1 Specification

This section contains the technical specifications of the modules and how we have designed them throughout the iterations.

1.1 Common Technologies

1.1.1 Google Cloud Messaging

1.1.1.1 Design Overview For our system we intend to utilize push notifications as the main way of initialising contact between our server and the users phone. As our system is initially being developed for the Android platform, we are going to implement the Google Cloud Messaging (GCM) [10] system as the main method of sending and receiving these notifications.

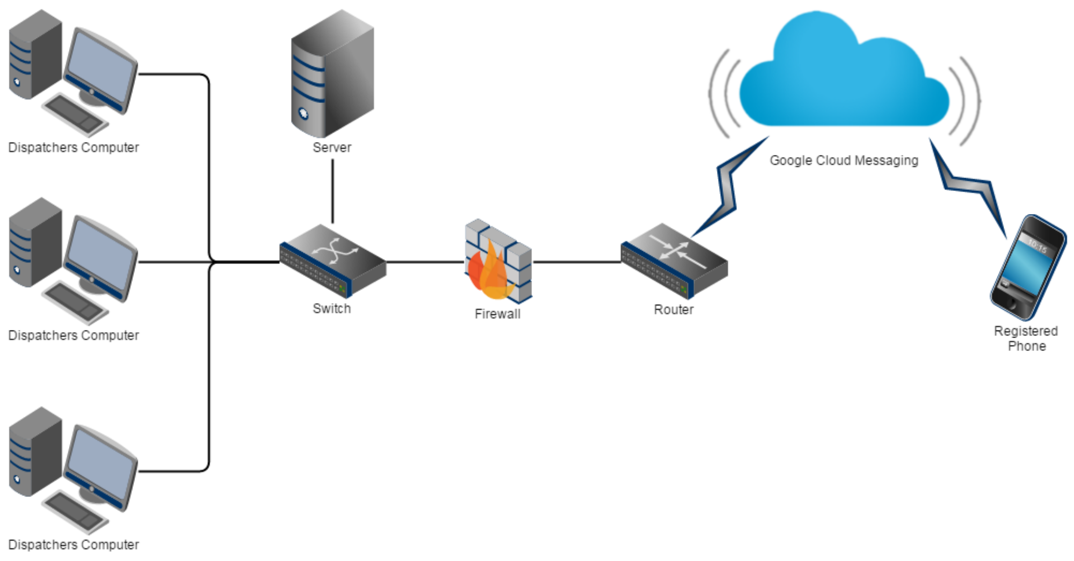


Figure 1: Network map of how our system will communicate between devices and the server

We have chosen to use this system for a few reasons. Firstly as our app is initially being developed for Android (due to current limitations with IOS) so using GCM is advisable as it is heavily embedded with the operating system and so reduces strain (such as power usage) [11]. GCM also allows the notifications to be received when the app is not running which is of huge benefit to the user who then does not have to keep the app open to offer their help. Other notable reasons include that it is free to use, can handle large scale push notifications [11], notifications can be canceled at a later time using a collapse key and that you can send data (up to 4096 bytes) as a payload to be used by the app. GCM can also be integrated with ISO push notifications [13] which could be useful if, in the future, an IOS app was being developed.

As a result of deciding to use GCM, we need to keep a database of the registered app users. This is because GCM uses a registration ID system to send a message to a specific phone, we need to store and relate this ID to a user of our system.

1.1.1.2 GCM Setup We intend to use GCM primarily for its ability to push notifications to a clients device. There are several things which we need to have set up before the notifications can be sent. Firstly, each phone will have a unique key associated with it which is used to identify an individual's phone so messages can be sent to it. This is obtained by the app when it registers with the GCM service. In order for our server to send messages to this device we need to know and store this registration ID, therefore the device will need to send it to our server to be stored in a secure database. The detailed setup instructions are available directly from google and have example code to assist in the setup [14].

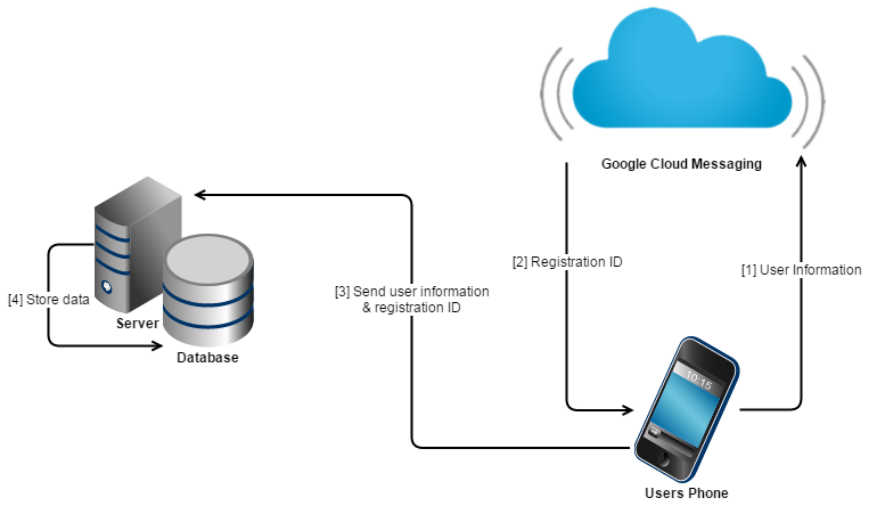


Figure 2: Diagram of the initial setup and registration with Google Cloud Messaging

We will also have to setup the database to store the information of the user that the phone is registered to along with the Registration ID (token) obtained from GCM. This design is detailed in the database section.

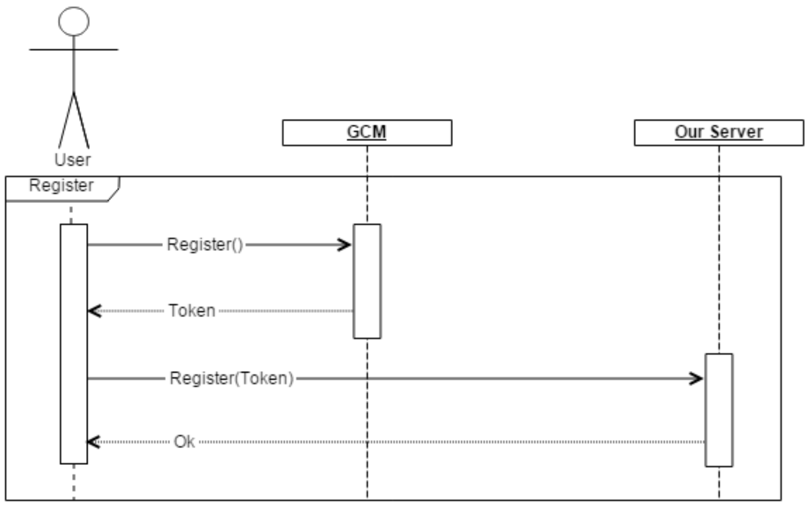


Figure 3: Sequence diagram showing interactions when registering a device

1.1.1.3 Sending Downstream Messages Once the initial setup is completed we can focus more on the message sending and receiving parts of the application. In order to push a notification to our users phone our server needs to send a request containing the registration ID and the message to Googles Messaging Server which then sends it directly to the device associated with that ID.

