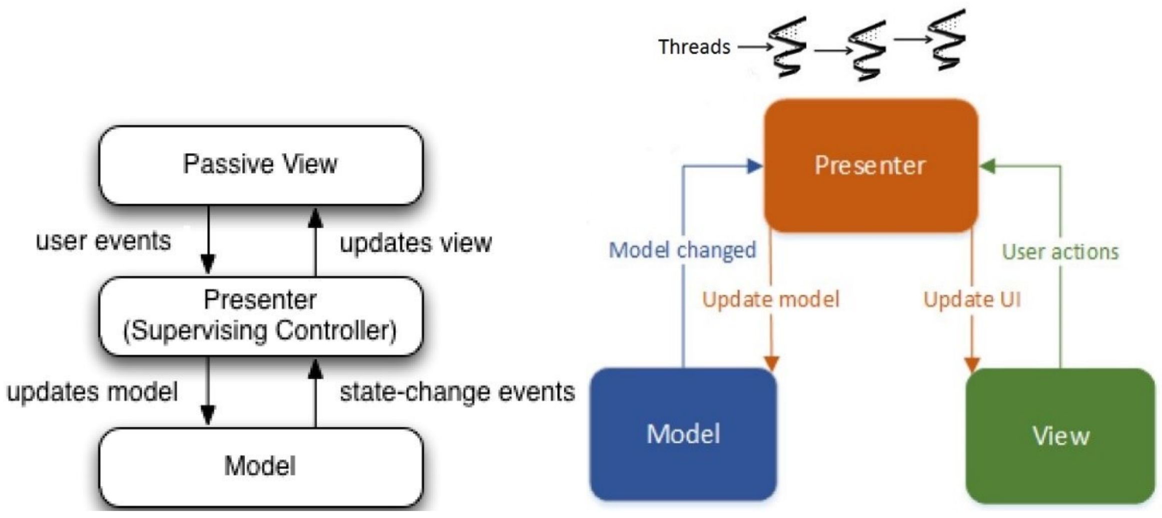


Fig. 2.2.: MVP Pattern

MVP also known as MVC (Model View Controller) pattern [7] uses a presenter/controller layer to update view that is presented to the user. The model component of the pattern stores system state that can be variables or content provider. View layer deals with user I/O. It takes the responsibility to taking various form of input from users like touch, click or scroll. View is also responsible to show the output or result of a user action by updating the user interface or playing a sound. Presenter layer contains all the business logic ie. It processes user input by necessary computation and logic and updates the model if needed. Presenter then request the view layer to update the view accordingly. Presenter layer abstracts computation burden from the view layer by running them on separate thread of control other than the user interface thread. Thus for long running operations that reside on presentation layer does not affect the activity life cycle events. Android framework provides fragment class discussed earlier to retain key state information in case of configuration changes during runtime. This process also has limitations for complex computation requirements. In short, activities do have significant limitations when used for long duration and blocking operations. Hence we need an alternate way to run long running background operations that runs concurrently with the user interface thread to give users a good experience. Android Service solves the problem with the activities and long running operations. Services can be running even If the activity gets killed. Services can be used to play music files when the user is away from the music activity doing another task or it can download a file over Internet when user is surfing the web. All the work done by a service is transparent to the user and usually done with a different thread of control. However, developers must explicitly create new thread or process to run away from the UI thread because by default service runs on UI thread. Service component can be used to implement remote procedure call to other process but that is not a requirement. Service can reside to the same process with the main thread. Android source code uses service in a lot of applications. For example, Gmail app uses synchronize messages with the cloud server database with a service. Service is also used in contacts, SMS/MMS, Calendar and other applications. All works done in a service does not require any user interaction. But services do interact with users in some limited ways like status bar notification. All the services implemented is connected to an Android Activity. Activity uses Binder IPC framework to talk to a service. As described in earlier section, Binder mechanism in android is implemented in the Kernel, middle-ware and the API layer.

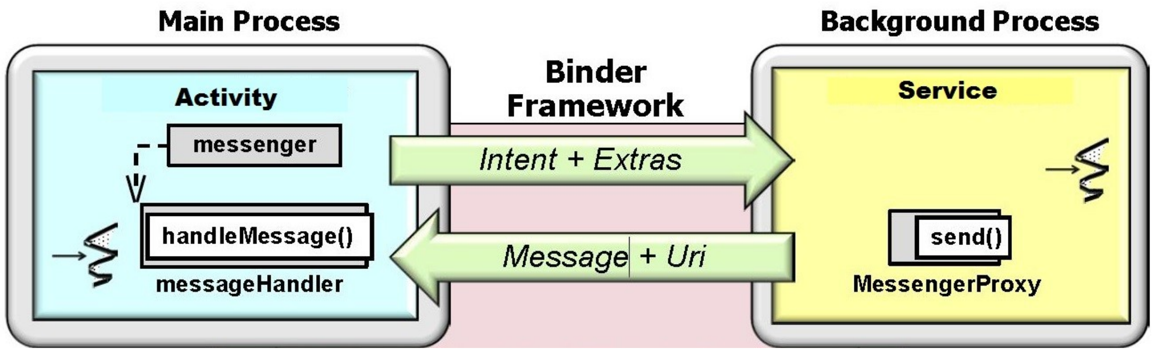


Fig. 2.3.: Service Transactions via Binder Framework

As described in figure 2.3 a service can run in different process and communicate via the binder framework of android. Main process where the UI thread resides, first starts the service via Intent along with some extra information to initialize the service and giving the service means to communicate back to UI thread. Services started in this example is done by intent. But android also allows broadcast receivers to start a service. Services itself can also start another thread if needed. Android supports three kind of service implementation. They are:

1. Started service
2. Bounded Service
3. Scheduled Service

Started service is started by the user by calling the startService() command. The life cycle of a started service is independent of its creating component. That implies that, started service once started will keep running even if the component that created it gets destroyed. Started service can run indefinitely on background if not stopped by either the service itself by calling the stopSelf() method or another component by calling the stopService() method with a proper intent along with any extra information to identify the service. In case of starting or stopping any service explicit intent is used to identify the service due to security issues. Without an explicit intent, any malicious service can be started without user consent and can exploit sensitive information on the device. The service implementation receives the intent sent by the started component in the onStartCommand() call back method. The intent received on the onStartCommand() method can also be used to instruct the service what to do. For example, the intent may contain a URL for a specific web resource that user want to download by the service. In this case, when the service receives the URL via intent it extracts the URL from the intent then downloads the file in a background thread and finally, sends back downloaded file path back to the user. After all is done, the service can stop itself by calling the stopSelf() command.

As we can see from figure 2.4, started service always started with the startService() method invocation. Started service is initialized by the virtual constructor as a result onCreate() method of service is called. Java factory method pattern is used to invoke the onCreate method. Necessary initializations can be done in the onCreate method like starting a new thread on which the service will operate. Then call back is made to the onStartCommand(). This call back can carry intent with additional information necessary to run the service. For example, the intent can carry the URL for an image to download or a Handler object to send back message. After the onStartCommand call has been made the service enters into running state. Started service typically runs for a short amount of time and not recommended for long running operations. For example, if a SMS message is received started service displays it to the screen. Started service however does not return any result to the caller who started the service. Hence to communicate back to the client of the started service a Broadcast receiver or a Messenger is used. Typically, the started service shuts itself down after work is done or the service needs to be shut by a component by calling the

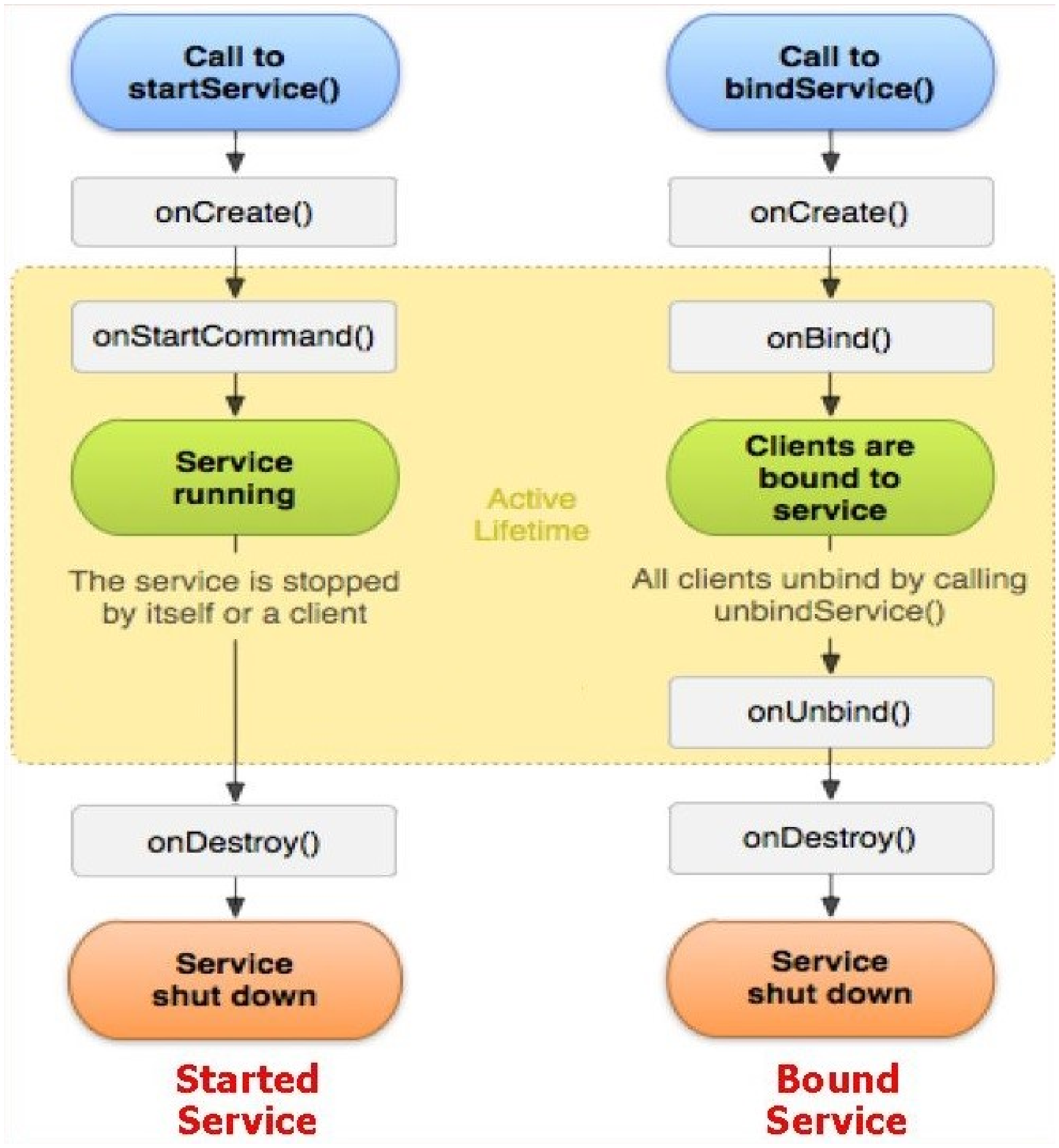


Fig. 2.4.: Started and Bounded Service

stopService method. Bound service on the other hand, is started using the bindService method. Bound service implements the client-server model between a component and a service. Initial connection is setup by handshake between client and a service. After established, the connection is used to send and receive messages by IPC (Inter Process

Communication) channels. A bound service is more desirable for a long running operation that needs to be run as a background process. Once a connection channel is established remote methods can be invoked as if it is local. Method invocation can be two way synchronous or asynchronous. Remote method invocation is supported by the Android binder framework. Remote method invocation in Android refers to calling methods in a different process. Android Binder IPC does all the work under the hood for dispatching the method call to another remote process. Running a service in another process provides flexibility and security. In particular, a process has its own address space. It also provides data isolation between two application running in different process. Processes can be configured not to accept request from foreign process thereby increasing the security of the application. Finally, a service if run in a separate process keeps running even if the application process gets destroyed or malfunctioned.

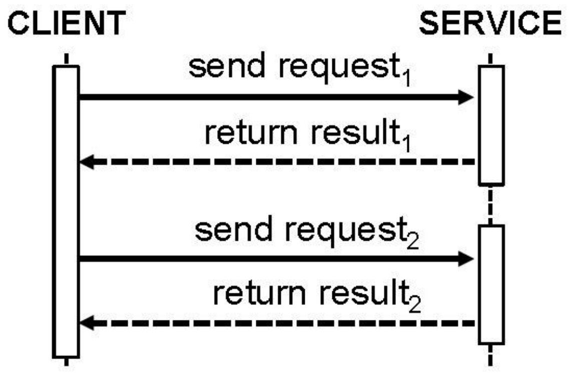


Fig. 2.5.: Bound service interaction between client and server

Android has many system services that require the binder to communicate with applications. Binder is used extensively in the process. Binder framework is used for RPC calls in a structured way.

Binder implementation resides in kernel written in C programming language. Binder driver does the work to enable inter process communication. Shared memory (Ashmem) provides high performance process to process communication. Each

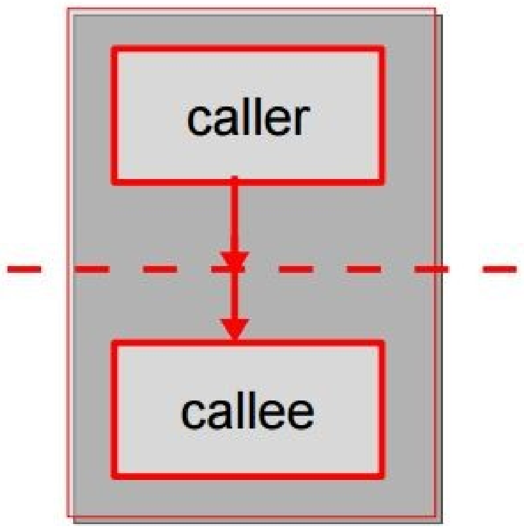


Fig. 2.6.: Method Invocation in same process

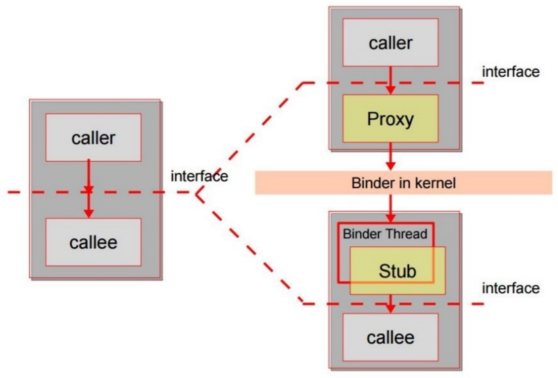


Fig. 2.7.: Method Invocation in different process

process possesses a separate thread pool to address the requests. Android provides an easy to use mechanism to use the IPC mechanism with the help of abstraction. Android Intent is the highest level of abstraction of underlying Binder access mechanism.

