PiNotify Design

Intro

PiNotify seeks to provide a way to notify and send short (140 character) messages via a tablet display, along with physical blinking lights and a physical button.

The flashing lights will blink until the message is acknowledged.

PiNotify allows sending messages from a web-based UI (implemented in Google AppEngine) to a mobile device (Android tablet), which will then relay the message to a Raspberry Pi via Bluetooth. The message recipient will be able to press a button (TODO: connected to Pi or to mobile device?) and dismiss / acknowledge receiipt of the message. The Raspberry Pi (TODO: or tablet?) will be wired to a bright notification light that will blink upon receipt of new messages.

It is assumed that Wi-Fi is not present, and that only the mobile device will have Internet connectivity.

The system is currently designed for a very small number of users (a single tablet + Pi recipient and ~10 web UI senders).

Design

Overview

As part of sending a message, the AppEngine code will notify the tablet via a Firebase Cloud Messaging (FCM) push notification. This allows the mobile device to immediately display the new message, rather polling.

Given the lack of ordering guarantees provided by FCM, this push notification will not contain the actual message payload. Rather, it will instruct the mobile device to immediately pull a message from AppEngine. (The push notification does have a single data key/value payload containing the push notification ID -- this is only used for performance monitoring).

This will be accomplished via a Cloud Endpoint API exposed by AppEngine. The response to requests will include the message payload and metadata details about the message (sender, date, etc.)

The endpoint API protocol is very simple, requiring only a single endpoint method, get_messages, to provide all the below functionality.

Authentication

All API calls and web UI usage must be authenticated. For now, this will be via a list of authorized Google accounts and the AppEngine “admin” account concept.

The server stores both a table of authorized email addresses and a table of authorized users. The two are separate as the latter contains the user identifier. This allows authorized users to change the email address of their Google account and remain authorized (as long as it is still the same account).

Users should be removed from the “by-email” table once their identifier is stored.

Only AppEngine “admin” users may authorize new email addresses.

TODO: Should friendly names come from the account, or should they be provided by users?

Device FCM Registration

The mobile device registers for notifications by reporting both its unique device ID and its FCM ID with every message sent by the device. This allows the server to be notified of FCM ID changes for a given device -- the server will store a map (device ID -> FCM ID).

The device will periodically (say, every hour) send a request unpromoted by push notifications.

Message Acknowledgement

Every sent text message has an associated numeric ID. The first such ID is 1, and the message ID increments as a 64-bit integer.

In the server response, the least recent unacknowledged message is returned, along with its ID. If no such message exists, all the returned fields are uninitialized (None in Python).

Messages automatically “expire” after a day and are treated as acknowledged. The UI can report that a message expired without being viewed. Web UI users may also force a message to expire, for instance, to force another message to be shown on the live display.

The device will report the ID of the last acked message. If the device hasn’t ACK’d any messages, this field will be uninitialized. ACKs are cumulative -- ACK’ing message N ACKs all messages in the range [1, N].

TODO: How does the server know what the current message number is, and know that consistently across AppServers? Maybe just go by date? Also, we should save the date on the device to solve the problem of acknowledgement not taking effect if it occurs while the device is disconnected.

Device disconnection

Health-checking and Performance Monitoring

In addition to the push notifications sent to inform the device of new messages, push notifications will be sent periodically (say, every hour) to health-check the system. This is in addition to the periodic unpromoted by push notifications requests in the “FCM registration” section above -- the unpromoted requests ensure that the device remains registered, whereas the push notification health checks verify that push notifications are working as well.

Each push notification shall have a unique, incrementing 64-bit integer ID in its data payload. This ID should be provided on each request by the device, and it will be used to measure the end-to-end response time of the system.

Unpromoted requests will leave that request field uninitialized.

Communication with the Raspberry Pi

Communication with the Raspberry Pi occurs over a simple protocol using serial over Bluetooth.

It's important that the Pi and the tablet device reconnect even if each restarts in arbitrary order, so a bluetooth listener socket should run on both the tablet and on the Pi. Even though bluetooth RFCOMM serial sockets are bidirectional (full duplex), each side will use a socket it obtains from its call to connect() to send messages to the other side. Existing connections may be reused, but if sending fails, the existing connection can be closed and another created using connect() to reconnect.

Each side will explicitly ACK messages over the same connection from which the message was received.

For simplicity, messages in each direction will consist of a single octet. ACK will be denoted by 0xFF. If the tablet's "receiver socket" (that is, a socket obtained by accept()) receives 0xFF, this is a health check message, which should also be acknowledged with 0xFF (only sockets obtained from accept() should respond to 0xFF).

While the Pi will send 0xFF over its connect() socket for health-checking (in addition to ACK'ing messages over its accept() socket), the tablet will instead repeatedly send its last message. The Pi's communication of dismissal of a message will be communicated by 0x00. When the Pi starts up, it will send 0x01 to force the tablet to re-send its state.

Messages from the tablet to the Pi shall be a bitfield of two parts, one for each nibble (4 bits). The first nibble will encode the time in quarters of a second the light should be on each duty cycle, and the second portion the time the light should be off. 0x00 means the light is always off.

If the Pi fails to communicate dismissal of a message to the tablet, it will still turn off the blinking light, and it will **NOT** attempt to retry transmission of the dismissal. This will mean that upon reconnection, the Pi may be told to blink again for the same message.

Blink duration should be long so as to not annoy the human message recipient.

The health check interval Pi <-> Tablet can be shorter than the Tablet <-> Cloud health check interval since both devices are plugged in, and no cloud quota is consumed by increasing this

health check interval. Shorter intervals decrease the reconnection latency in the event of a disconnect.