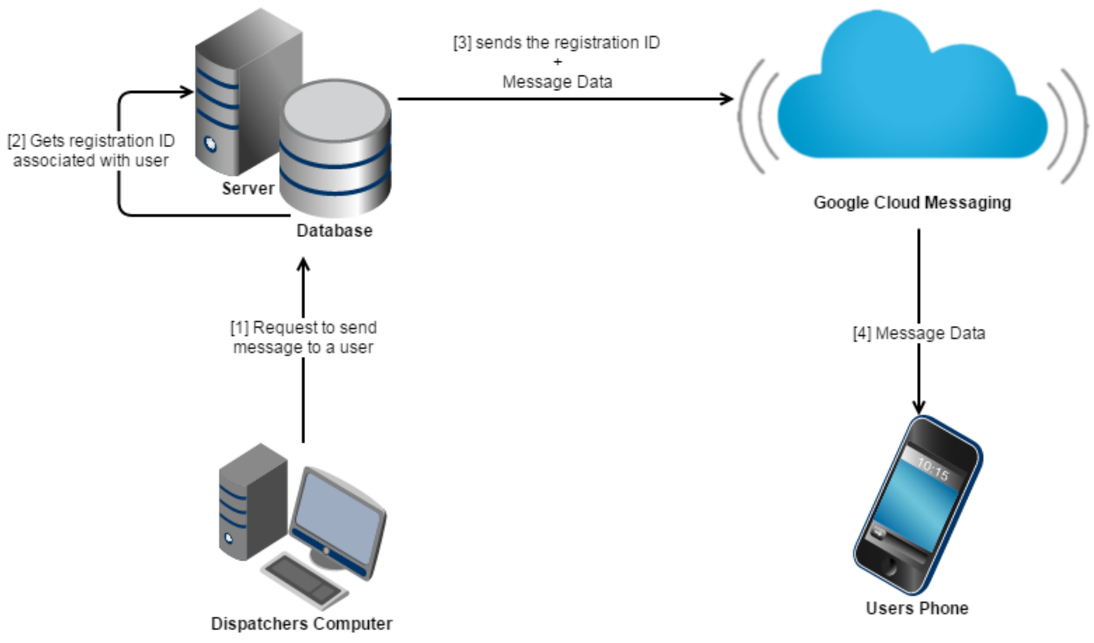

[3] sends the registration ID
+
Message Data
[2] Gets registration ID associated with user
Server
Database
Google Cloud Messaging
[1] Request to send message to a user
[4] Message Data
10:15
Dispatchers Computer
Users Phone

Figure 4: Diagram illustrating sending of a message to a clients device

We have decided to use the XMPP protocol as the connection between our server and GCMs server. This was chosen (over HTTP) for a few reasons. Firstly, the XMPP protocol is faster than HTTP and so would allow the system to send messages to users quickly which is a crucial aspect of any emergency system. It also uses the existing GCM connection on the device to receive the data, saving battery useage as you don't have to open your own connection to the server. XMPP does not however allow broadcast messages to be sent to multiple devices, instead you have to send a new message for each device you wish to contact. This should not present a problem in our system as we will only need to send a message to one device (in this iteration) and even if a later iteration was to require sending to multiple devices, it would be a small number.

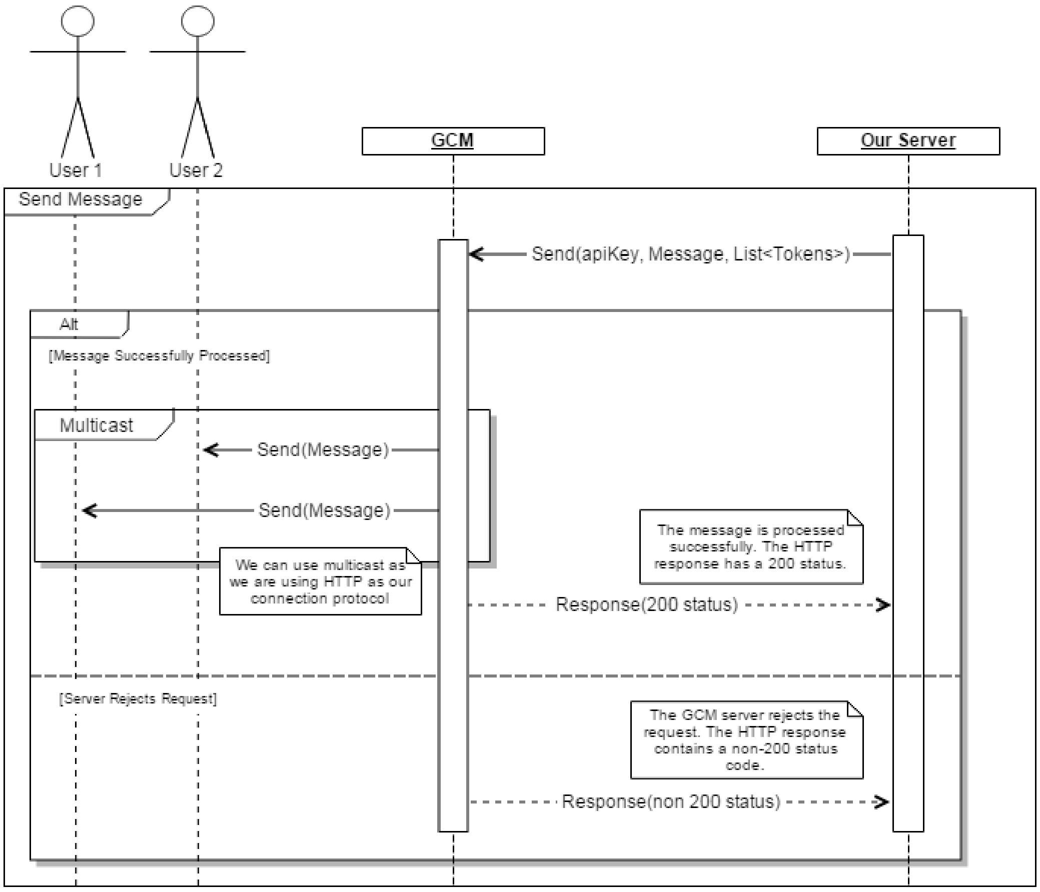

User 1
User 2
GCM
Our Server
Send Message
Send(apiKey, Message, List<Tokens>)
Alt
[Message Successfully Processed]
Multicast
Send(Message)
Send(Message)
We can use multicast as we are using HTTP as our connection protocol
The message is processed successfully. The HTTP response has a 200 status.
Response(200 status)
[Server Rejects Request]
The GCM server rejects the request. The HTTP response contains a non-200 status code.
Response(non 200 status)

Figure 5: Sequence diagram showing the interactions when sending a message to the client device

Once a XMPP connection is established our server can use normal XMPP ¡message¿ stanzas to send a JSON-encoded message. For this iteration our message will contain five components, the registration ID of the device we are sending the message to, a message ID to uniquely identify this message, a collapse key we can use to overwrite the message later, a time-to-live after which the message with expire and not be sent, and the payload data we wish to send. The format of the message will look like this :

```
<message id="">
  <gcm xmlns="google:mobile:data">
  {
      "to":"REGISTRATION_ID",
      "message_id":"UNIQUE_MESSAGE_ID",
      "collapse_key":"UNIQUE_COLLAPSE_KEY"
      "time_to_live":"TIME_IN_SECONDS",
      "data":
      {
          "KEY":"VALUE",
    }
  }
  </gcm>
</message>
```

For each device message your app server receives from GCM, it needs to send an ACK message. If you don't send an ACK for a message, GCM will just resend it. GCM also sends an ACK or NACK for each message sent to the server. If you do not receive either, it means that the TCP connection was closed in the middle of the operation and your server needs to resend the messages[10].

These response messages will need to be dealt with according to their responses. The ACK will be sent if the message was successfully delivered however there are two types of error messages (NACK and Stanza error) for situations such as Invalid JSON, bad registration ID or Device Message Rate Exceeded. Each of these messages contains an error ID and message which informs you of what the problem is so that it can be dealt with [15].

In some situations, we will include a payload of data that will be sent (in the message) to the clients device. This payload is easily incorporated into the message and forms part of the JSON-encoded message string. This payload data has no limit to the number of key-value pairs however there is a total limit of 4kb maximum message size. String values are recommended to all other data types would need to be converted to strings before sending them to the device [15].

1.1.1.4 Sending Upstream Messages We can also use GCM to send upstream messages from the clients device to the server via Googles CCM (Cloud Connection Service). This works by embedding the servers registration ID (obtained when registering for the GCM service) into the clients software and passing this ID along with a message to the GCM service to be forwarded to the server.