As described in figure 2.8 AIDL (Android Interface Definition Language) can also be used to access the binder framework. When using an intent, the whole process of binder RPC calls are abstracted from user and seems like a simple local method

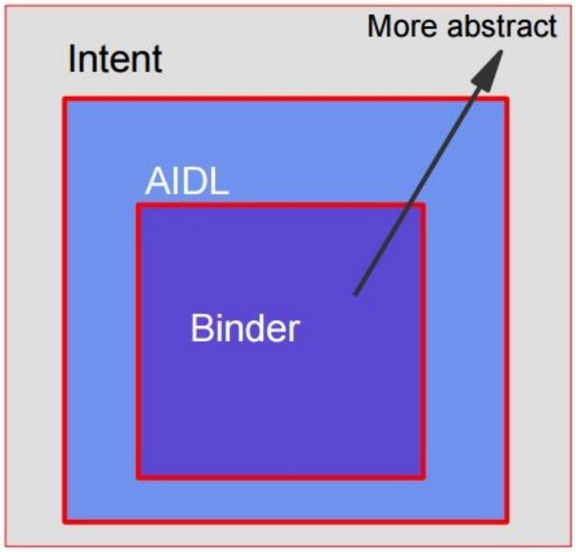


Fig. 2.8.: Android binder access abstraction

call. However, when used with AIDL we need to define the interface by which remote method calls are dispatched and received. A service in android must be registered with a service manager. Service manager is provided by the Android OS and is written in Java.

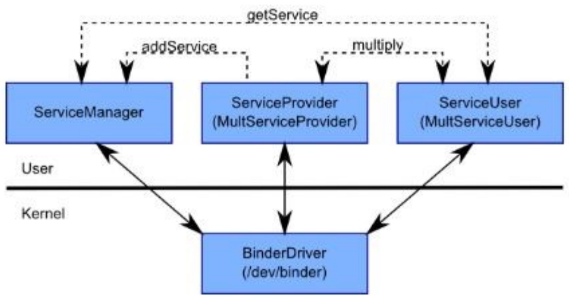

getService
addService
multiply
ServiceManager
ServiceProvider (MultServiceProvider)
ServiceUser (MultServiceUser)
User
Kernel
BinderDriver (/dev/binder)

Fig. 2.9.: Communication mechanism of a multiplication service via binder

Figure 2.9 describes the interconnection between various components of Android IPC. BinderDriver which is an Android component has the core functionality of the IPC system. The job of the BinderDriver component is to dispatch the RPC data from ServiceUser to ServiceProvider.

ServiceProvider provides the service. ServiceProvider can be a service that is developed by Android OS itself or it can be a user defined service. ServiceProvider after receiving the RPC calls from ServiceUser needs to perform necessary computation on behalf of the caller and send the message back to the requested user by the Binder-Driver mechanism. Service Manager in android has a special purpose and is run by default when the Android OS boots up. This component is provided and maintained by android. Service Users are typical users who requests the service provider to do some computation. Service users uses the service from service provider to do a work as needed. Service user develops custom application by using the service provided by service provider. Example in figure 2.9 describes a service called MultiService which is basically a multiplication service. Service user can request the multiservice to do a multiplication for the user and send back the result. A typical service request and reply follows a pattern or protocol. For example, a multiplication request and reply will follow the following steps:

- Service Manager (SM) registers itself with Binder Driver with a ID 0
- Service Provider calls defaultServiceManager() method to get an IServiceManager Proxy object to talk to SM
- Service Provider registers itself to SM by invoking defaultServiceManager() -¿ (Multiplier, new MultiserviceProvider()). The RPC call requests the SM to add a service named Multiplier indicating the implementation class is called MultiServiceProvider. Call to addService is routed to SM by the Binder Driver.
- Binder Driver in addition to routing the calls also assigns an ID for ex. 1 to the new service that is being registered to the SM. This ID is hidden from the service user at this point.
- SM upon receiving the IServiceManager.addService RPC call registers the new service named Multiplier in its record.

- MultiServiceUser in order to get the Multiplier service firsts contacts with SM by invoking the defaultServiceManager() method to get a IServiceManager proxy object. /item The client invokes gerService RPC method on the proxy object with the parameter (Multiplier) to get the ID 1 that was previously assigned by the Binder driver. RPC method getService() is also routed through the Binder Driver. /item Multi Service User can now invoke the Multi Service Provider methods by MultiServiceProvider.multiply(int a, int b)
- The multiply method invocation also travels through the Binder Driver to the Service Provider
- Service Provider upon receiving the invocation does the calculation and returns the result to Service User via the Binder Driver.

2.6 Deluge

Deluge [8] is a data transmission protocol implemented in TinyOS and written in NesC programming language. It basically disseminates the object or program binaries over wireless network. It uses another protocol Drip to disseminate packets over the network. Drip uses Trickle [9] timer to control the dissemination rate in the network. Initially, Trickle starts to transmit at a short timeout interval but the data rate exponentially decreases as time passes. Trickle tracks packet sending rounds. In the first round of transmission, sending timer is set to the lowest threshold and if new data is there to transmit then the sending rate does not change. If in a certain round no new data is there to disseminate in the network, then sending rate is halved. This process continues until the sending timer reaches the upper threshold. Drip uses a reliable TCP like 3-way handshaking mechanism for transmission. In the first phase the data is advertised in the network. Second, nodes request for the data and finally the requested data is transmitted to the node. Deluge divides the code image in fixed size pages with each page having a number of packets. Deluge also implements block and volume manager to manage the flash storage of the nodes.

Those managers enable copy, erase and query operation for the embedded flash in the nodes. Deluge by default can hold four images in the flash of the nodes but that can be increased. Also, for validity and robustness deluge has Re-programmer guard. Re-Programmer guard checks whether a node is capable of loading and reprogramming itself a new program. Deluge even though a reliable protocol has some drawbacks. First, Deluge was originally designed for Always On nodes. But in real world testbed nodes run in LPL mode. This incompatibility causes slowness in the operation and increases power consumption which is a significant issue for nodes running on battery power. Second, Deluge cannot selectively re-program nodes based on the platform and hardware specifications. But typically in a testbed, nodes from various manufacturer and vendors are used. Also, similar hardware can run different application in the same testbed. In that case deluge cannot provide the flexibility to re-program the network as needed.

2.7 Mobile Deluge

Mobile Deluge(MD) addresses the limitations of the original Deluge protocol and implements the idea of re-programming motes in one hop neighborhood selectively. Selective re-programming solves the problem of diverse node platform in the network and also in the case of different image in same platform. In addition to that, MD also switches to a new channel and disables low power listening (LPL) while re-programming which provides performance improvement and efficiency [10] by transmitting lower number of packets and thereby reducing transmission time. MD uses broadcast to connect and detect neighboring nodes which implies that the protocol works in a single hop range. MD works with a base node and one or more target nodes. Base node works as a medium to connect the computer/laptop to the 802.15.4 network. As MD uses Deluge under the hood, original TinyOS Deluge commands work as it was originally implemented in Deluge. Other commands that are specific to MD behaves in a different way. First, to reprogram target nodes a specific DISS command is sent in the original channel. Nodes that receives that command checks whether message was intended to them. If yes, they send a reply and wait for acknowledgement. If acknowledgement from base station is received, then target nodes switch their channel to the requested new channel defined in the message by the base station and disables LPL. Base node stops sending the message if it gets reply from all the target nodes. In case it doesnt receive the reply, it will keep sending the message until a threshold value is reached and then switch over to the new channel. Now, deluge base and the target nodes are in new channel and target nodes are in always on mode. The next step is to issue the dissemination command to reprogram the nodes. MD also provide commands to abort the dissemination. The abort command is there to retain nodes that does not need to be re-programmed. Nodes that receives the abort command while waiting in new channel for re-programming switches back to the original channel and enable LPL. While sending the Abort command the base node does not except acknowledgements. Hence, nodes after receiving the Abort command switches back to the original channel immediately. But for reliable transmission of the message Abort command is broad casted more than once. As we know, wireless is a broadcast medium so when deluge base transmits the message in the medium all the nodes receive the message. After receiving the message, nodes then check in the message structure whether that specific message was indeed sent for it. If it is, only then it will process the message. Otherwise it will simply ignore the message. Nodes behave different way for DISS and Abort messages. If it receives DISS message, it will send a reply packet and will wait for an acknowledgement back. If It receives the acknowledgement for the packet, then switches to new channel for re-programming. In case, if it does not receive the acknowledgement packet after several transmissions, it will ignore the message. But for the Abort message once received it will immediately go back to the original channel and enable LPL. While being in the new channel target nodes starts a timer. If timer is triggered and no re-programming command is received by the node, then it will reset itself to the original channel and enable LPL. Moreover, MD uses a modified mechanism of the original Drip protocol called SimpleDrip, to disseminate the message to the target nodes. In this modified version, original Drips many-to-many transmission pattern is simplified to one-to-many. Unlike the original Drip implementation where every node sends data periodically accordingly Trickle timer, it is only the base station who send the data packets. Hence, the packet only travels single hop. A linearly increasing timer is followed in the base node that will broadcast the new packets to the network. Receiver nodes on the other hand, does not reply. This method of dissemination is simpler and generates lower traffic than traditional Drip protocol. However, MD is not a secure protocol since it is built on top of Deluge. A secure implementation of Deluge named Seluge [11] exists in the literature along with DoS attack resistant features. Improvements to original Seluge protocol also exists as LR-Seluge [12] and FEC-Seluge [13]

2.8 SQLite

SQLite is an open source RDBMS (Relational Database Management System) and is integrated with Android OS. It provides ACID (Atomicity, Consistency, Integrity and Durability) transactions. SQLite database is server-less and does not require a complex client server model. Typically, RDBMS is implemented as a client-server model where client sends query to the database server running in a different process via connector. This process involves inter-process communication overhead and latency. Moreover, this method is not suitable for embedded small applications. SQLite solves this problem by simplifying the architecture by storing the database as a single file in the local file system. In a SQLite DBMS, database access can be as simple as a simple function call in the program. Database access control is managed by the file system permission set by the Operating System in which the database file is stored. Database user does not need to setup complex access control to use the database system. Traditional RDBMS serves concurrent database write requests by internal locking mechanism in the daemon process but SQLite only relies on the file system lock in the OS. Therefore SQLite is not suitable for serving concurrent write requests from various processes however SQLite can serve multiple read requests concurrently. Small embedded applications typically does not require high concurrency which makes SQLite suitable for embedded applications.

3. DESIGN & IMPLEMENTATION

To talk with 802.15.4 network Android OS must have to send and receive wireless messages that are sent over the network. But Android device itself does not have any hardware device that can understand 802.15.4 wireless message. Hence we have used a mote (TelosB) as a bridge between our mobile device and Android. Then, we have implemented the serial stack of TinyOS in the Android OS to understand the messages that are being sent by other motes running TinyOS. Finally, we have evaluated our work by implementing MD on top of our stack.

MD is implemented based on the idea of the paper [10] Mobile Code Dissemination for Wireless Sensor Networks with some improvements that provide more stability in the re-programming operation. One added feature is to re-program multiple mote type by a single device. For example, in previous implementation it was necessary to change the base station with specific mote image for that platform. Now, it is possible to store up to three images in the base station for re-programming multiple platforms. Secondly, due to a bug in Deluge protocol itself a mote after receiving the re-programming command sometime does not re-program itself and become in-responsive. Then the mote needs to be re-started again to become accessible which is inconvenient and sometime impossible without human interaction. Primary cause of the in-responsiveness is due to the voltage limitation of the target mote that needs to be re-programmed. Although this safety mechanism saves the mote from corrupting its image in case of power failure but the mote becomes in-responsive and does not respond to the Abort command which instructs the mote to return to the original channel i.e. Channel other than the re-programming channel. In our app design, before issuing re-program command each nodes voltage is checked and filtered per the specification of Deluge T2 [14]. Thus, the command is only sent to those devices who has met the minimum voltage criterion thereby solving the in-responsiveness problem. The application also provides a handy feature for advanced users to send the re-program command even though the voltage is lower than the standard defined in the Deluge T2 documentation for testing and debugging purposes. Currently our app supports six commands. They are:

1. Connect: This command connects to mote that the user input in the app by node id. For single click this command checks the input node voltage and issues them the dissemination command if voltage is within the range. On the other hand, long pressing the button sends command unconditionally to the given nodes. Following diagram illustrates the command flow:

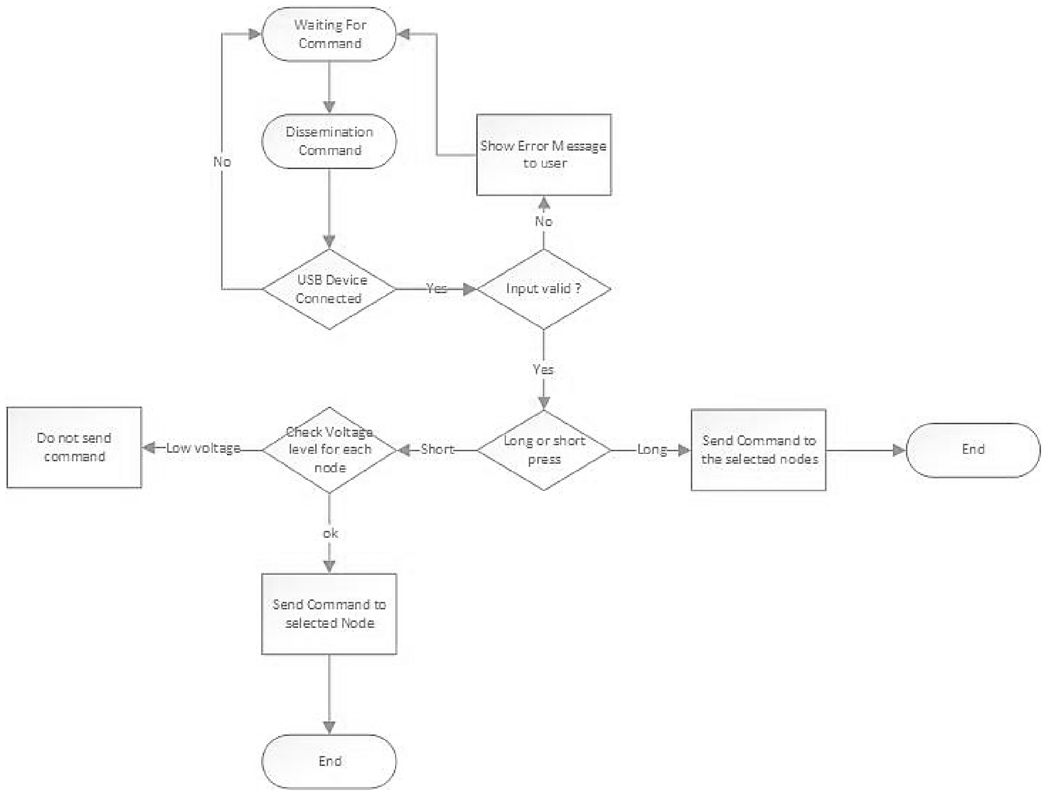


Fig. 3.1.: Connect command workflow

2. Disseminate: This command is used for starting the re-programming process. Once nodes receive the command they start to re-program themselves and

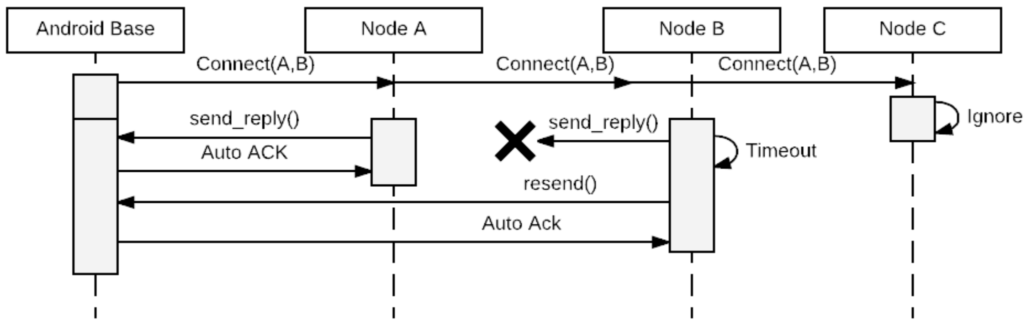


Fig. 3.2.: Connect command sequence diagram

restart. After the restart the node starts to run the new code image that was disseminated

3. Stop Dissemination: Stops the dissemination process. Nodes after receiving the command changes back to normal operating channel from the dissemination channel.