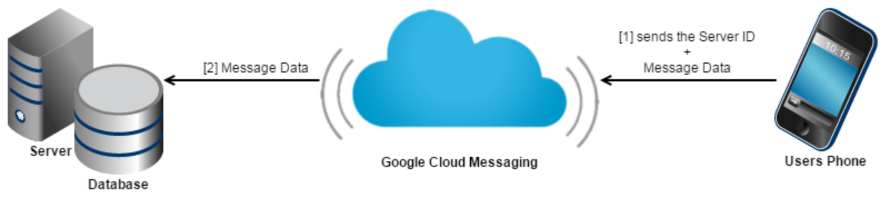


Figure 6: Diagram showing the sending of a message from the client device to the server

This will once again use the XMPP protocol for the reasons stated in Sending Downstream Messages and then we can use normal XMPP ¡message¿ stanzas to send a JSON-encoded message in the following format[25] :

```
<message id="">
  <gcm xmlns="google:mobile:data">
  {
      "category":"com.example.yourapp", // to know which app sent it
      "data":
      {
          "KEY":"VALUE"
      },
      "message_id":"UNIQUE_MESSAGE_ID",
      "from":"DEVICE_REGISTRATION_ID"
  }
  </gcm>
</message>
```

This message is then forwarded by GCM to our server and the message is parsed for its data. The device uses the send() method from the GCM API to construct and send the message. This takes the following format:

```
gcm.send(GCM_SENDER_ID + "@gcm.googleapis.com", id, ttl, data);
```

Where `GCM_SENDER_ID` is the stored GCM ID number of the server, id is the unique message ID, ttl is the time to live of the message (seconds) after which the message will expire and not be sent and finally the data to be sent [25]. This will produce a stanza in the format described above and send it to the server ID specified.

After the message is received from CSS by our server we are expected to send an ACK message back to google. This should be in the following format [25]:

```
<message id="">
  <gcm xmlns="google:mobile:data">
  {
      "to":"DEVICE_REGISTRATION_ID",
      "message_id":"MESSAGE_ID_OF_RECIEVED_MESSAGE"
      "message_type":"ack"
  }
  </gcm>
</message>
```

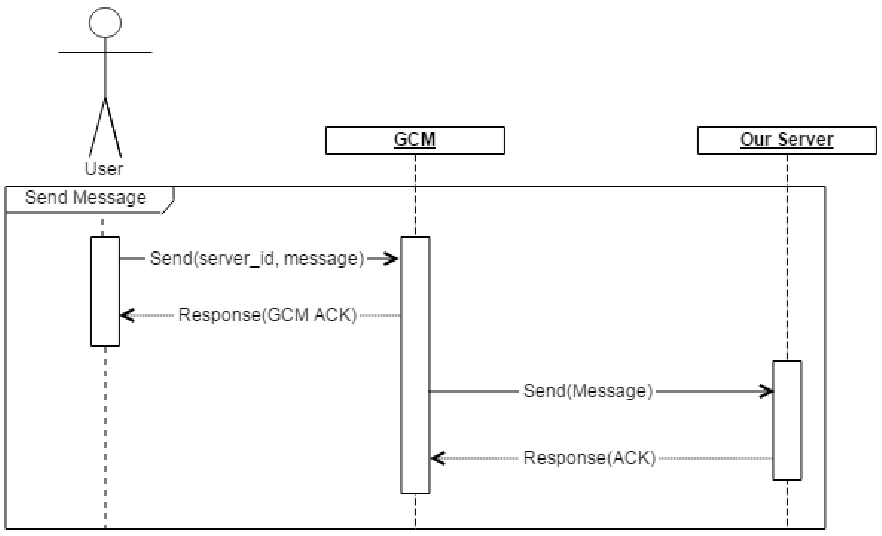


Figure 7: Sequence diagram showing the interactions when sending a message to the server from a client device

1.1.2 Database

It was immediately apparent that both the Emergency App and CPR System would need a database backend. To ensure we chose the best system, we investigated the three most prolific database management systems (DBMSs): MySQL, Oracle, and SQL Server.

Almost immediately, we discounted MySQL as being immature compared to Oracle and SQL Server. Transactions and stored procedures are a relatively recent addition to the system, and both are technologies that we will rely on heavily to ensure data is consistent and intact. It also has far fewer features relevant in enterprise environments, lacking fine-grained access control, table partitioning and disaster recovery.

Next, we scrutinised the remaining contenders. Ultimately, there isnt a lot of difference between them; SQL Server has a marginal advantage with in-memory tables and better performance in write-intensive environments, while Oracle is supposedly more suited to applications with equal read and write traffic due to its locking system.

In the end, we decided to go for SQL Server due to its Spatial Data extensions, which would be used particularly heavily in the CPR System. This functionality allows us to do accurate geometric calculations, like finding the distance between two sets of coordinates, within the database itself. In addition, SQL Server is renowned for its stability, with a transaction and journaling system that detects and automatically fixes errors, preventing corruption and increasing uptime - all of which are desirable traits in such life-critical applications.

1.2 CPR System

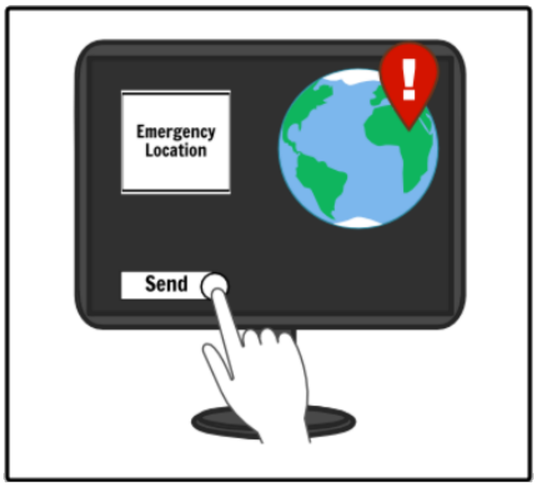
1.2.1 Iteration 1 - Basic Functionality

1.2.1.1 Aims This iteration develops the core functionality of the CPR system; it is intended to be the simplest application possible that has the required functionality. Subsequent iterations will refine and build on top of its services and components.

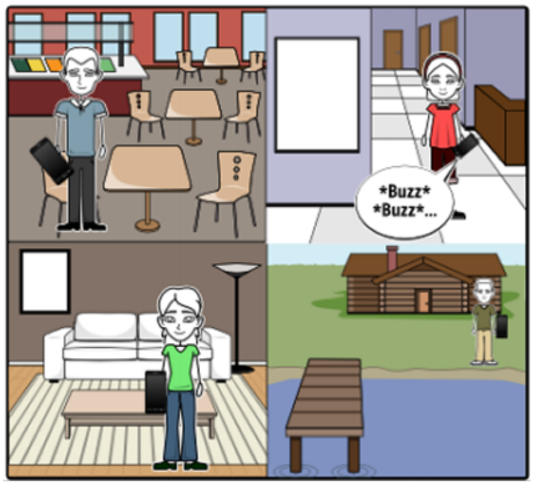
At its most basic level, the system needs to be able to send a message to a responders devices to notify them of the location of an emergency. This individual should be the closest person in the database to the incident. They are then expected to attend the emergency location and provide assistance where necessary. The individual is assumed to have the appropriate qualifications and training which has been verified, and have willingly signed up for the service.

1.2.1.2 Storyboard This is a storyboard to help visualise the intended functionality of the system.

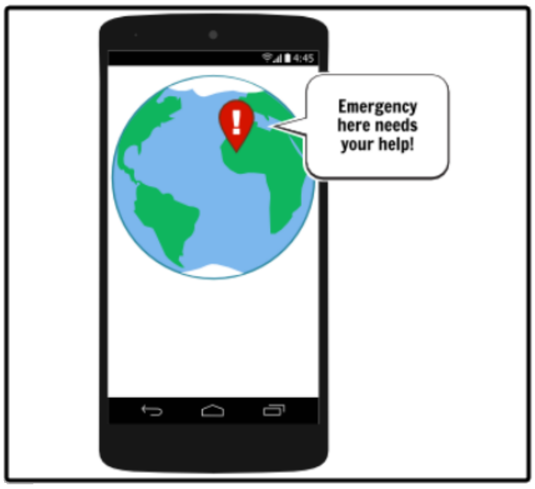
Emergency Situation!
The emergency Dispatcher gets a call about an emergency situation.



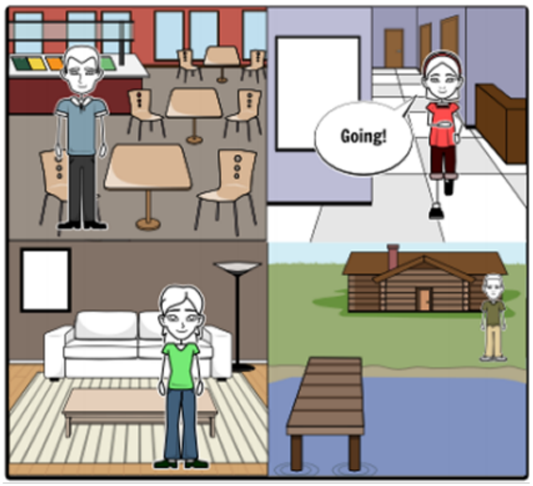
It is an appropriate emergency to ask for help from anyone nearby. The dispatcher sends the help request.



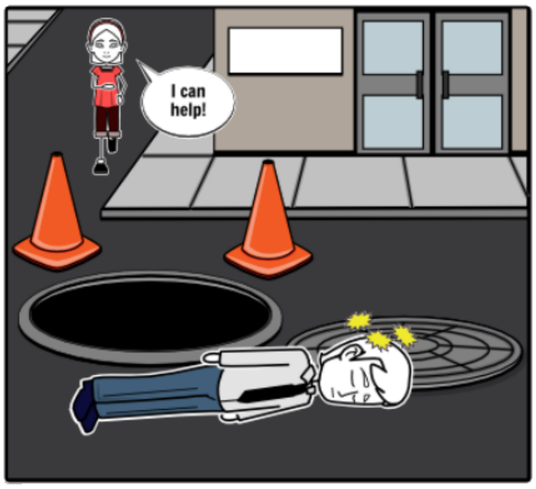
The closest person to the emergency is alerted to the situation.



They are shown the location.



If the responder is available to help they make their way to the emergency.



Once they arrive they help as appropriate.

1. The emergency services receive a call detailing a medical emergency which is triaged.
2. While waiting for a response vehicle to arrive, the dispatcher has the option to send the emergency location to a trained responder (first aid or CPR) who may be able to attend quicker than the emergency services. This option to request this responders assistance should be limited to certain situations and locations so that responders are not sent to potentially dangerous environments.
3. Once the request has been sent, the responder is notified that an emergency requires their presence. The notification shows the responder the location of the incident
4. The responder then decides if they are able to attend the emergency to help the situation

and makes their own way there.

In theory, the responder gets to the location quicker than an emergency response and they are able to provide first aid to those who need it until the ambulance arrives.

1.2.1.3 Sending Notifications In order for responders to attend an emergency, they need to know where it is. The standard way of doing this is to send a latitude and longitude, which we will include in the payload section of a GCM message. This information can be directly passed to all mapping applications to plot the location. The structure of the message data should be as follows:

```
...
"data":
     {
         "message_type":"new_emergency",
         "longitude":"VALUE",
         "latitude":"VALUE"
     }
...
```

In this message the `"message_type"` field describes what kind of message this is. To send a new emergency to the user the value of this field should be `"new_emergency"`. This is checked when the message is received and the appropriate action is taken (i.e. a new emergency alert is created). The alternative is `"close_emergency"` as the value and this is used when the dispatcher wishes to cancel the assistance of the user either because they are no longer required or the casualty has been dealt with and the emergency case closed (see App CPR pane for more details).

The other data we need to send is a coordinate pair corresponding to the location of the emergency, (where longitude and latitude values are in the range of [-180, 180] and [-90, 90] respectively [16]) as this is a standard way of referencing a location and is the coordinate system major map applications use. This will allow us to parse the coordinates directly into a map app on the device to show the user where to go. This can be easily done in two ways, one is opening the default maps app with the location entered into a geo-URI [17] and opening it. This URI contains the coordinates and when called opens the default map with the location shown as a drop point (Example: `"geo:0,0?q=34.99,-106.61(Emergency)"`). The second option is to incorporate the Maps API into our app, this would allows us a greater degree of control on what the user can see and also disable features (such as removing the emergency marker). The implementation of this is relatively easy and adding a marker simply uses the `LatLng` data type [18].

```
public void onMapReady(GoogleMap map) {
    map.addMarker(new MarkerOptions()
        .position(new LatLng(10, 10))
        .title("Emergency"));
}
```

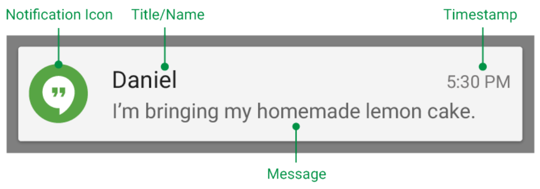
Integrating a map into our app is the route we have decided to go down as this allows us greater granularity and control over what the user sees and is able to do. This should be shown to the user when they click on an emergency notification.

1.2.1.4 Notification Interaction and Alert

When an emergency is sent to the phone the user will need to be notified immediately so that they can attend the scene as quickly as possible. The first step towards doing this is to create a popup notification so that the user can immediately see (either on top of other applications or on the lock screen) that their assistance is required.

There are two main factors of information that the user needs to be informed of, the first is that their assistance is required, the presence of this notification does imply this however it should also be stated somewhere in the notification that this is an emergency situation. The second is the location of the emergency, for this we have identified two main points of information that are useful to the user. The first is the address of the emergency location, this should be present so that if the user knows the area around them they can instantly recognise where they are needed and start making their way towards it. It also allows them to give directions, say to a taxi driver, directly from the notification without having to open the app at all. The second is the distance to the emergency that the user is currently This information is very useful to the user as it allows them to instantly gauge how long they will take to attend the scene and make decisions based off this such as the best mode of transport.

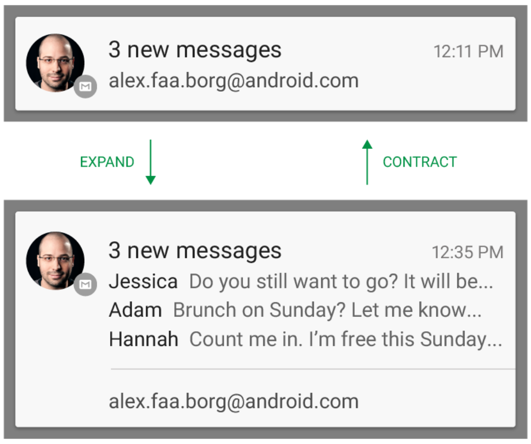
To provide both of these points to the user we will need to use Androids built-in location API, as the information provided to the device is in coordinate form. There are several features which will allow us to obtain the required information, the first is in the Geocoder class of the location API [19]. The method `getFromLocation()` rmation, the first is in the Geocoder class of the location API [19]. The method getFromLocation()takes a coordinate pair (latitude and longitude) and returns a list of addresses which are known to describe the area immediately surrounding the coordinates. Upon calling this we can then extract the address line by line using the associated methods with the `Address` class [20] and insert them into our notification. We can also use the `Location` class of the location API to get the `distanceTo()` a destination [21]. This takes a location as a parameter (which we can set the longitude and latitude of) and returns the distance in meters to the destination from the users current location.



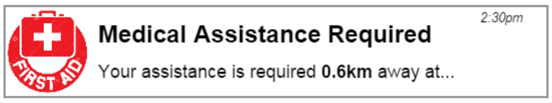
Android supports a variety of notification formats [22] however they all have four common features, a Title/Name, Notification Icon, Timestamp and a Message. Our notification should also have these features, the title should be a common one (for all emergency notifications) which relates to the fact that the users assistance is required at an emergency. A title similar to Medical Assistance Required should both inform the user of what the notification is for and grab their attention.

The notification icon should also be striking to grab their attention, and also needs to be unique to the app and tell the user, at a glance, the nature of the notification. Something similar to this image would be best as the image is striking and so should catch the users attention while being unique among other notification icons and being related to the theme of the notification.

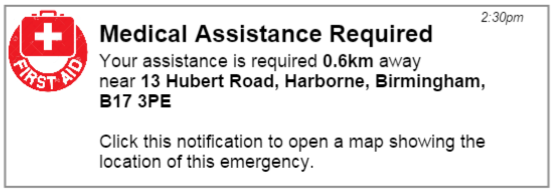
The timestamp of the notification should be set to the time that the push notification was received and the message body should contain both the distance to and the address of the emergency. Android allows expandable layouts [22] which allows the notification to be minimal initially and expand when clicked on. The minimal layout still contains both the title and the icon (along with a line of text) which means the user can still instantly see what the notification is about and can click on it to expand the description (which contains the address).