4. Abort Dissemination: This command is used in case if some node id is accidentally provided in connect command to instruct the specific nodes to return to normal operating channel instead of the dissemination channel.

5. Detect: Detect Neighbors is used for detecting the nearby nodes and their voltage. The command also provides the version number and the platform as well as node id of the nearby devices. Nodes after receiving this command replies to the node with the above-mentioned information. Figure 3.3 describes the protocol logic and corresponding output in the android phone is shown in figure 3.4

6. Detect Subset: Detect subset command behaves similar way as detect command but it provides the granularity to detect specific nodes by node id. This feature can be helpful if we would like to know the status of a specific node. Figure 3.5

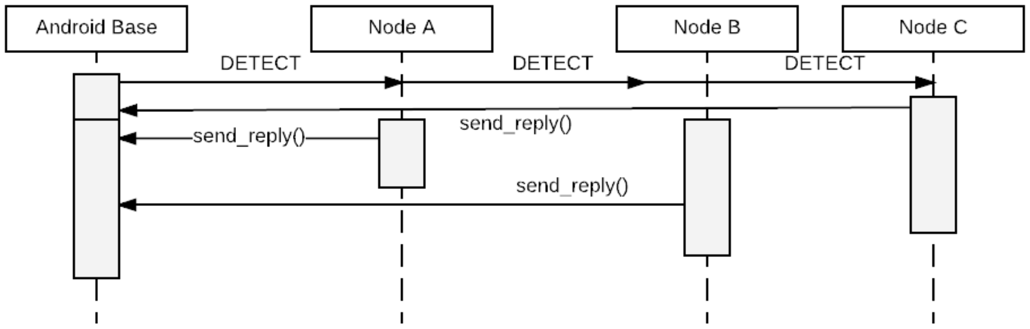
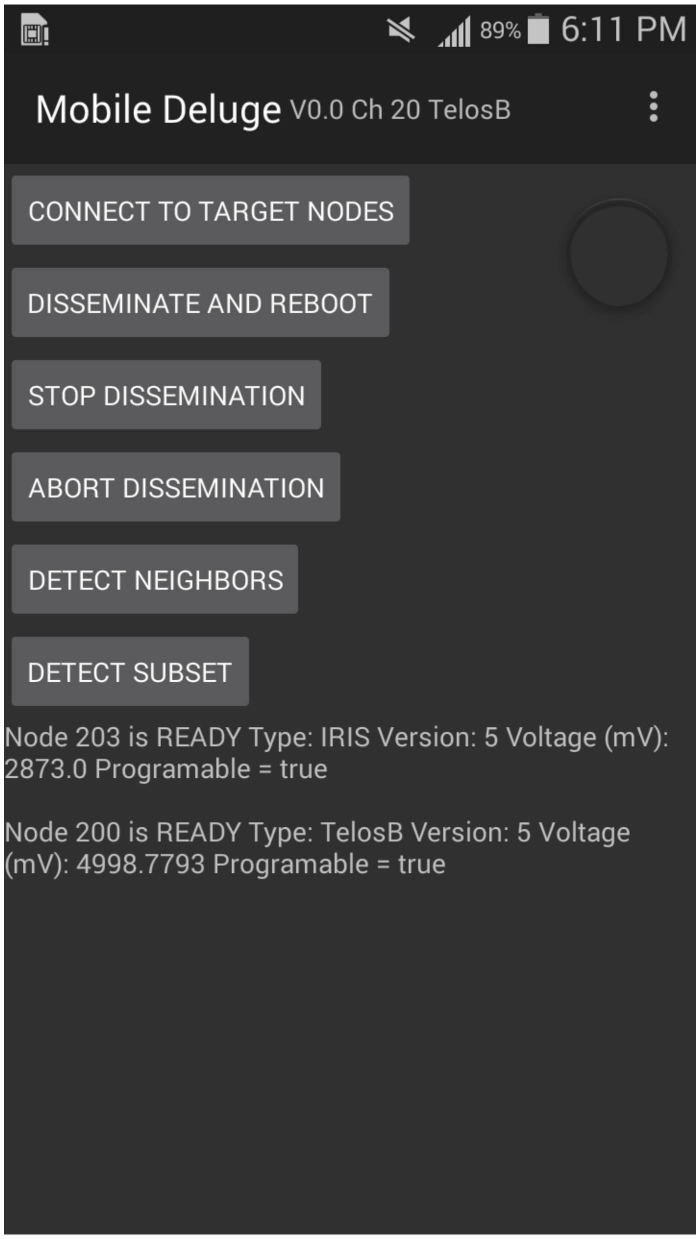


Fig. 3.3.: Detect command sequence diagram

describes the protocol logic and corresponding output in the android phone is shown in figure 3.6

3.1 Application Architecture

MD application has a layered architecture with each layer implementing a separate functionality. Figure 3.7 describes the layer and their interaction. Application layer is part of the application module. Application codes reside in this module. Complex application codes can be written in this layer using high level languages like Java. Mobile Deluge application logic is written in this layer. Likewise, if we want to implement other functionalities then we can do so in this layer. Message, Packet and Data Link layer is packaged as TinyOS SDK module. This module includes bulk of the code that is ported from the original TinyOS SDK and contains all the network stack operations starting from bytes to TinyOS message. Physical layer is implemented as Communication module. This module includes the library [15] which facilitates the communication between the hardware mote and the Android OS. TinyOS SDK and communication module is implemented as application engine. Application module codes call TinyOS SDK and Communication module by designated interfaces. Application module also connects to a separate module SQLite Database. SQLite database is natively available in Android. Database is used for internal accounting



```
Node 203 is READY Type: IRIS Version: 5 Voltage (mV):
2873.0 Programable = true

Node 200 is READY Type: TelosB Version: 5 Voltage
(mV): 4998.7793 Programable = true
```

Fig. 3.4.: Detect Command Output in Android Application

of the re-programming process. Finally, Android USB is the physical port where the device is connected. Following sections describes each module in detail.

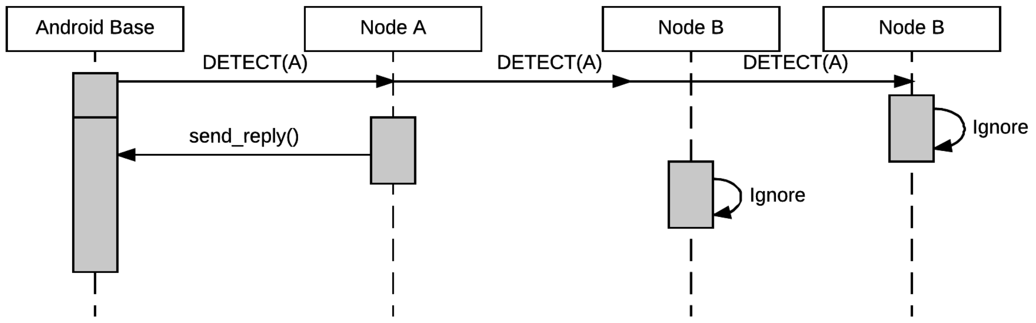


Fig. 3.5.: Detect subset command sequence diagram

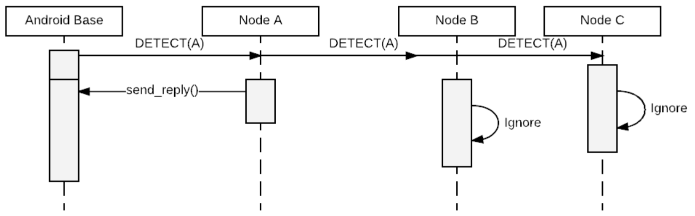


Fig. 3.6.: Detect Subset Command Output in Android Application

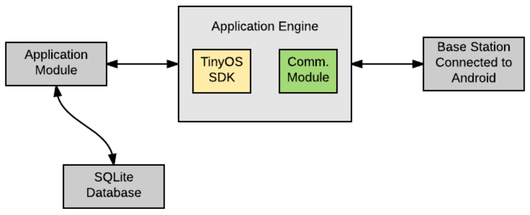


Fig. 3.7.: Android Application Architecture

3.1.1 Application Module Implementation

Android application when launched is governed by a single thread called the main thread or UI thread. When our application is first launched, it requests android framework to create a service called UsbService. UsbService is used to communicate with the 802.15.4 hardware interface which connects to the mobile device via micro USB port. Since the 802.15.4 hardware in our case is a TelosB/MicaZ mote which only supports USB interface. Therefore an USB OTG cable is used. Once the application is started, UsbService probes for hardware device with specific PID (Product ID) and VID (Vendor ID). If present, then UsbService class notifies MainActivity that a USB device of our desired PID and VID is found and connects to the hardware by USBSerial library [15]. Otherwise, UsbService sends broadcast notification to MainActivity which then let the user know that no USB hardware is connected. However, if at a later time Android senses that a new USB hardware is connected, UsbService checks the devices VID and PID and tries to connect to it if supported VID/PID is found. UsbSerial library provides convenient functionality to write and read asynchronously to underlying FTDI chip based hardware device. The library under the hood uses methods from android.hardware.usb package [16] . Also, there are some FTDI hardware specific configuration that UsbSerial configures automatically like the header, stopbit, flow control etc. Once UsbSerial successfully connects to the device, multiple threads are started to communicate with the device. Figure 3.8 illustrates the initialization of the application.

The mote-Android communication is provided by moteIF class. MoteIF provides commands to send and receiving data to and from TinyOS communication stack respectively. MoteIF also spins up another thread that constructs TinyOS packet from the frame and pushes them into Message layer of the TinyOS stack. Phoenix thread constructs the packet by obtaining the TinyOS frame from the data link layer. Different application can register for their specific message types with MoteIF to get call back when the desired message is received from the message layer. After receiving the message application can process the data as needed. Typical data transmission from host to device works in the following way. First, TinyOS serial protocol stacks message layer is used to construct a TinyOS Message object. Then this message is framed to be compatible with HDLC serial protocol by the link layer. The constructed frame then converted to byte stream. Finally, the byte stream is sent to usbSerial service component of the Mobile Deluge application. Usbservice then

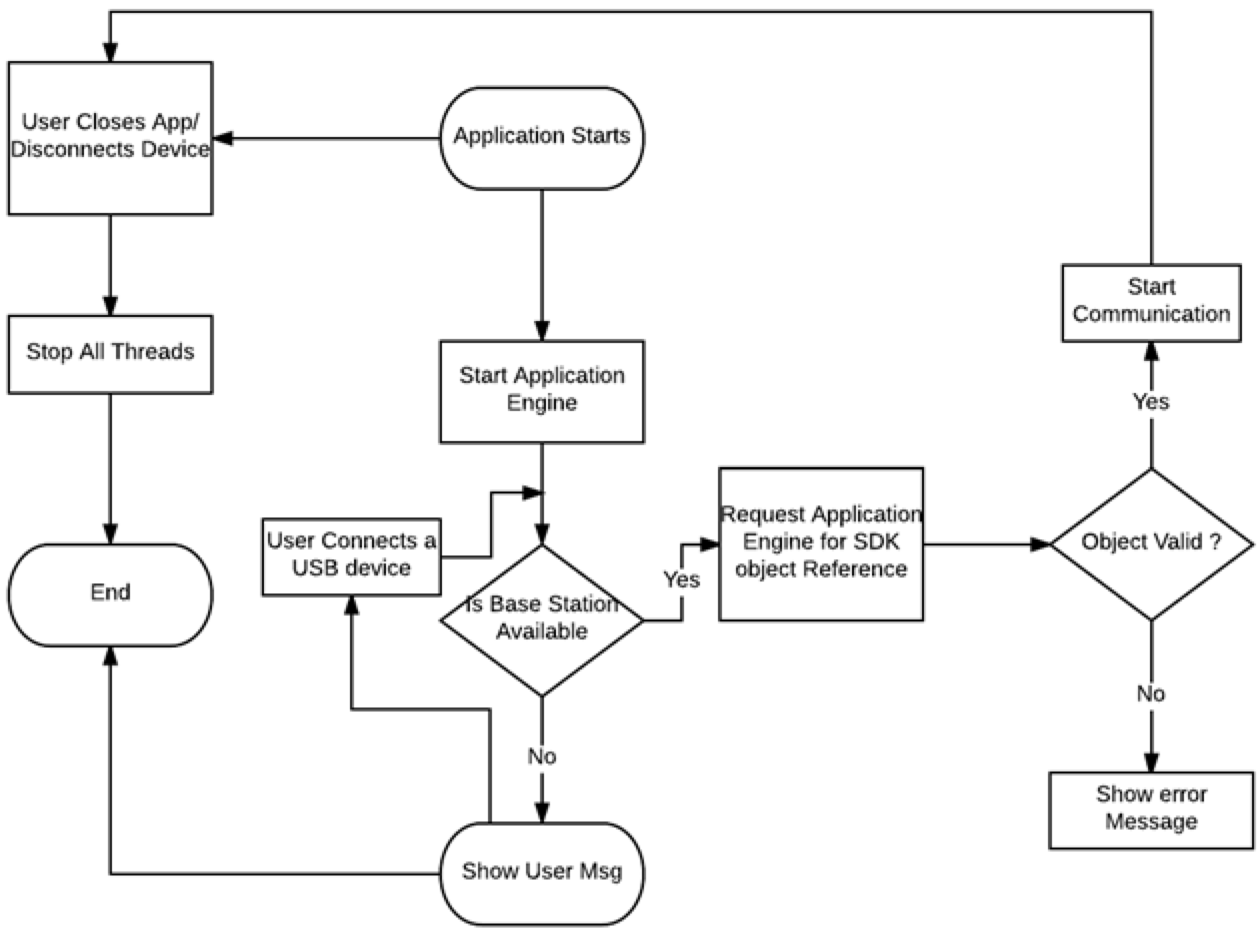

User Closes App/ Disconnects Device
Application Starts
Stop All Threads
Start Application Engine
Start Communication
Yes
User Connects a USB device
Request Application Engine for SDK object Reference
Object Valid ?
Yes
Is Base Station Available
End
No
No
Show error Message
Show User Msg

Fig. 3.8.: Application Initialization

sends the raw byte to the connected hardware via Android.hardware.usb package. Figure 3.9 illustrates the data flow in the application

Data transmission from hardware to the mobile device goes in the reverse way. Starting from hardware it then goes to the UsbService layer. UsbService layer takes care of the FTDI vendor specific encoding, baud rate and flow control mechanism. Afterwards bytes travel up to the application layer via the TinyOS serial Stack.