3 new messages 12:11 PM
alex.faa.borg@android.com

3 new messages 12:35 PM
Jessica Do you still want to go? It will be...
Adam Brunch on Sunday? Let me know...
Hannah Count me in. I'm free this Sunday...
alex.faa.borg@android.com


2:30pm
Medical Assistance Required
Your assistance is required **0.6km** away at...

Minimal notification


2:30pm
Medical Assistance Required
Your assistance is required **0.6km** away near **13 Hubert Road, Harborne, Birmingham, B17 3PE**

Click this notification to open a map showing the location of this emergency.

Maximized notification

This is a design shown how our notification might look to the user on either the lock screen or while actively using the phone. Upon clicking this notification the app will be launched and show a CPR pane which would not be normally accessible. This pane contains a map with the location of the emergency shown clearly with a marker, the users current location and a directions button which will plot a route to the emergency from the users current location.

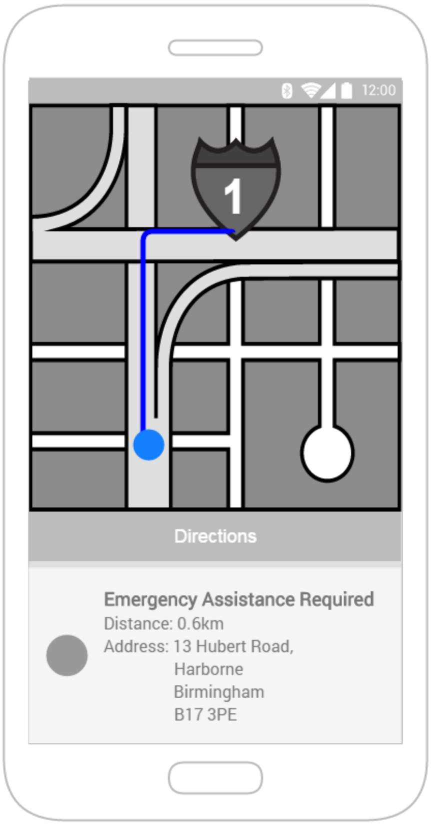
1.2.1.5 Cancel Notification The dispatchers should also be able to cancel the assistance request remotely for a variety of reasons such as the emergency has been dealt with or paramedics have already arrived to name a few. To do this we will utilise the collapsible feature of messages sent over GCM [23] which allows us to overwrite previous messages with the same `collapse_key`. We can combine this with the field in the payload of the message `"message_type"` which is set to `"new_emergency"` when the dispatcher wishes to ask for assistance. The new message should have a `"message_type":"close_emergency"` so that, when the message is read the message type should be checked and the appropriate action taken, in this case it should, if its currently accessible, hide the CPR pane and clear any associated data. A message field should also be present which will be displayed in the message field of the notification.

```
...
"data":
      {
          "message_type":"close_emergency",
          "message":"Emergency resolved"
      }
...
```

1.2.1.6 Notification Sound of the App This was the first thing which we thought the user would like to customize as this has been commonplace among phones for years and our users would expect to be able to do this with our app. However, after some discussion we thought about how this might negatively affect the users ability to be alerted to incoming emergencies. For example, if a user was allowed to set their own notification sounds and they chose one that was the same (or similar) to their message alert or ringtone then they might be more inclined to ignore the phone for a period of time believing it was just a regular notification. This is more likely as our app is likely to be only used occasionally when the individual is the closest responder to an emergency, it will be a case of setup and forget for most users who might even have forgotten they installed it until they are being asked to help.

For this reason we have decided against allowing the user to change the notification sounds. We intend to use a custom alert noise which is unlike any other provided by default on the phone. We believe that keeping this sound is more beneficial than allowing the user more personalisation as they will not recognise the sound and therefore are more likely to check why their phone is making a noise.

1.2.1.7 CPR App Pane This is the design prototype for the CPR pane of the application. By default (i.e if its opened without an emergency) this shows a blank map and information pane which, when an emergency notification is received from the dispatchers, changes to display the information for that emergency. Upon receiving the notification (with the payload data) the coordinates are sent to this app and become the destination marker of the map.



The associated information is displayed at the bottom third of the pane which can be obtained (as in the Notification Interaction and Alert section) from coordinates provided using the location API. The distance measurement should be re-calculated at a set interval so that the value is updated while moving.

The map also contains a Directions button which, when pressed, will start navigation assistance from the users current location to the emergency.

We have decided to implement an in-app map view rather than opening the directions in the default map app because it gives us a greater degree of control over what we can show the user and allow them to interact with. For example it allows us to disable the draggable state of the marker [18] so that the user cannot accidentally move it to the wrong location and then delay their response time. It also allows us to display our own content on the screen (in the lower third) at the same time that the user is getting directions to the emergency. This is currently displaying the address and distance of the emergency from the user however it could be used to display extra information as appropriate.

There are two ways for this information to be hidden once again. The first is that the phone receives a new notification from the dispatchers which contains the same `collapse_key` as this emergency (detailed in Cancel Notification) in this case it should clear any associated data from the app (the destination marker and info pane). The second way is when the app is first loaded a timeout timer should be started which contains a reasonable amount of time to allow the user to attend the scene, but once expired should hide this pane from view and clear any previous emergency data. A timer of 60 minutes should be enough time for any first responder to attend a scene and pass the casualty over to the emergency services (this time should in fact be more than enough). This is so that if, for some reason, the collapse message is not received the information will be removed on its own accord

1.2.1.8 GPS Location We need to maintain a database of registered users locations so that we can target an individual who is the closest person to the emergency and send a notification directly to them via GCM. There are a number of ways which we could implement this, however we decided to (for this iteration) push the location from the device to our servers on a regular time interval. A time period of 1 hour is what we will use initially, this should be enough to maintain a reasonably accurate location of the user (more accurate when they are stationary for a while and less accurate while traveling) while not heavily impacting battery useage or performance of the device with more frequent updates. While not being a very complex or clever system this will be enough for us to get started and will be sufficient for this iteration.

An alternative implementation would have been to push the location of all emergencies to all phones registered on the service. It would then be upto the phone to determine if they were close enough to alert the user to the emergency. We decided against this method for a few reasons, firstly it does not scale well with a large user base. If this was to be rolled out nationally you could easily be sending hundreds of emergencies to thousands of phones on a daily basis. This would also drastically affect battery life of the users device as it would then have to process each emergency received to determine if it was near them, possibly leading to increased user dissatisfaction and uninstallations of the app. Finally, there would be a lot more data being sent and received in this implementation. Rather than sending each clients location once every 1 hour (for example) you would have to send each emergency to every registered phone which would quickly rack up with multiple emergencies per day.

Other ideas we thought about were having the user subscribe to an area that they are likely to be in (possibly one for work and home) then using this location to send messages to people who are likely to be in the area. This is currently how the Swedish project Project SMS-livrddare [6] is contacting its users. We feel however, that this is not accurate enough as there will be situations when they are not around the area (e.g traveling to another city) and using the GPS location of the phone would allow us to contact users with greater geographically accuracy.

To send this data to our server we decided to use GCMs upstream message feature to send the message via Googles CCM (Cloud Connection Service). We chose this over establishing our own connection for a few reasons but primarily for increased efficiency. GCM upstream messages are sent over the same connection used for receiving, which is managed by the operating system and is left in an always open state. It makes sense to utilize this connection rather than use excess resources which would decrease performance (such as battery) [26]. It is also very easily implemented as the API provided does most of the work that would have to be managed if we were to implement our own connection (e.g. checking if the network is available) [27]. It is also not necessary to establish our own connection as the only data we will be sending to the server (in this iteration) is a GPS coordinate pair for the users location, this can be easily converted into string format and sent as a payload GCM message.

The message should be sent according to the specification detailed in the GCM justification section under Sending Upstream Messages. This details that the sent message will have the users registration ID attached which the server can use to associate this sent location with a registered user. The structure of the message data should be as follows:

...

```
"data":
      {
          "message_type":"location_update",
          "longitude":"VALUE",
          "latitude":"VALUE"
      }
...
```

This message, when received by our server, will then be able to parse the message type to see that it is a new location that's being sent and add it to the database appropriately. The longitude and latitude values are the devices current location and can be obtained by the getLastLocaiton() method of the Location API [28]. Android's Network Location Provider determines user location using cell tower and Wi-Fi signals, providing location information in a way that works indoors and outdoors, responds faster, and uses less battery power. This method returns the most recent location currently available, if another app has recently updated the devices location then this will be used otherwise it will be updated before the most recent value is returned.