Mobile Deluge (MD) android application works as a controller for the mobile base mote. Mobile base hardware motes are connected to the android phone via the USB

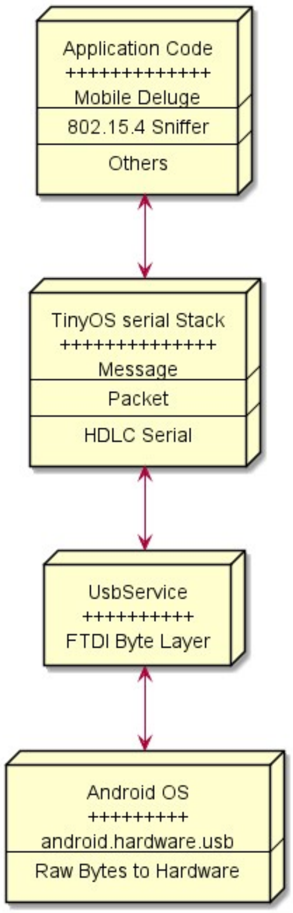


Fig. 3.9.: Different component of the application

interface available in the android device. Users can interact with the Wireless sensor networks(WSN) by sending predefined commands. Commands are executed by the mote base by using a button click from the android application. The hardware mote connected to the phone is called the base since, it acts a bridge between the android application and the other motes running mobile deluge protocol. The command packet is sent to the designated mote for further operation.

MD is based on modified version of original Deluge protocol of TinyOS operating system. MD also supports additional packet format to facilitate the one hop re-programming. In one hop communication, the base station sends command to each node separately. Dissemination command is used to start the re-programming process

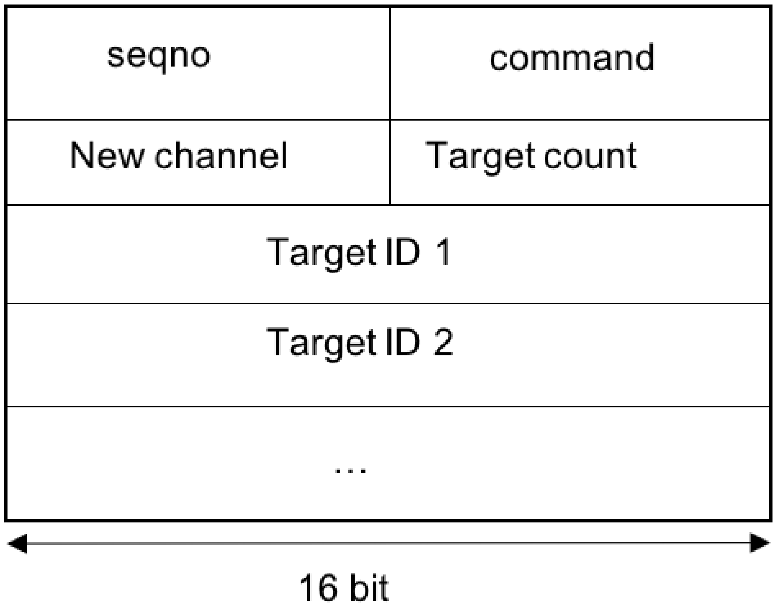


Fig. 3.10.: Command Packet Structure

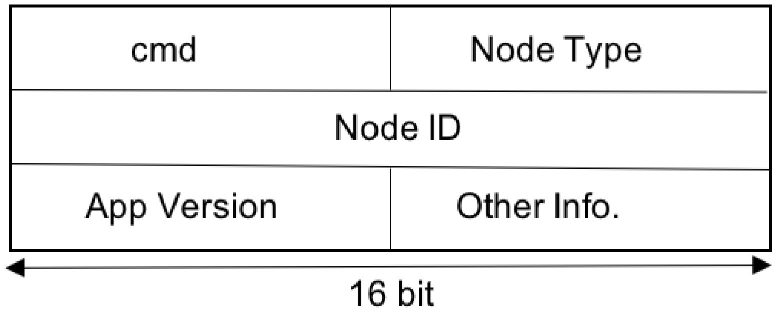


Fig. 3.11.: Reply Packet Structure

whereas the Abort command is used to stop the dissemination the process. There are two commands in MD communication as defined in [10]. They are called Command Packet and Reply Packet. In the beginning, the command packet as defined in figure 3.10 is dispatched from the base to selected nodes by Node ID, the selected motes residing in the wireless range of the base replies back to the base.The reply packet is then sent back to the android application for further processing. Dissemination command instructs other nodes to switch to a new channel and stop the default Low

Power Listening(LPL) behavior. In LPL mode nodes periodically wake up to transmit wireless traffic and other time remains in standby mode to preserve power. The nodes in reply send the reply packet in the format defined in figure 3.11. Base mote always acknowledges the reply packet received from the target motes thereby making a three-way handshake. Once the base gets reply from all the nodes or the maximum number of packet is sent, base changes the channel to new channel for re-programming. Then it is possible to re-program the nodes by regular Deluge commands. However, if for some reason some nodes are selected for re-programming in error, Abort command is used to let those nodes know to change back to their default operating channel. Unlike Dissemination command, Abort command takes in effect immediately without any acknowledgement from the target node to base. After receiving the Abort command target nodes check the packet to make sure abort packet node id matches its own node id. If it is a match then the node starts LPL and switches back to default channel. A detailed analysis of this communication logic in described in figure 3.12 and 3.13. The figure is a slightly modified version of the state machine described in [10]

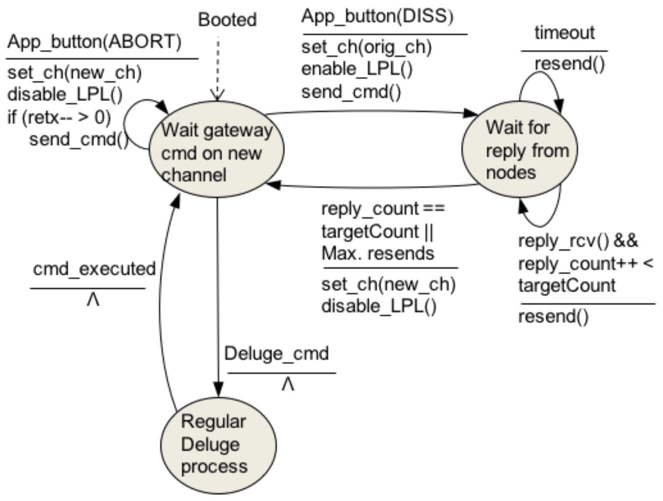


Fig. 3.12.: Base mote control logic

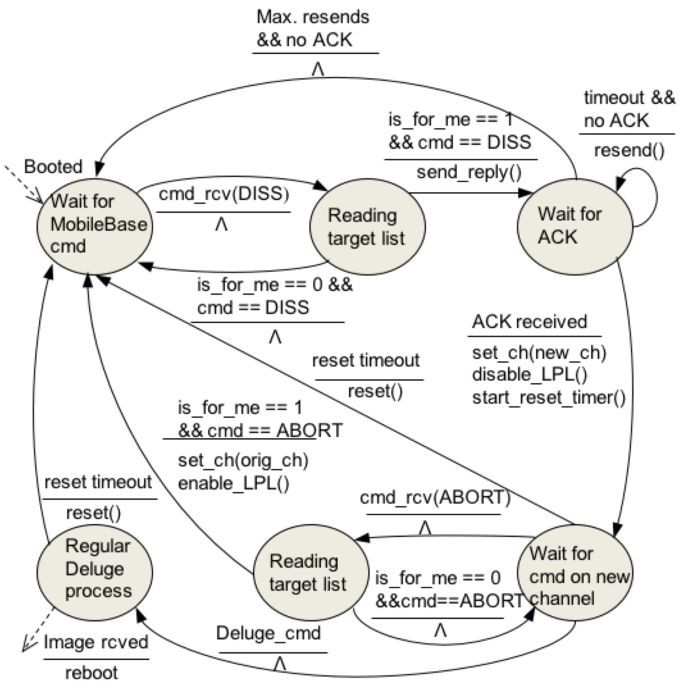


Fig. 3.13.: Target mote control logic

3.2 Application Engine Implementation & Design

Application Engine is comprised of TinyOS SDK module and Communication Module. TinyOS SDK module is responsible for Encoding/Decoding RAW HDLC protocol payload. Communication module on the other hand is responsible for Encoding/Decoding RAW hardware bytes connected to the USB port.TinyOS SDK operation and Communication module operation is run concurrently using separate threads. Once a supported device is found in the Android phone the required threads are started to start communication with the base station mote. Figure 3.14 describes the initialization phase of the application engine.

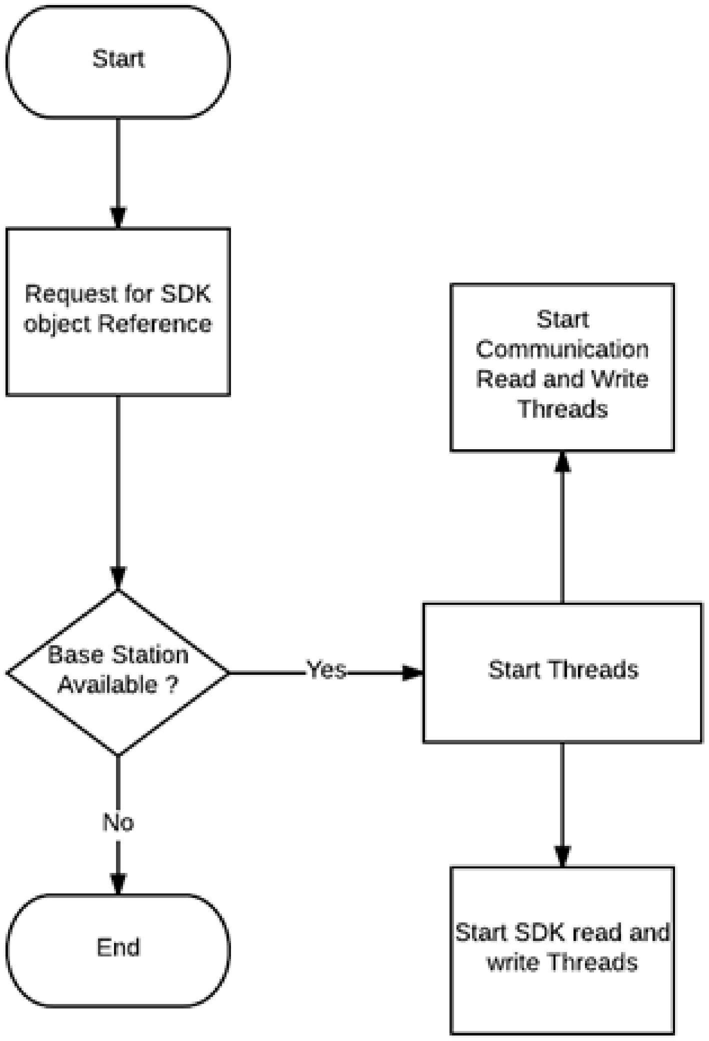

Start
Request for SDK object Reference
Base Station Available ?
Yes
No
End
Start Threads
Start Communication Read and Write Threads
Start SDK read and write Threads

Fig. 3.14.: Application Engine Initialization

3.2.1 TinyOS SDK & Communication Module Design and Implementation

TinyOS SDK (Software Development Kit) is not available for Android OS. So we had to port it with few modification from original SDK that is targeted for Linux OS. Much of the communication logic remains the same to maintain compatibility with all the devices. TinyOS SDK implementation can be subdivided into two categories. They are:

1. Serial Stack
2. Network Stack

Serial Stack Architecture in TinyOS

To access the wireless sensor network nodes running TinyOS we need to access motes via an interface. The interface can vary and depends on the application. It can be USB, network but motes talk via UART (serial) port. TinyOS supports platform independent serial communication protocol thus supporting heterogeneous WSN motes UART packet format. TinyOS serial communication stack can be broken into five parts [17]. They are

- Serial AM
- Dispatcher
- Protocol
- Encoder/Framer
- Raw UART

The lowest level of the stack is RAW UART. This layer can be compared to Physical layer of OSI model. This layer is closer to hardware and gives the user the opportunity to setup UART configuration settings like Buadrate, Stop byte, flow control and others. This layer is also responsible to flush the hardware buffer before sending or receiving new information. This is the platform for reading and writing bytes over serial connection. It provides byte level interface to the layer above it. Next layer upward is the Encode/Framer. In this layer, raw bytes are translated into HDLC [18] protocol bytes. HDLC bytes are constructed using serial protocols encoding. Once raw bytes are received encoder/framer layer looks for two kind of bytes. They are delimiters and data bytes. Delimiter and data bytes are handled differently in this layer as well as the layer above it. Encoding/Framing is done similarly to the HDLC protocol. In HDLC scheme, 0x7e is reserved as delimiter byte and 0x7d is called an escape byte. If this layer receives the delimiter byte it informs the upper layer accordingly. In case it receives an escape byte, it sets an escape flag. When it receives other bytes it checks whether the escape flag is set if set, it then XORs incoming data byte with 0x20 and passes to the upper layer as data byte and clears the escape flag. Similarly, on the transmit side, if the layer is instructed to send a delimiter or escape byte as a data byte it sets the transmit flag and stores the delimiter or escape byte XOR 20. Then, it sends the escape byte 0x7d to the layer below it. Once the escape byte is successfully transmitted then it sends the stored byte and clears the flag. This layer also provides byte interfaces to the layer above and below it but provides encoding while sending and decoding while receiving bytes. Protocol layer sits on top of the Encoder/Framer layer and provide byte level interface to the layer above it. This layer implements the serial protocol using PPP/HDLC like framing. Buffer management are left to the layer above it. This protocol is stop-and-wait in the host-to-mote direction and best effort in mote-to-host. Stop-and-Wait refers to the fact that, host side PC/Android after sending one packet wait for an acknowledgement from the mote. This design was due to the fact that; motes usually have low buffer size and stability is considered more important than throughput in these devices. While sending protocol layer is responsible for encapsulation of the upper layer packets. The sending process starts from the layer above it with a method call. Protocol layer then collects subsequent bytes from the upper layer by generating an event until another method call from above layer declares the end of packet. This layer incorporates a small sending window for collecting bytes that are yet to send in the lower layer. The sending window can be customized based on the link condition. Protocol layer also sends an event to the upper layer indicating that the stream of bytes of previous packet was sent successfully or not. Packet reception in the protocol layer is started once it receives an inter-frame delimiter and new packets header from the layer below it. Protocol layer indicates its upper layer that a serial packet is arriving and each byte is signaled to the layer above it. After the frame is fully received upper layer is informed with a success or failure message. In the current implementation, protocol layer acknowledges the frame it receives. But it does not expect acknowledgement for the data sent by it. Acknowledgements are of high priority and has separate queue than data packets. Hence, it is possible that data packet to be sent may begin spooling into protocol layer while it is actively sending an acknowledgement. Dispatcher is the layer above the protocol. Dispatcher layer sends and receives serial packets. Dispatcher understands TinyOS message structure message_t and decodes the received data bytes accordingly. When dispatcher receives the known packet type then it determines the offset from which the payload starts. It then spools data bytes and fills the message_t buffer. Per the same logic when dispatcher receives command from upper layer to send data bytes it first sends the type byte i.e. type of the message. Then it sends the data from the index denoted by the offset i.e. from which the payload starts. Serial packet format:

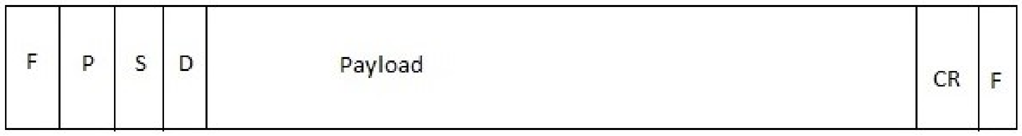


Fig. 3.15.: Serial Packet Format

F = Framing Byte, P = Protocol Byte, S = Sequence Number Byte, D = Packet format dispatch byte, Payload = data payload, CR = two byte CRC over S to end of the payload, F = Framing byte denoting the end of the packet. As we can see from figure 3.15 serial packet always start with a framing byte. TinyOS defined this byte as (x7e). If any byte from P to CR starts with 0x7e or 0x7d it will be escaped to 0x7d 0x5e or 0x7d 0x5d respectively. Layer above the dispatcher is the Active Message (AM) layer. AM layer provide virtualized access to the serial stack of TinyOS. Each application code creates an instance of AM Sender or AM Receiver component to send or receive serial message respectively. AM Sender or Receiver component each have their own queue thus different application trying to access the serial stack does not have to contend with each others sending or receiving queue. TinyOS implements a fair share scheduler to schedule processing time to each application sending or receiving process. Sender and Receiver instantiations wires themselves with this layer with unique AM ID to send or receive serial messages.

Message Buffer in TinyOS

Message buffer implementation in TinyOS is called message_t [19]. Message_t is a designed to be a contiguous region of memory location. Contiguous memory location enables datagram transfer between different link layer within a platform without copying. TinyOS messages are defined as Abstract Data Type (ADT). Above layers always access the data link layer fields via interface. These interfaces are implemented with get/set operations that take and return values. The message_t structure is TinyOS is defined as follows:

```
typedef nx_struct message_t {
    nx_uint8_t header[sizeof(message_header_t)];
    nx_uint8_t data[TOSH_DATA_LENGTH];
    nx_uint8_t footer[sizeof(message_footer_t)];
    nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

This format keeps data in a fixed offset from the header. Also, the footer and the metadata are well defined in the structure. Inconsistent data payload offset would require a memmove operation (basically a copy). The keyword nx* stands for external type; nx types ensure cross platform compatibility [20]. It also forces structures to be aligned on byte boundaries. However, a platform may have multiple link layer implementation. For example, a platform sample_platform may have two different radio stack and a serial stack. Each of the link layer defines their own structure. If a platform has both CC1000, CC2420 radio stack then it should define a header file like the following:

```
typedef union sample_platform_header {
    cc1000_header_t cc1k;
    cc2420_header_t cc2420;
    serial_header_t serial;
} message_t
typedef union sample_platform_footer {
    cc1000_footer_t cc1k;
    cc2420_footer_t cc2420;
} message_footer_t;
typedef union sample_platform_metadata {
    cc1000_metadata_t cc1k;
    cc2420_metadata_t cc2420;
} message_metadata_t;
```

Union keyword ensures that enough space is allocated for all underlying link layers. The message_t structure first defines the message_header. If a platform has multiple link layers, then the message_header will be the union of all data link headers. For example, Telos platform has the following setup for the message_header:

```
 typedef union message_header {
        cc2420_header_t cc2420;
        serial_header_t serial;
} message_header_t;
```

The cc2420 header size is 11 bytes. Serial header ends at the beginning of the payload but is only 5 bytes. If radio message is being sent, then full 11 bytes are filled up but serial message cannot fill the space. So, a padding of 6 bytes are provided before actual 5 bytes of serial header thus header and data payload remain in contiguous memory location.

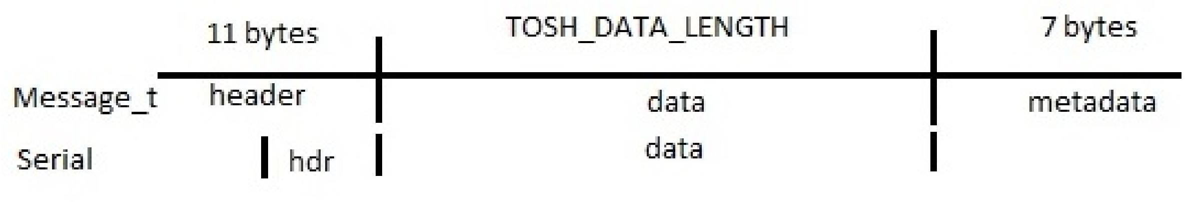


Fig. 3.16.: Serial and message packet structure

As we can see from the figure 3.16 above, serial header has only 5 bytes of header and no meta data. The hdr portion is accessed by an offset from the data field. The offset is calculated by taking the negative offset from the data field. Offset = data sizeof(serial_header_t) Offset value is used to decode and encode TinyOS messages (message_t). Metadata are the fields that are used for internal accounting and not transmitted over the network. For example, wireless metadata includes transmission power, received signal strength, CRC byte, acknowledgement information and others. TinyOS message data structure is optimized for fixed size packets but variable size packets can also be supported. In our implementation, we have only considered fixed size packet headers and footers.

3.2.2 TinyOS Network Stack Design in Android:

To communicate with the TinyOS serial stack, exact network protocol and layers must be implemented in the android platform. Android does not natively support the Software Development Kit (SDK) of TinyOS which is typically used for communication between foreign host and TinyOS serial stack. Android does not recognize the motes either. Hence, we have used the typical layered approach to build a communication mechanism between Android OS and TinyOS by porting the TinyOS SDK to the android platform. In the SDK, Layered approach abstracts complex low level

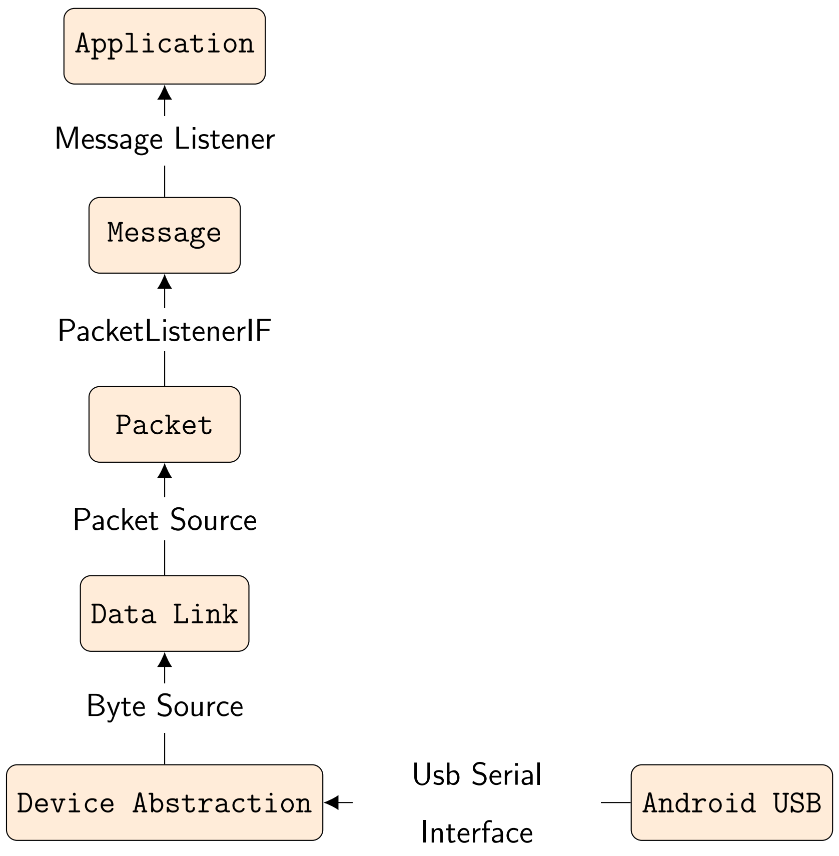


Fig. 3.17.: Serial Communication Stack

configuration and provides flexibility to user to develop any application on top of it. The communication between TinyOS and Android happens by following the layered approach described below. In figure **??** the layers are described bottom up. First, is the Android USB port where the physical device is connected. This layer is closest to the hardware. It is android USB interface. In this layer signal is passed in binary form 0 or 1. Bit level access is abstracted away by Android USB Manager. Android provides USB access by android.hardware.usb package. The package contains Usb-Manager. UsbManager and all other related classes lies in the application framework layer in android architecture. UsbManager also provides the information about the state of the USB and can communicate with the hardware. Android.hardware.usb package also contains UsbDevice class which enables the user to communicate with hardware in case the android is acting as a Usb host. Usb host refers to the fact that, when the device is connected in the USB port then the device gets power from the connected USB interface. Now, in our case, android phone is used as USB host and we are using a TelosB/MicaZ/Iris device which has a FTDI chip that does the communication over the USB channel. RAW bytes from the devices with hardware specific encoding are sent from the Android USB layer to the Device Abstraction Layer. This layer abstracts away different vendor specific configuration needed to get the payload. For example, FTDI devices has its own set of headers, specific codes to change buadrate, stopbits, error correction mechanisms and communication mechanism that are not natively implemented in android and is taken care of in this layer.

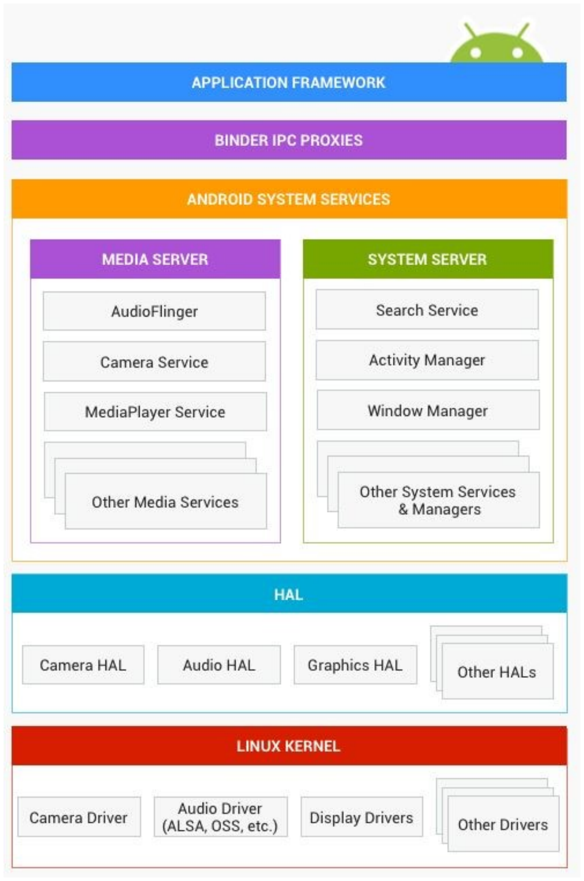


Fig. 3.18.: Android system architecture

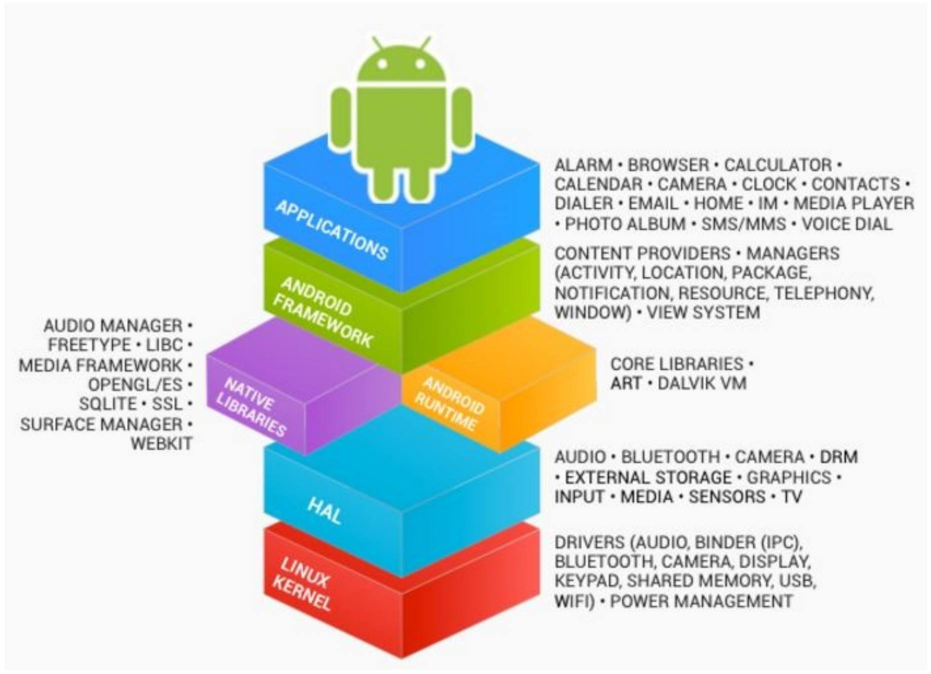

APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • ART • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

Fig. 3.19.: Android stack details

To facilitate the operation in this layer, we have used an open source library [15] to do the bit level communication between FTDI device and android. The library uses android.hardware.usb package to connect and send data to the FTDI device. Android comes with an USB package that is implemented as a system service and uses IPC (Inter process communication) to call native function implementing USB connectivity in Linux kernel. As described in figure 3.19 and figure 3.18 android.hardware.usb package is under System Server but there are internal mechanisms to call native codes in Linux Kernel. Android supports accessing the underlying hardware both synchronously and asynchronously. UsbRequest is a class that requests data from hardware. Synchronous requests can only be sent to hardware endpoint that supports bulk transfer. They are also called bulk endpoints. However, interrupt endpoints support both bulk and asynchronous transfer. Asynchronous transmission is implemented internally by android system by posting data request from hardware in a queue. Threads then check the status of the request whether requested data is available. If data is available, it is passed to the Device Abstraction Layer by Android OS. All these functions are taken care of by the open source library discussed above. Device Abstraction layer implements the functionality of the Communication Module. Data from WSN motes in the form of bits are transferred from the device to the android phone. Then, FTDI specific headers are striped out and data payload is pushed up in the Data Link layer. In our implementation, received data from the Android USB layer is passed by a asynchronous call back to UsbSerialInterface.USBReadCallback interface. Then data bytes are transmitted to upper Data Link layer. On the other hand, write operation via the Device Abstraction layer is synchronous and uses the bulk transfer of bytes on a USB endpoint. UsbReadCallback interface is defined as follows:

```
interface UsbReadCallback
 {
     void onReceivedData(byte[] data);
 }
```

Now that the payload i.e. the raw data bytes of HDLC protocol is at the Device Abstraction(DA) layer we then use Byte Source interface to pass the received payload byte-by-byte to the upper layer named Data Link. Data received and sent in this DA layer are raw TinyOS message serial bytes along with hardware specific encoded bytes. After receiving raw serial bytes from the hardware, hardware specific encapsulation bytes are stripped off and HDLC protocol payload is then passed to data link layer as stream of bytes. Data Link layer receives raw HDLC bytes via Byte Source interface. ByteSource interface is defined as follows:

```
public interface ByteSource
{
    public void open() throws IOException;
    public void close();
    public byte readByte() throws IOException;
```

```
<Packet Type> <Data Bytes 1n> <16-bit CRC>
```

Fig. 3.20.: TinyOS packet structure

```
    public void writeBytes(byte[] bytes) throws IOException;
}
```

Next layer upward is the data link layer. In this layer, raw HDLC data bytes are encoded/decoded to a meaningful frame using a HDLC like protocol. The implementation is identical to RFC 1663. Current implementation of the protocol does not contain any establishment phase. Single byte identifies the start, end, target and other key information in the byte stream. The communication protocol can be thought of two sides for ex. A & B are connected by an unreliable channel. As identical to TinyOS serial stack two sides start to exchange packets framed by a 0x7e also known as sync byte. Sync byte denotes the start and end of a frame. Each packet has the following format: CRC (Cyclic Redundancy Check) is performed on packet type and data byte 1 through n. Since we are using 0x7e as a distinguishing byte, it may be possible that data or packet byte may contain the 0x7e byte as well. To solve the problem, escape byte 0x7d is used. If however, escape byte itself is present in the payload or other fields that also needs to be escaped as well. Escaping is done in similar way as TinyOS serial stack to maintain compatibility. Any byte (0x7e or 0x7d) that needs to escaped is always XORed with 0x20 followed by 0x7d (escape byte). For example, if a payload byte contains 0x7e 0x45 0x7d after escaping the resulting byte stream will be, 0x7d 0x5e 0x45 0x7d 0x5d since 0x7e XOR 20 = 0x5e and 0x7d XOR 20 = 0x5d Protocol specification states that 0x7e and 0x7d bytes must be escaped but additionally 0x00 0x1f and 0x80 0x9f bytes can also be escaped. In our implementation, 5 packet types are defined. They are listed below:

1. P_PACKET_NO_ACK: A user generated packet with no ack required. It implies that, receiver after receiving this type of packet does not need to send and acknowledgement back to the sender.

2. P_PACKET_ACK: A user packet with ack required. Receiver understands this by a prefix byte that denotes the packet type. Receiver must send an acknowledgement packet P_ACK back to the sender using the prefix byte as its content.

3. P_ACK: This is an acknowledgement packet sent by the receiver. The P_ACK packet is sent whenever the receiver receives the P_PACKET_ACK packet.

4. P_PACKET_UNKNOWN: Unknown packets are defined as this type. This packet is sent by the receiver to the sender of the unknown packet type. After checking the packet type byte field if the receiver does not resolve the packet type to be valid then it does not process the packet and sends a reply to sender as P_PACKET_UNKNOWN. However, new type of packets can always be defined by user if need arise.

Along with the packet type, if for some reason a larger than MTU (Maximum Transmission Unit) packet size is received then it will be dropped and no information is sent back to the sender. Sender after a specific timeout period must re-send the packet. For reception, a separate thread is created and placed in a per-packet-type queue for processing. Each packet type described above has their own queue thus different packet types does not need to contend with each other for queue space. After a complete packet is formed by interpreting appropriate signal bytes it is sent to the upper packet layer via the Packet Source interface for processing. Packet layer does decoding when it receives byte stream from data link layer and encoding when it receives packet from the layer above it. Encoding includes creating a new serial packet with appropriate header, footer and payload. Bytes that needs to be escaped are taken care in this step. Then CRC is also calculated and put as a 2-byte field before the end of the serial packet. Then, the packet bytes are sent to the lower data link layer as array of bytes. Data link layer then send the packet bytes to the hardware by appropriate interface. Decoding on the other hand does the opposite job by taking off the serial packet encapsulation and payload is sent to the upper layer for further processing. Decoding starts with the parsing of incoming data bytes.

If the decoder sees a signal byte that indicates the start of the packet it parses the data bytes assuming a new packet is started. Packet layer reads incoming data byte-by-byte and creates a TinyOS serial packet to the upper layer. Couple of error and integrity checks are also done in this layer. First, incoming packet needs to pass the CRC check. Second, the packet has to be within a max MTU limit. Third, other unexpected error condition are also checked to ensure proper synchronization with the hardware. For example, if we receive a byte signaling a start of packet following an escape byte that is an error because the purpose of escape byte is to abstract the sync byte that is sent at the start and end of the packet. Finally, too small packet frames are ignored since they do not produce a meaningful message. Decoder after checking errors and other conditions discussed above sends the packet as byte array to Message layer above it. Packet Source Java interface is used to send the byte array. The Java method to read and write packets are defined as readPacket() and writePacket(). Packet Source interface is defined as follows:

```
public interface PacketSource
{
    public String getName();

    public void open(Messenger messages) throws IOException;

    public void close() throws IOException;

    public byte[] readPacket() throws IOException;

    public boolean writePacket(byte[] packet) throws IOException;
}
```

Packet layer is implemented on top of Data Link layer. In TinyOS SDK Data Link and Packet layers are not separated and their operation is combined. But a separate

PacketListenerIF is used to provide packets from Data Link layer/ Packet Layer to upper message layer. Packet format is defined in figure **??**. Ideally, Data Link layer should have been concerned with only the serial HDLC format decoding/encoding but current TinyOS SDK does packet encoding/decoding in same layer and then pushes the packet to the upper Message layer as byte array of TinyOS packet payload. PacketListenerIF is defined as follows:

```
public interface PacketListenerIF extends java.util.EventListener {
  public void packetReceived(byte[] packet);
}
```

For sending packets from Message layer however, PacketSource interface is used.

Message is implemented on top of Packet/Data Link layer. Message layer serves two purposes. One, it provides abstractions for serial and radio communication interfaces. However, another communication interface can also be added if needed. Two, it implements TinyOS message data structure that enables cross platform compatibility and endianness. In TinyOS radio and serial communication, packets are structured differently. Moreover, different platform has different message structure for the same protocol. TinyOS is open source hence any new device message structure can be added to the OS. Also, there can be platform which possess multiple radio chip or serial interface for which message structures are defined as union of multiple message structures as described earlier in this paper. TinyOS message structure enables memory alignment between different packet structure of the platform. Various message structure per the hardware are defined in TinyOS as header files. Customized message can be added if needed in TinyOS which is defined in NesC language. Android does not understand NesC hence, we must interpret the same message type in Java. For that reason, TinyOS provides a tool called MIG. MIG is a short hand for Message Interface Generator. MIG generates NesC message structure header files into equivalent Java file. The generated file by Mig is equivalent to message_t type in TinyOS. In Android environment, message_t is defined as Java type Message. In addition to that, each Message type defines a AM (Active Message) type. AM type helps distinguish between different TinyOS messages. Message layer is responsible for multiplexing different Message to the appropriate receiver. Message is dispatched to the intended receiver by a Message Listener interface. Applications register themselves with the Message layer by an AM ID. Message layer maintains accounting of AM ID when senders register themselves. After receiving a packet, message layer checks whether a receiver exists if yes, then packet is forwarded to the receiver otherwise it is dropped. Message layer also has another method for unregistering the receiver. Any sender/receiver component in the application layer can use the unregister mechanism to remove the record from the message layer. Application should unregister themselves when they are done with the communication or if the application throws exception. Message layer only sees AM ID and forwards the packet. Thus, for a single message type multiple receiver can register themselves using the register listener method. Once the message is received, a copy of the message is then forwarded to all the registered listener. This functionality enables code re-use and abstraction to the application layer. Message Listener interface is defined as follows:

```
public interface MessageListener {
    public void messageReceived(int to, Message m);
}
```

Next layer upward is the application layer. In application layer complex application code are written using high level language like Java and NesC. Most of the application program written by users reside in this layer. Various network services can also be supported by application layer such as, reliable transmission, secure transmission and error detection. Reliable transmission can be accomplished by defining a specific acknowledgement message in the application layer. If acknowledgement is not received, then data will be re-transmitted. Secure transmission can be accomplished by implementing various cryptography protocols in the application layer. Error detection can be done by CRC, parity check or another application layer error detection algorithm. Our test application, Mobile Deluge is implemented in this layer. Other applications for example, 802.15.4 sniffer and custom protocols can also be run in the same way.

Interaction between SDK, Communication and Application module is facilitated via different threads. Communication between the threads are described in figure

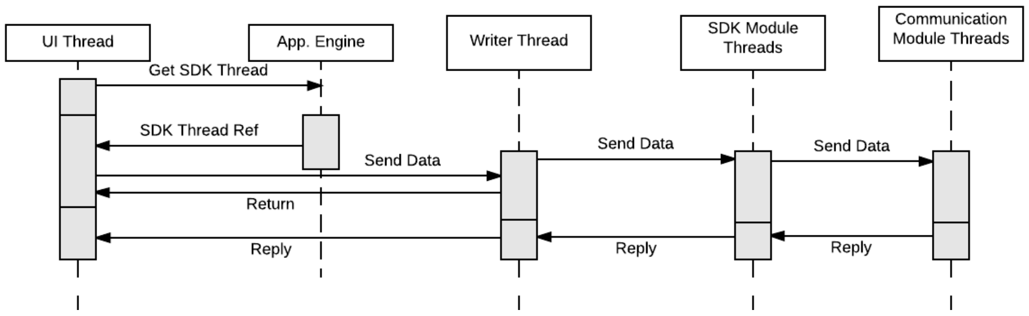
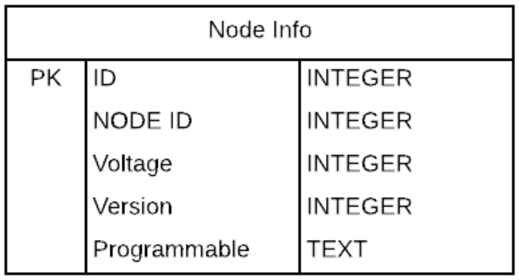


Fig. 3.21.: Communication Between modules via threads

3.21. As we can see the communication between the UI thread and other module threads are asynchronous. Asynchronous communication enables UI thread to do other works as needed while we do the communication over the network. Moreover, it is not recommended to block UI threads more than 5 seconds since it may trigger the Application Not Responding error.

3.2.3 SQLite Database Module Design and Implementation

SQLite database is implemented as a separate module from Application Engine. Application Module can communicate with the database module. Database module defines schema and organization of different tables for accounting purposes. Currently the database module is used to keep an account of the devices that have been reprogrammed successfully. Figure 3.22 describes the current database table schema to store information. As we can see from the figure we store ID field as our primary key which is managed by Android OS and auto incremented. However, each row is unique by NODE ID field which is maintained by the UNIQUE NOT NULL constraint of SQLite. Hence, each node must have to have a unique node id for communication this approach works well. Then, we also take an account of the voltage, version and programmable status of the node. Version field can be used to indicate



Node Info		
PK	ID	INTEGER
	NODE ID	INTEGER
	Voltage	INTEGER
	Version	INTEGER
	Programmable	TEXT

Fig. 3.22.: Database Schema for MD Application

a successful re-programming. We can compare version number before and after of the re-programming process to ensure that our node has successfully re-programmed itself and restarted. Voltage field store the current voltage of the mote device. Re-programmable field identifies whether a node is programmable by comparing it to a threshold voltage depending on the platform.

3.3 Android UI Design & implementation

Mobile Deluge re-programming operation is backed by a simple UI with buttons to execute the required commands. Reply packets from remote motes are shown in the user interface screen. Error conditions and information are displayed as a toast message. Following is a screen shot of the application running in Samsung Galaxy S3 phone. Figure 3.23 illustrates basic layout of the application. As we can see from the figure, the app dashboard describes the program version, reprogramming channel and target device platform on the appbar. Program version is the version of the program that will be disseminated by the application. Re-programming channel is used for transmitting new program to the selected devices. Re-programming channel can be changed from the application menu. Target device selects appropriate file for dissemination depending on the hardware. Currently three platforms are supported. They are TelosB, MicaZ and Iris. Commands buttons are used to send the appropriate commands to the device. Finally, below the command buttons there is a area to display the status or result of each command is displayed.

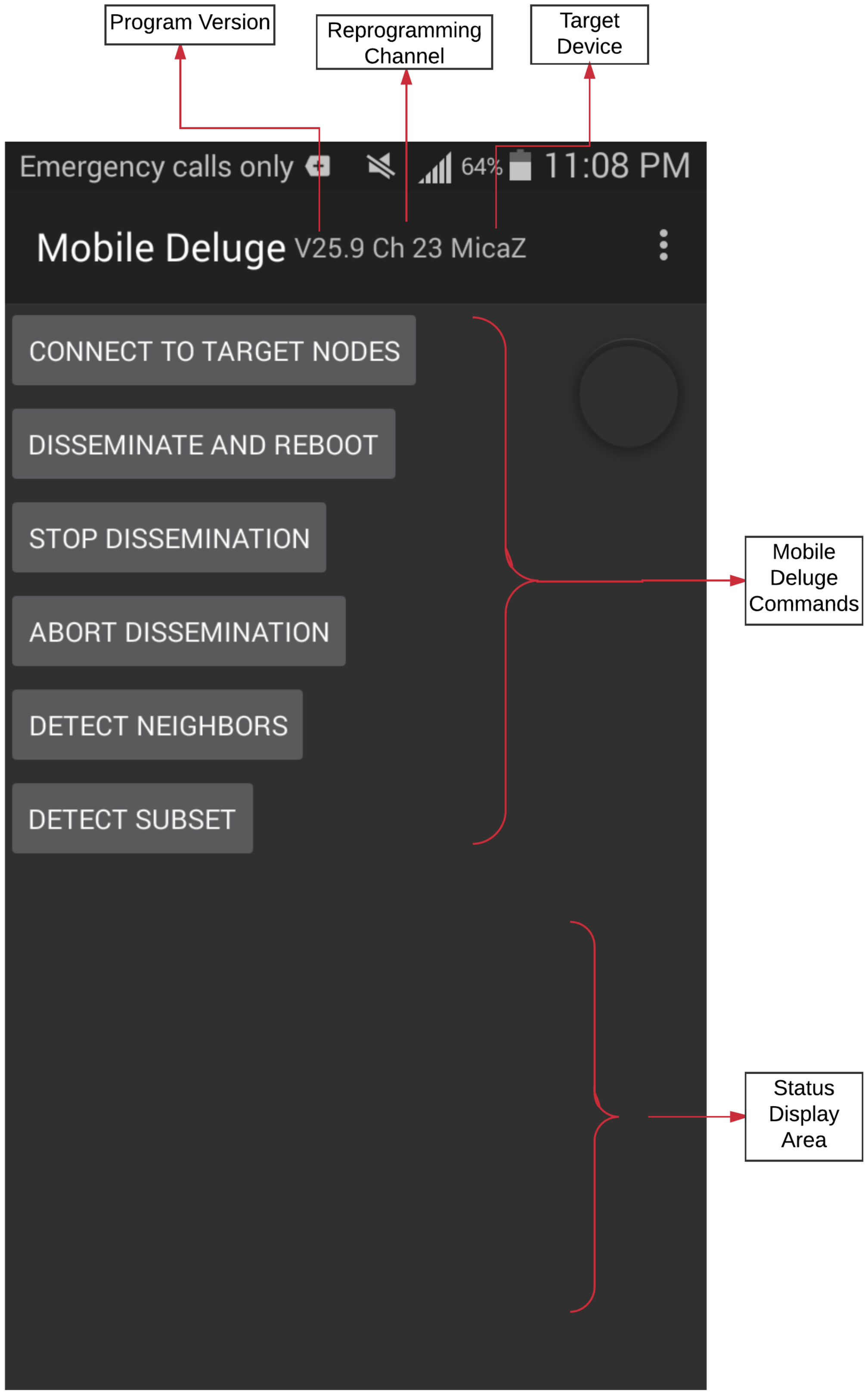


Fig. 3.23.: Mobile Deluge Mobile Application

4. PERFORMANCE & TESTING

In order to test our application we have connected a MicaZ mote to our android phone and measured some performance metrics. As discussed before, the android phone is connected via the USB OTG cable as shown in figure 4.1

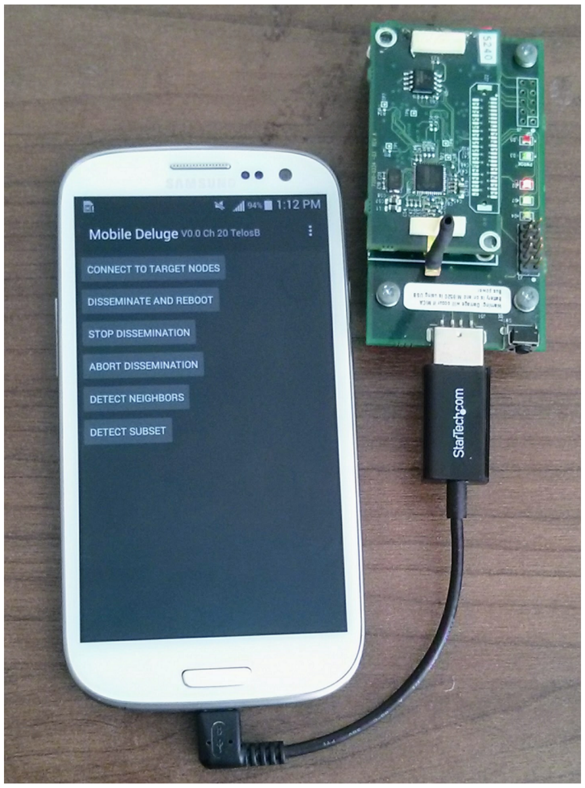


Fig. 4.1.: Mobile Base Station (MicaZ) device connected to android phone

Performance of an App can be subdivided in 3 steps. They are:

1. UI Performance

2. Application Algorithms

3. Battery Performance

4.1 UI Performance

4.1.1 Rendering

Rendering is the process of generating or drawing an image in the device screen. Android OS updates the screen or activity in every 16ms which means that developers need to complete all necessary computation to update the screen after each user interaction in that 16ms time window.

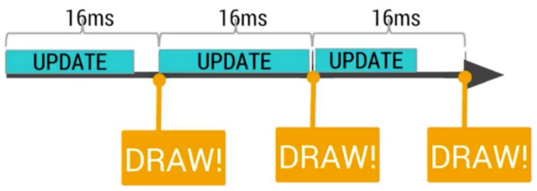


Fig. 4.2.: Android rendering intervals

If the developer fails to catch that 16ms window, then even though user does some interaction with the screen underlying hardware fails to update the screen.

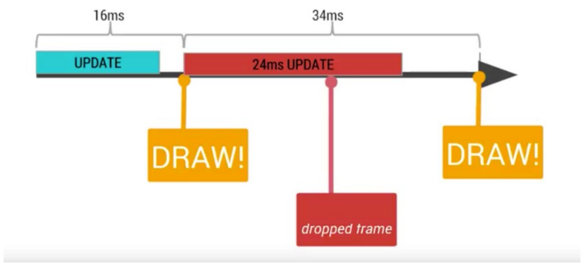


Fig. 4.3.: Dropped frame

Multiple such missing window can cause a noticeable lagging performance to the user. Hence any fancy animation or computation should be done efficiently to stay within 16ms limit. Typically, any display is updated at 60 HZ rate which means the screen is updated 60 times in 1000ms. If we divide 1000 by 60 then we get 16.66ms or 16ms per frame. So any frame that is not ready within 16ms is not received by the display hardware. These missed frames are also called dropped frames. Updating display requires both CPU and GPU hardware. CPU and GPU has specific roles in updating the screen.

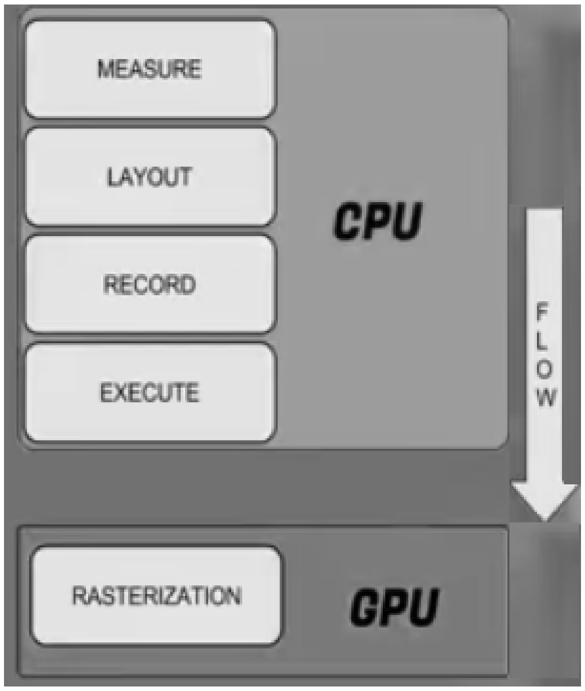


Fig. 4.4.: Android display update hierarchy

As we can see from figure 4.4 that CPU and GPU both works co-operatively to display the image in the screen. More specifically CPU works with layout creation, generation and view invalidation. To display a button, widget or any graphical object CPU must have to create a polygon or texture of that object then it is passed to the GPU by OpenGL ES API to rasterize. Rasterize is a method of drawing pixels in the screen from those polygons or textures. GPU specializes to do the rasterization work but still it is expensive because of two reasons. First of all, CPU needs to create textures of new objects before they can be rasterized and second those textures needs to be transferred or copied to the GPU memory which consumes precious CPU cycles. So in order to get premium performance it is desired that less copy to GPU memory space is done and if once copied to the GPU memory keep it on there as long as possible to gain the maximum throughput. Once a texture is copied to a GPU memory space that can later be referenced if that particular object needs to be re-drawn. Re-drawing instructions are fed by OpenGL API.

One of the core performance problem of an application is GPU overdraw. Drawing multiple pixels in a single frame causes overdraw. For example, in a deck of cards not all the cards are displayed instead all the cards are viewed partially except the last one. So if an app draws all the cards on top of another that wastes GPU and CPU processing time and thereby slows down the app. Android provides tools to analyse of debug overdraw scenarios. Each overdraw if represented by a color code for example if a portion of the display is red that means significant amount of overdraw has occurred in that pixel. On the other hand, if that pixel is blue that means that pixel is drawn only one time which is desirable.

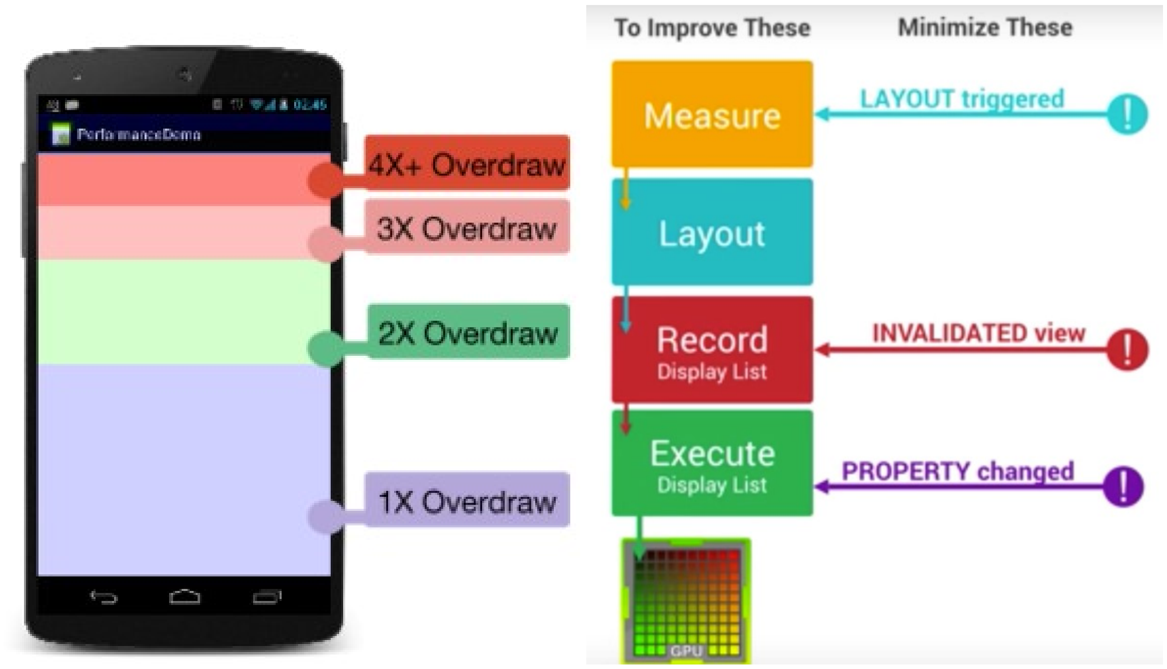


Fig. 4.5.: Overdraw indicators in android debugging

Android provides mechanism such as clipRect to reduce the overdraw problem. ClipRect feature allows developers to assign a section of a bitmap or image that will be drawn and will actually be visible in the final picture. Another performance issue related to display comes when too much dynamic change occurs while user interaction. For example, layout changes, button size or shape changes. Developers define their app layout using XML then android creates a display list out of the XML and creates corresponding OpenGL ES execution commands to draw the pixel in screen via GPU hardware. Android divides the whole process in four layers. Measure, layout, record and execute. Any high-level object such as buttons are first converted into a display list which is an internal data structure that contains all the information that is required to display the draw the image via GPU. That includes all the data that is fed to the GPU and execution commands to OpenGL to render the image on later time. If we want the button to be re-drawn on other part of the screen, then only the execute phase is necessary because the button shape is already cached on the GPU. However, if the button visual is changed for example, color then the display list might be invalidated and CPU cycles are needed to execute display list followed by creation. Moreover, if the scale of a layout object changes then it triggers the measure phase of the rendering hierarchy. In that case, all view will be traversed to evaluate possible configuration change. That in turn causes the layout phase to be called along with subsequent phases namely Record and Execute. In contrast, if only the layout objects are interchanged or moved then layout phase of the pipeline will be triggered. Following is the screen shot taken to visualize any overdraw in Mobile Deluge app using android GPU overdraw debug feature.

As figure 4.6 suggests there is no red spots in the screen which means no overdraw.In addition to overdraw complicated layout may slowdown app performance. For example, nesting multiple layouts can cause significant delay in case the layout is drawn dynamically in a list view or Grid view. Each time in the rendering pipeline Measure and Layout phase will be triggered. To solve these problem android provides hierarchy viewer which indicates processing delays in colour code. Red rep-

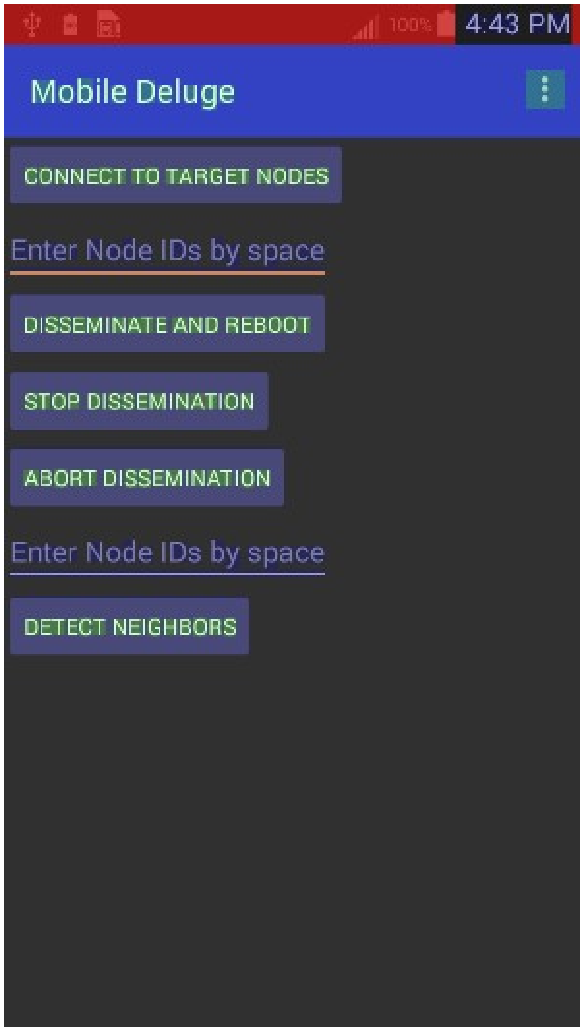


Fig. 4.6.: Mobile deluge overdraw evaluation

resents worst processing time, Yellow as moderate and Green is desirable. Following is the screenshot of Mobile Deluge app hierarchy viewer:

As we can see from figure 4.7 the main layout is a relative layout and most of the buttons and widgets are shown as green means they in an acceptable state

4.2 Application Algorithm Performance

Mobile Deluge app does use data structures like queue, linked lists and others to facilitate communication between mote hardware. The most important consideration

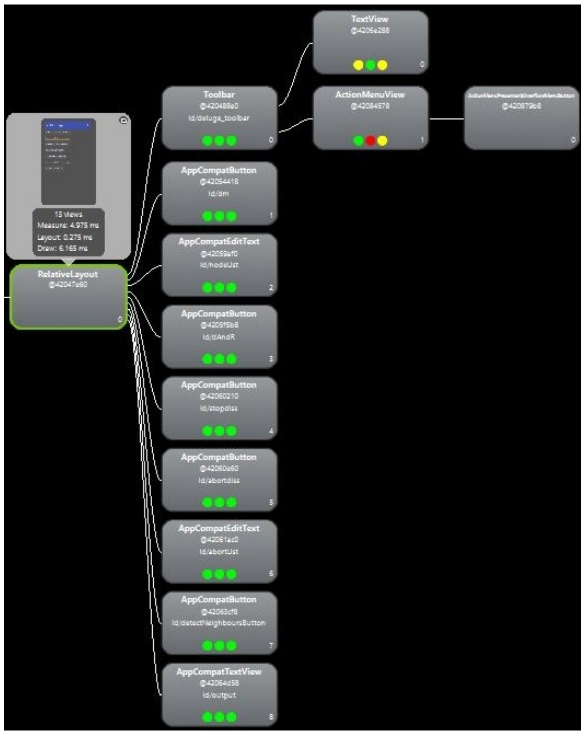
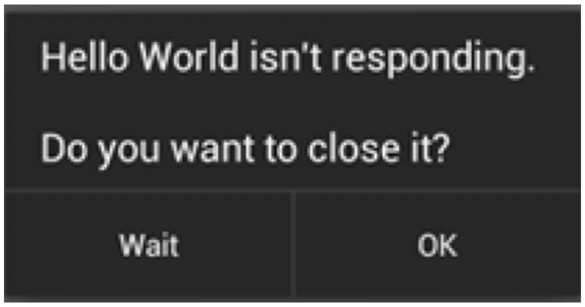


Fig. 4.7.: Mobile deluge hierarchy viewer

in android development is not blocking the UI thread. If that condition is not satisfied, then Application Not Responding (ANR) error message is displayed in the phone as shown in the figure which is not a good experience to the user. If user hits the Ok button, then the application process is killed by android or if the user presses wait then the application keeps running but remains in-responsive to the user if significant amount of work is being done in the UI thread. Hence, Android provides easy to use API for developers to run long running computation or blocking IO calls in a separate thread. Also there are various communication mechanisms present to communicate back to the UI thread the computation result or data received from a IO device. Mobile Deluge app uses android Handler and Message [21] to communicate back to the UI thread. The primary reason behind this communication to update Activity resources such as handing click events, taking input from user and showing results in the screen which can only be done using the UI thread. In our application, we have used handlers, broadcast receivers, services for efficient communication with the hardware and the user interface thread.



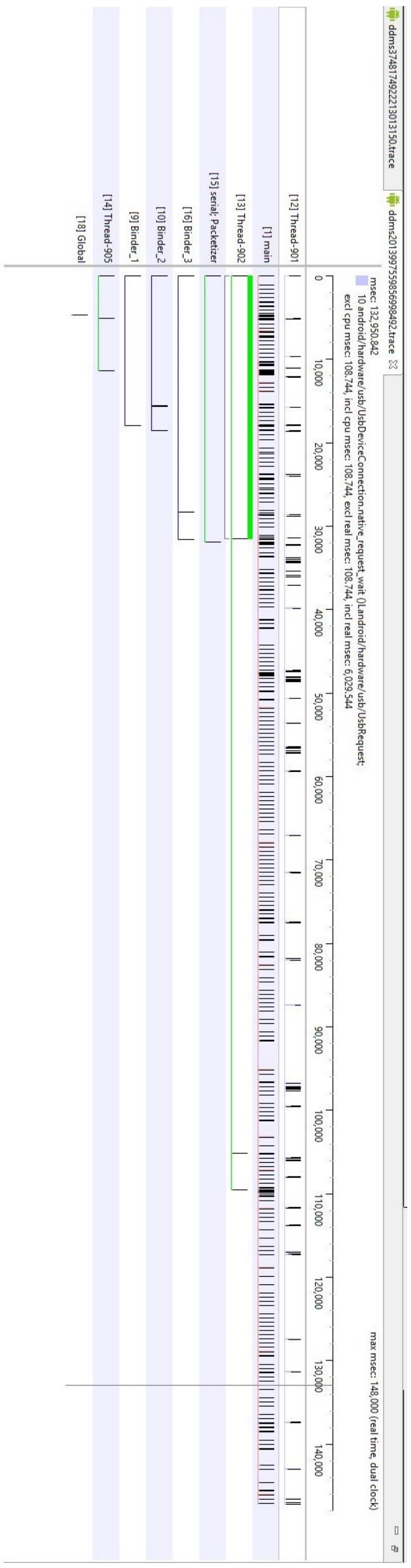
Hello World isn't responding.

Do you want to close it?

Fig. 4.8.: Application not responding error in Android

Android does provide some handy tools to analyze and debug ANR as well as misbehaving function. In our test, we have used android monitor to profile our application in runtime while reprogramming motes. In the test. three TelosB motes were used in which one of them was the base which was connected to the android device. The tests revealed some interesting information regarding which thread was running and whether out app was in good shape in using UI thread exclusively. Figure 4.9 describes performance parameters while our Mobile Deluge app was reprogramming two telosB motes. As the trace suggests, our UI thread is not blocked for significant amount of time due to computation or blocking IO to USB hardware connected. More specifically, it also shows the name and number of threads running in the app.

Application performance also depends on how developers allocate and manage memories. Developers can allocate and de-allocate memory manually. In that case, memory management is solely the responsibility of the developer. But that process sometimes become complex or if programmer fails to de-allocate memory appropriately memory leak will occur. Memory leak implies memory space that cannot be used by other processes because that was not freed properly. If a program allocates a lot of memory but never releases that memory space the whole system may become slower and app will crash because of low memory. To solve the problem automatic memory



msec: 132,950.842
10 android/hardware/usb/UsbDeviceConnection.native_request_wait ()Landroid/hardware/usb/UsbRequest;
excl cpu msec: 108.744, incl cpu msec: 108.744, excl real msec: 108.744, incl real msec: 6,029.544

Fig. 4.9.: App trace of test run of mobile deluge application

management was invented in which case memory management is done by programming language on behalf of the programmer. In Java Garbage Collector (GC) is used to free up memory that was previously used by other application and no reference of those memory spaces are held. Even though the idea is simple behind GC but it comes with its own disadvantages. First, memory leak can still occur if programmers keep reference to objects even after the program is done execution. Secondly, GC events requires CPU processing time. Too many GC event may cause a system to slow down. More specifically, GC is done by the Main or UI thread in Android therefore, too many GC event will slow down user experience with the app. One probable cause of too many GC events in memory churn. Memory churn means allocating too much memory in a shorter period. For example, creating objects repeatedly in a loop will cause memory churn to occur. To detect and debug all those problems, Android studio provides a tool called Memory Monitor. Memory monitor provides a real-time view of the device memory and CPU status. The tool also provides a way to cause random GC events to see if any application is leaking any memory. Leaky memory can be identified by causing a GC and monitoring allocated and freed memory. If allocated memory keeps increasing and GC does not claim the memory that was allocated it might indicate that they may be a memory leak in the app implementation. Memory churn on the other hand, caused large spikes in the memory monitor since a large memory is suddenly allocated and GC events tries to claim the memory. Memory churn can also be detected by another tool called allocation tracker provided by Android Studio. Allocation tracker helps developers detect in which method a large allocation is done and which thread is doing the allocation.

As we can see from figure 4.10 a large chuck of memory is being allocated in a short period and android to cope up with the apps memory request allocates a higher memory to the app. GC events are very frequent because large amount is memory is de-allocated and allocated rapidly. However, Mobile Deluge app provides a steady state memory graph while re-programming which indicates no memory churn.

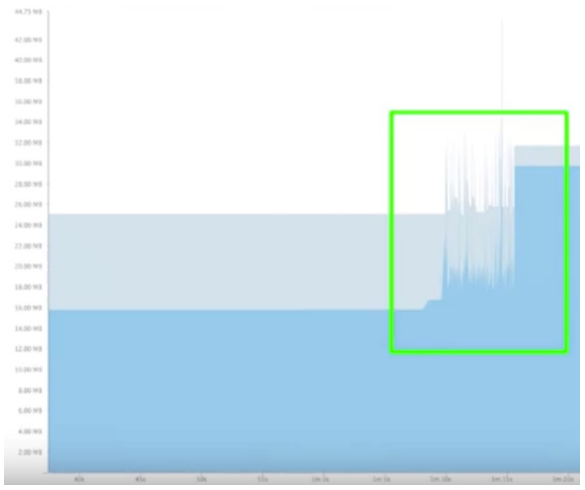


Fig. 4.10.: Memory churn event in memory monitor

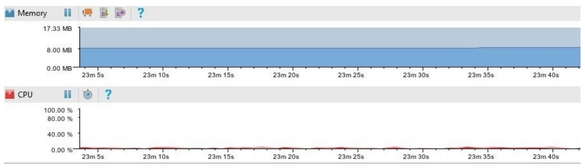


Fig. 4.11.: Memory monitor status of Mobile Deluge app while re-programming

As the figure 4.11 suggests there are very little CPU usage and memory monitor does not show any memory churn or frequent GC events.

4.3 Battery Performance

Battery consumption is critical in mobile devices since the power is limited in smaller devices. In our case, it is even more important since the mote is powered through the android phone. So, in order the measure the power consumed by our application we have used an application Power Tutor published in android play store and measured the CPU usage in Milli Watts(mw). We have also compared our app power usage with popular android apps like Google Chrome and Youtube. In the test run, we have conducted some over the air reprogramming with our application and interacted with 3 motes. We have run several commands for about 10 minutes and recorded the amount of power consumed by the application. Same was done for the other apps. For Youtube app video playback was done for 10 minutes and for chrome browsing was done for the same amount of time. Table 4.1 illustrates the power consumed in each of the cases described above. As we can see from the table MD application uses less power than popular application that will enable us to run the application in the test bed in remote areas where no wall power is available. The power consumed is measured only by CPU cycles. The device also draws power from the phone but that was not taken into consideration while doing the test.

Application	**Power Consumption(mW)**
Mobile Deluge	130
YouTube	160
Chrome	157

Table 4.1.: Sample power consumption by application

5. CONCLUSION

Wireless Sensor Network(WSN) deployments require regular maintenance and software upgrade. Sensor nodes in real world deployment are often deployed in harsh environments which makes it unfeasible to do re-programming each manually. So re-programming over the air is essential in large deployment. Our Android application provides an easy and efficient way to re-program the WSN nodes in real world deployments. Our implementation provides a extensible design to further improve the application by adding more features to current re-programming application and to even add a completely new application like 802.15.4 sniffer. Our current implementation of the re-programming application can be further improved by integrating a database with the application to keep track of the nodes which are being re-programmed. Also, the database can be synchronized with the on line database by using the mobile data networks. Another important feature is secure communication over the wireless medium. Currently all the data are transmitted unencrypted and is vulnerable to various network attacks. Nevertheless, current implementation of our application is found stable in lab testing as well as in real world testbed.

REFERENCES

REFERENCES

[1] Wikipedia. Ieee 802.15.4, Jul 2017.

[2] I. Demirkol, C. Ersoy, and F. Alagoz. Mac protocols for wireless sensor networks: a survey. *IEEE Communications Magazine*, 44(4):115–121, April 2006.

[3] Open Handset Alliance. Industry leaders announce open platform for mobile devices, Nov 2007.

[4] eLinixWiki. Android kernel versions, Jul 2011.

[5] Mark Wilson. T-mobile g1: Full details of the htc dream android phone, Sep 2008.

[6] Patrick Brady. Anatomy & physiology of an android, May 2008.

[7] Wikipedia. Model-view-controller, Sep 2017.

[8] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. *Proceedings of the 2nd international conference on Embedded networked sensor systems - SenSys 04*, 2004.

[9] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[10] Xiaoyang Zhong, Miguel Navarro, German Villalba, Xu Liang, and Yao Liang. Mobiledeluge: Mobile code dissemination for wireless sensor networks. In *Proceedings of the 2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems*, MASS '14, pages 363–370, Washington, DC, USA, 2014. IEEE Computer Society.

[11] Sangwon Hyun, Peng Ning, An Liu, and Wenliang Du. Seluge: Secure and dos-resistant code dissemination in wireless sensor networks. *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, 2008.

[12] Rui Zhang and Yanchao Zhang. Lr-seluge: Loss-resilient and secure code dissemination in wireless sensor networks. *2011 31st International Conference on Distributed Computing Systems*, 2011.

[13] Sangwon Hyun, Kun Sun, and Peng Ning. Fec-seluge: Efficient, reliable, and secure large data dissemination using erasure codes. *Computer Communications*, 104:191203, 2017.

[14] TinyOS Wiki. Deluge t2, Nov 2010.

[15] Felipe Herranz. Usb serial controller for android, Nov 2016.

[16] Google Inc. Usb host, Aug 2017.

[17] Ben Greenstein Levis and Philip.

[18] W. Simpson. Ppp in hdlc-like framing. STD 51, RFC Editor, July 1994.

[19] Philip Levis.

[20] Wikipedia. nesc, Jun 2017.

[21] Google Inc. Handler, Aug 2017.