1.2.1.9 Database Design The simplicity of this iteration means that it does not require a complex database. We fully intend to have to make sweeping changes to this design in future iterations, however it should form an easily-expandable base on which to build.

The DBMS used for this system will be Microsoft SQL Server 2014, due to its prevalence in professional applications, spatial data types and comprehensive high-availability features, including redundancy and failover, which are critical for a system as important as this.

1.2.1.9.1 Requirements

- *Registration details of responders*
  This need only be a unique identifier and forename and surname for now - it can be trivially expanded later on, so we will not dwell on implementation details.
- *Responders devices*
  Each responder can (and should) register as many devices as they have to their account, in order to increase the chances of them being successfully notified that their assistance is required. In order to do this, we need to store information about responders devices, in particular the push notification key that allows us to tell GCM (or any alternate service, such as APNS) which device to send notifications to.
- *The locations of these devices at any given time*
  To be able to find the closest responder to the incident, we need to track the locations of all responders devices in as close to real-time as possible. We could only keep the latest position, however maintaining the full history (or at least a certain amount of it) is a trivial change, and could be used in the future to improve the responder selection algorithm.
- *Incidents, and the responder chosen to attend that incident*
  The database needs to be able to store the location of incidents so it can send these to responders. It must also keep a history of which responder attended which incidents for logging purposes.

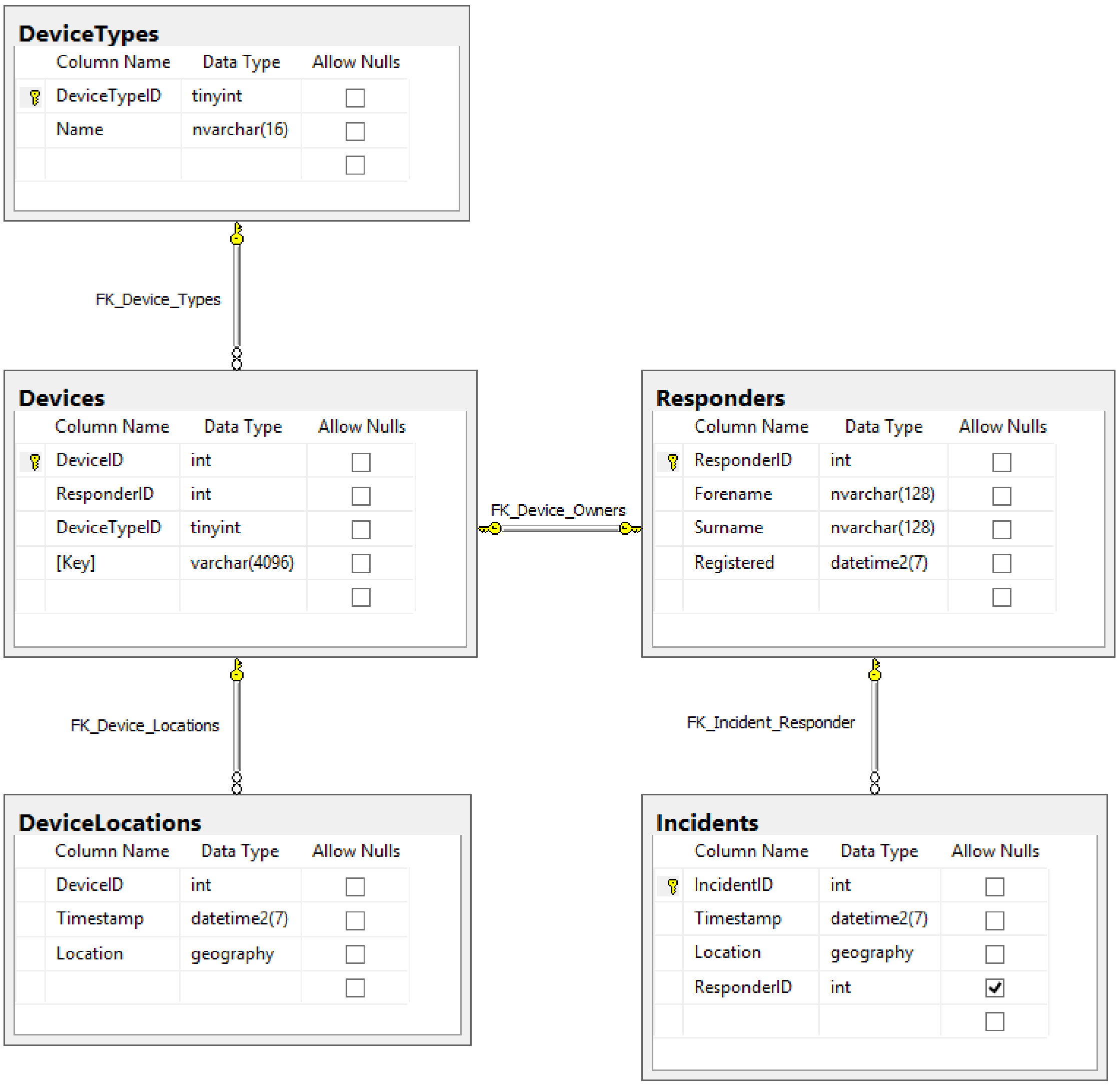
1.2.1.9.2 Data Types Location data is commonly represented as a latitude and longitude pair, however modern databases have dedicated spatial data types. SQL Server 2014 is no exception; it has the `geography` type for ellipsoidal data. The key `geography` instance type that well be using is `Point`: a 0-dimensional object representing a single location. Well use it to identify the whereabouts of incidents and responders devices. It has the added benefit of supporting elevation, should we need it in the future to increase accuracy of location data.

Primary keys will be `ints`, unless we can guarantee they will not overflow if a smaller width type is used. Unfortunately SQL Server only supports signed types, so 32-bit integers will allow up to 2,147,483,647 (231 - 1) rows per table. Considering there are roughly 65m people in the UK, this is more than enough to store all possible responders. Even at this limit, the type is large enough to handle each of them having 33 registered devices each - also an infeasible number. As already mentioned, there are an estimated 60,000 [2] incidents in the UK every year that this system could be relevant to. Assuming this figure doesnt exceed, an `int` primary key will sustain the system for 35 millenia. The only exception is device types: there are less than 255 services offering push notification facilities, so a `tinyint` will suffice for this purpose.

Device keys are allowed to be up to 4096 bytes, as GCM `registration_ids` can be up to 4KB in size, however in practice they will be much smaller than this - it is this long to handle edge cases.

1.2.1.9.3 Entity Relationship Diagram

Entity relationship diagram of the database.



DeviceTypes

	Column Name	Data Type	Allow Nulls
🔑	DeviceTypeID	tinyint	☐
	Name	nvarchar(16)	☐
			☐

Devices

	Column Name	Data Type	Allow Nulls
🔑	DeviceID	int	☐
	ResponderID	int	☐
	DeviceTypeID	tinyint	☐
	[Key]	varchar(4096)	☐
			☐

Responders

	Column Name	Data Type	Allow Nulls
🔑	ResponderID	int	☐
	Forename	nvarchar(128)	☐
	Surname	nvarchar(128)	☐
	Registered	datetime2(7)	☐
			☐

DeviceLocations

	Column Name	Data Type	Allow Nulls
	DeviceID	int	☐
	Timestamp	datetime2(7)	☐
	Location	geography	☐
			☐

Incidents

	Column Name	Data Type	Allow Nulls
🔑	IncidentID	int	☐
	Timestamp	datetime2(7)	☐
	Location	geography	☐
	ResponderID	int	✓
			☐

N.B. `Incidents.ResponderID` is nullable as it is conceivable that an incident might be created, and it take a few moments for a responder to be assigned to it.

1.2.1.10 Selecting the Responder to notify

In this iteration, we are simply choosing the single closest responder to the incident. In terms of the above schema, we need to alert the owner of the device whose latest position is closest to the incident.

We have a table of device locations; the first step is to extract the latest location of each device:

```sql
SELECT LatestLocations.DeviceID, LatestLocations.Location
FROM (SELECT DeviceLocations.DeviceID, DeviceLocations.Location,
            ROW_NUMBER() OVER (PARTITION BY Devices.DeviceID
                               ORDER BY DeviceLocations.Timestamp DESC) AS Rank
      FROM Devices
         INNER JOIN DeviceLocations
            ON DeviceLocations.DeviceID = Devices.DeviceID
    ) AS LatestLocations
WHERE LatestLocations.Rank = 1
```

As well as spatial data types, SQL Server also includes operations on these types, one of which is `STDistance`, which returns the shortest distance between two `Point`s. We can now intersect this with the known location of the incident, producing a list of `DeviceID`s in ascending order of distance to the incident.

```sql
-- will hold the location of the incident
DECLARE @incident geography;

-- look up the location of the incident with identity ‘x‘
SELECT @incident = Location FROM Incidents WHERE IncidentID = x;

-- sort devices by their proximity to the location of the incident
SELECT LatestLocations.DeviceID,
       LatestLocations.Location,
       @incident.STDistance(LatestLocations.Location) AS Proximity
FROM (SELECT DeviceLocations.DeviceID, DeviceLocations.Location,
            ROW_NUMBER() OVER (PARTITION BY Devices.DeviceID
                      ORDER BY DeviceLocations.Timestamp DESC) AS Rank
      FROM Devices
         INNER JOIN DeviceLocations
            ON DeviceLocations.DeviceID = Devices.DeviceID
    ) AS LatestLocations
WHERE LatestLocations.Rank = 1
ORDER BY Proximity ASC;
```

From here, it is trivial to look up the owner of the closest device and notify all of their devices of the emergency.

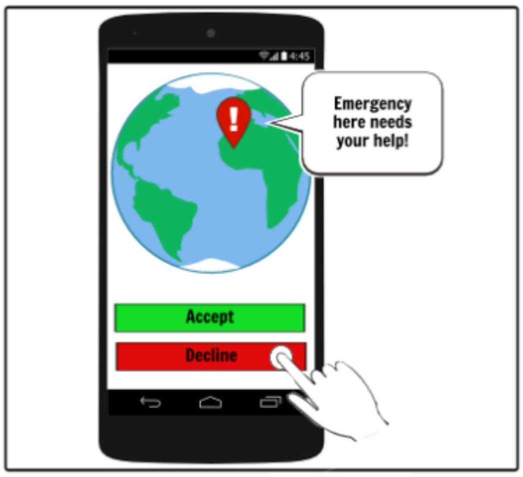
1.2.2 Iteration 2 - Accept & Decline an emergency

1.2.2.1 Aim This iteration addresses the issue of an individual either not being able to attend an emergency or not responding to the emergency request in a reasonable time. It will also address whether the person contacted is unreachable and how the system should react in that situation.

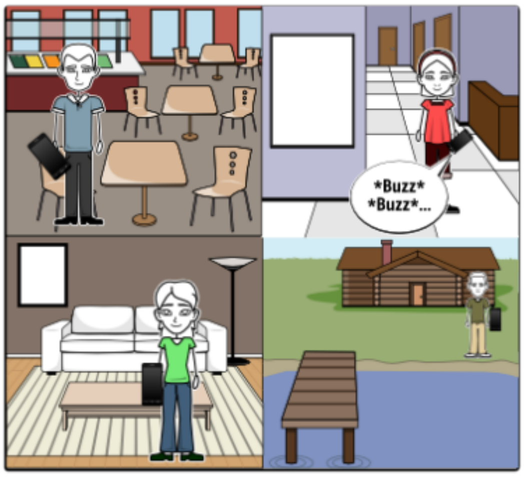
On completion, this iteration should give responders the choice to either accept or decline a request for attendance at an incident, and depending on their decision, the system should be able to react appropriately. It should also address the issue of the contacted user being unreachable after the system sends a help request notification.

1.2.2.2 Storyboard This storyboard visualises the intended functionality of the system.

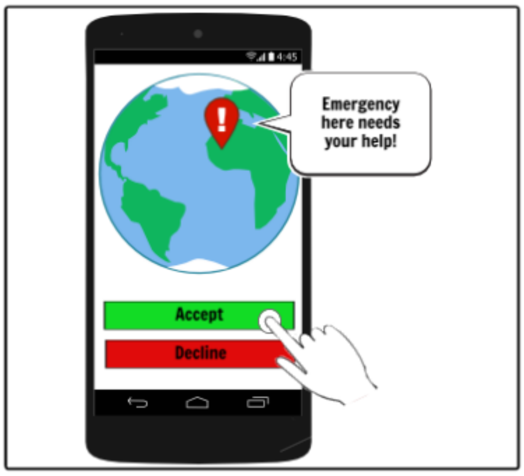
The closest person to the emergency is alerted to the situation.



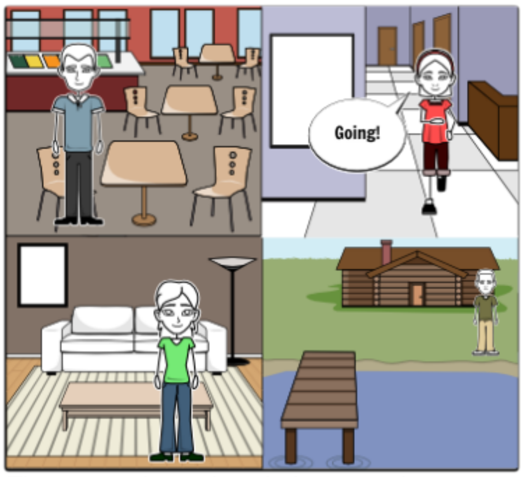
The user is unable to attend and so declines the request.



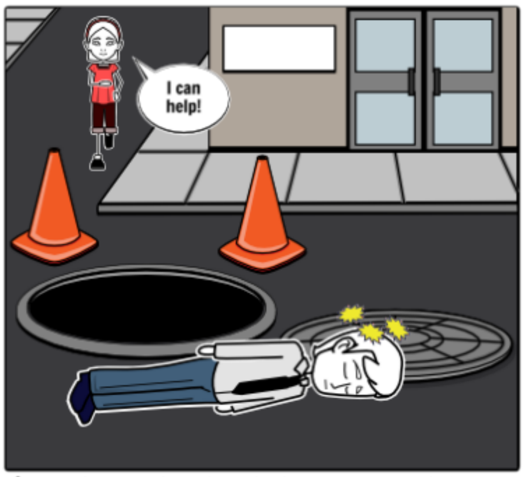
The system finds the next closest person to the emergency and notifies them.



This user is able to attend and responds saying they will accept the request.



The responder makes their way to the emergency.



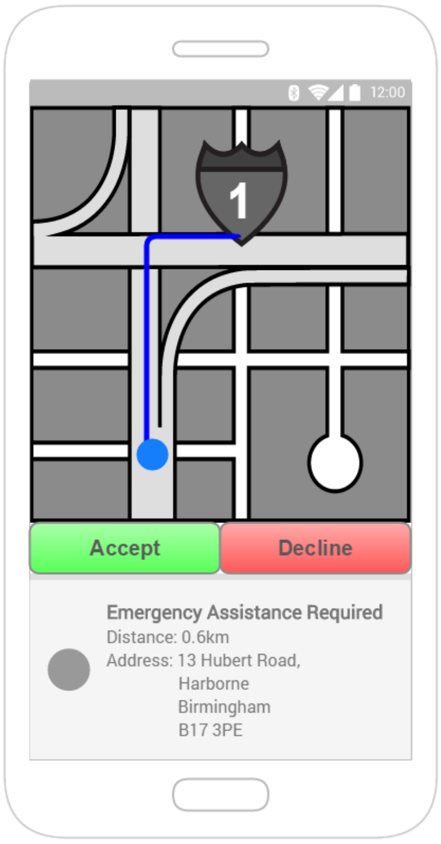
Once they arrive they help as appropriate.

1. The closest person to the emergency is contacted (as per the previous iteration).
2. They are now given the opportunity to accept or decline the request.
3. If they accept the request, they go off to help at the emergency. If they decline, the system contacts the next closest person to the emergency. Subsequent responders are also given the option to accept or decline.
4. This continues until a user says they are able to help or the request is cancelled/resolved by the dispatcher.

This will make our system more robust as we are no longer assuming that the first person contacted can attend the incident. It also gives responders more control over when they can help rather than simply being contacted and their attendance assumed (as in the previous iteration).

This reflects a much more realistic use case, as in reality responders will be busy, or their devices will be unavailable, or notifications will be delayed.

1.2.2.3 Confirming Attendance The best way for us to allow the user to tell the server if they intend to respond to an emergency help request is through Accept and Decline buttons. This presents the choice to responders in an obvious way that requires no training or learning time.



These buttons should be present in the app rather than the notification as then responders are able to see more information about the emergency (such as its the location on a map) before making their decision. It is also less likely to be miss-tapped, as the user first has to acknowledge the notification and open the app to be able to respond.

This prototype demonstrates the button placement of the Accept and Decline buttons. They have taken the place of the directions button (which started route guidance to the emergency), as this is not applicable until the user has indicated they are attending the incident. In addition, this placement does not interfere with any other visual elements; all information is still readable and clear.

The Accept and Decline button colours should be red and green respectively to fit in with the standard format for Yes/No buttons, and to encourage responders to accept requests for help. The placement of the Accept button should be to the left, as this makes it harder to mis-tap - the majority of the population are right handed, [29] and this positioning requires more of a stretch to reach.

We have also decided to implement a background timeout for the user to acknowledge a notification. This should start when an emergency request is sent from our server (via GCM) and the notification is displayed. The user will then have 30 seconds to open the notification before the timeout kicks in. If the user does not respond within this time, the notification is automatically declined and the server will send the notification to the next closest person as if the user had declined it themselves.

This is implemented because of the time sensitive nature of these type of emergencies. The idea of the first responder is to be able to attend the scene before the emergency services can get there (the target response time being 8 minutes[30]), so any delay in alerting a potential first responder reduces their usefulness. As a result, if the first person contacted does not reply quickly enough then our system will discard them and move onto the next person who might be able to help faster.

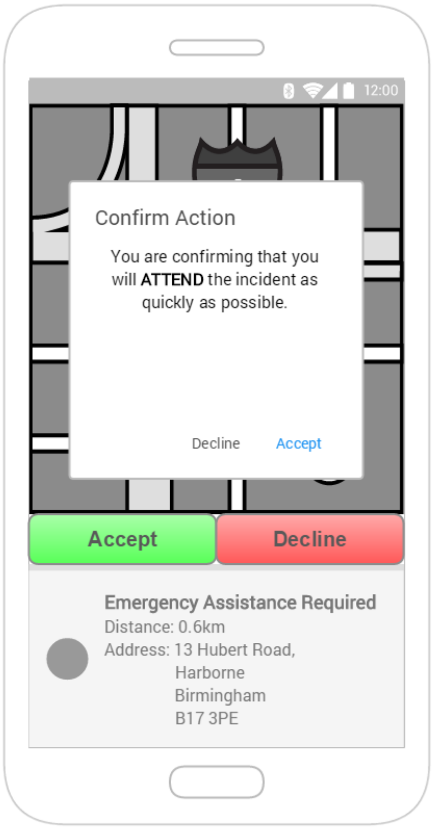
This cycles should continue until one of the following conditions has been met:

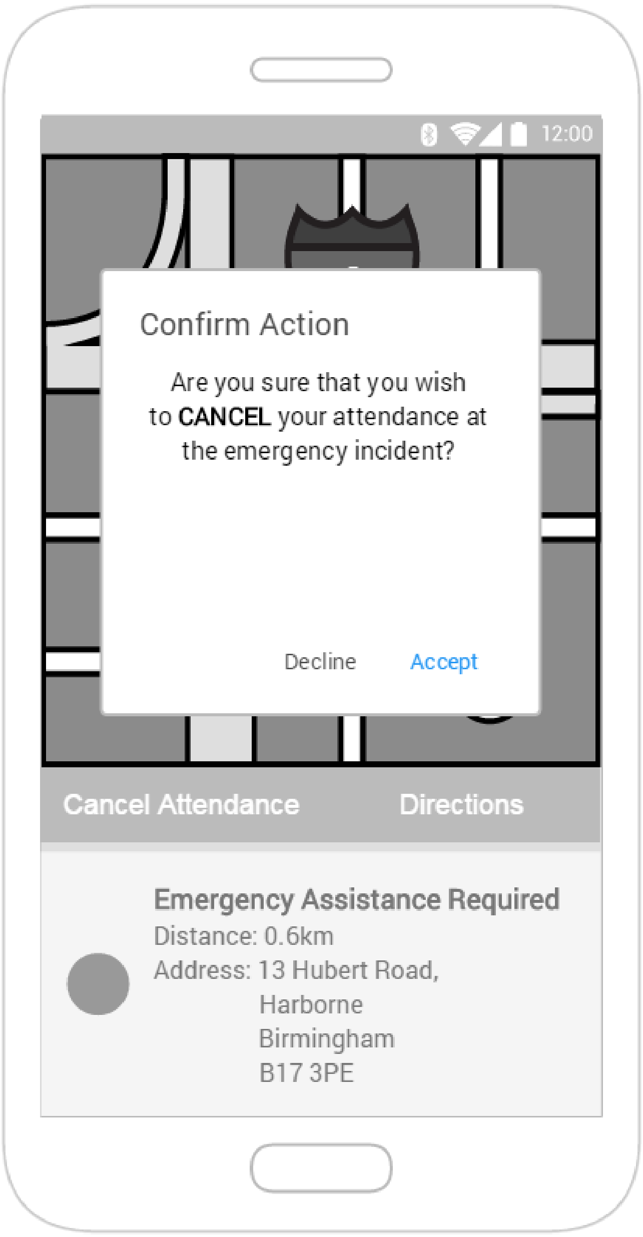
1. The 8 minute average response time has passed, after which we expect an emergency response to be either at the incident or very close. After this time the likelihood of a first responder being able to help the injured individual has also decreased dramatically, to the point where it is highly unlikely they would be able to do anything.
2. The next individual who is due to be contacted with the emergency is beyond the reasonable response distance for a first responder. This stops the chain of help requests reaching anyone who is too far away to be able to help. This distance should be based off the time it would take an individual to travel to the incident. On average an 8 minute journey would take the traveller around 2-3 miles in an urban environment, and up to 5-6 miles in a more rural setting. Therefore our cut-off distance should be towards the upper end to allow people in rural areas the chance to attend the incident. Around 5 miles should be appropriate.

We cannot use the state of the emergency as a cut-off for our system, as emergency responses tend to not report on the status of their emergency until the casualty has been delivered to a hospital. If we were to use this then our system would still be sending first responders to the emergency after the ambulance had left, wasting their time and reducing the likelihood of them continuing to use the system.

1.2.2.4 Confirmation Messages The Accept and Decline buttons should also have a verification message box to ensure the user intended to select that option. This should clearly state the action (accept or decline) and be placed on the screen so that the bottom information pane is still visible.

Finally, upon the user accepting an incident, the app pane should have a Withdraw button still visible for cases where there was a missclick (even after the verification), or a situation arises where the user is unable to attend (such as unforeseen traffic jam). Upon clicking this button the system should resume as if it were a normal decline, and alert the next user if the end criteria have not been met.


12:00
Confirm Action
You are confirming that you will **ATTEND** the incident as quickly as possible.
Decline
Accept
Accept
Decline
Emergency Assistance Required
Distance: 0.6km
Address: 13 Hubert Road, Harborne Birmingham B17 3PE



This button should be placed on the left hand side of the screen for the aforementioned reasons, and be next to the Directions button so that it does not interfere with any of the information being displayed to the user.

Upon clicking this button, the user should be presented with a confirmation screen which clearly states what the action they are performing means.

1.2.2.5 Keep playing alert sound until user accepts/declines This is another feature of the app which we discussed and decided on what to implement. Should the phone play a single notification sound or for the sound to keep playing (like an incoming call) until the user accepts or declines it. We decided to play a continuous alert until the user begins to actively use the phone, this means that the alert will play until the phone is turned on (from sleep state), the volume buttons are pressed (standard way to mute an alert on most devices) or the user accepts or declines the emergency alert.

This can be easily implemented as we are able to run any code which we like upon receiving a GCM message for this app. This will allow us to start the alert tone and add hooks into the various stop events which will cancel the playing of the alert.