Pi Starts First:

Pi starts server.
Pi -> Tablet: connect() if existing socket None or broken, 0x01. Recv No response, tablet is off (connect() fails?). This is broadcast to all found devices with registered UUID.

Tablet turns on.
Tablet starts server.
Tablet -> Pi: connect() if existing socket null or broken, 0x00 (if no message), or something like 0x88 (2 sec on, 2 sec off). Recv 0xFF ACK.

Tablet Starts First

Tablet starts server.
Tablet -> Pi: connect() if existing socket null or broken,

TODO: startup and reconnect details?

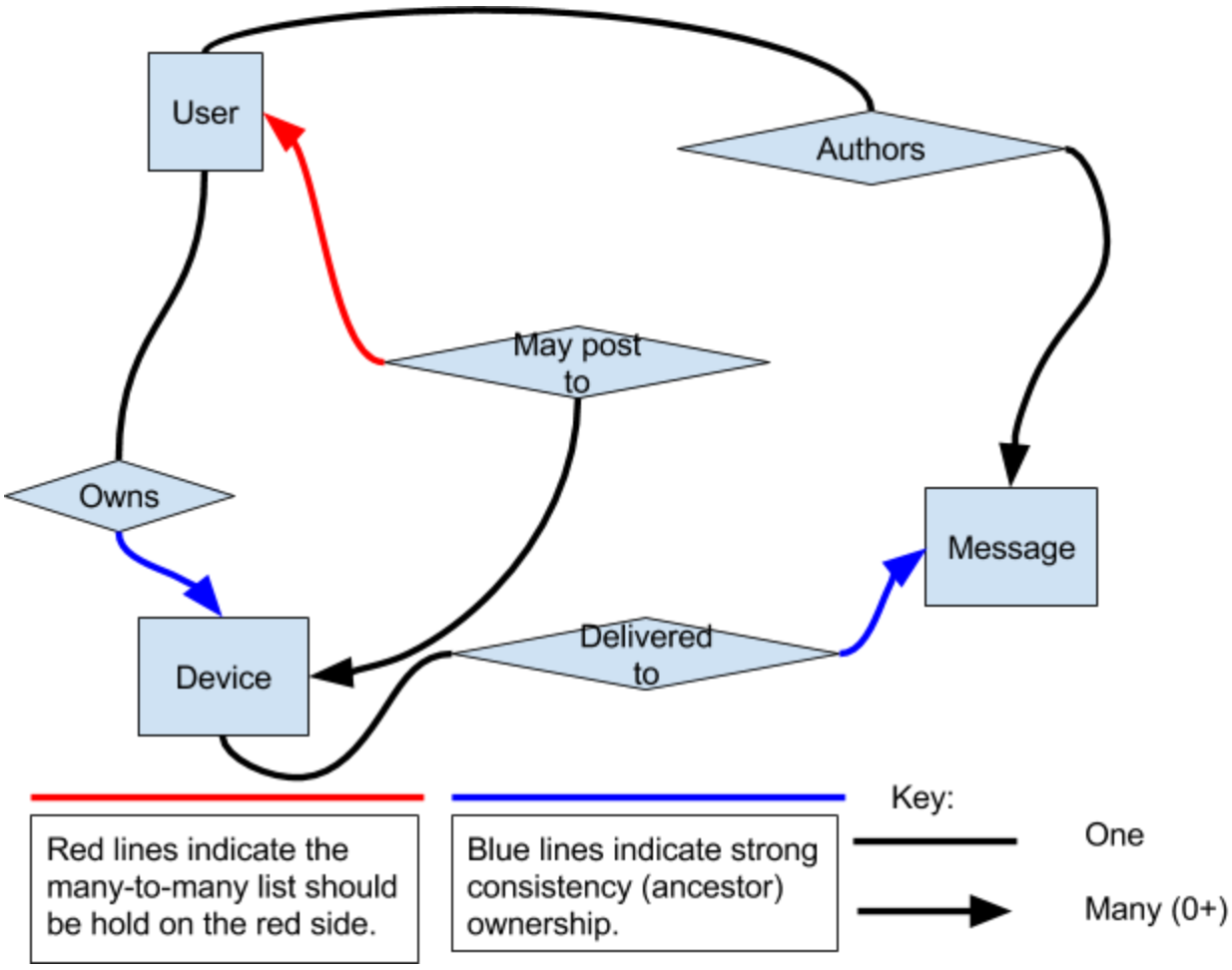
Android Lifecycle, Tasks, and Activities

There will be a single Activity that's declared as a "singleTask".

The FCM broadcast receiver and an AlarmManager alarm setup on boot (for health checks) will both make backend requests. If there is new information to display, an intent will be sent to the activity that includes the encoded message details.

In the event the user isn't logged in, a separate intent will be sent to the activity to initiate login.

Data ER Diagram



The mobile app client will persist the details of the last message it received in local persistent storage. This allows it to handle restarts and crashes, and inform the server of acknowledged messages upon reconnection.

Queries for the messages for a given device shall be **strongly consistent** so that when the device receives its push notification, it can safely assume any associated new message is already accessible via query. It is assumed that the post rate to a given device will be less than 1 post per second -- more than this would be extremely overwhelming to a user anyways.

Other queries may be eventually consistent, and shall be maintained by tables storing keys of other tables (for one to many), or a list on one side of such keys (many to many). For the user-device relationship, this list will be held on the user side, as it is expected that the number of devices a user may post to is smaller than the number of users who may post to a device.

Because of these consistency requirements, messages shall have the key of the device to which they are addressed as their parent so that that parent may be used as an ancestor key.

Ancestor queries in Google Cloud Datastore are strongly consistent, but writes with parent keys must be limited to once a second.

Only users in the User table may access the system. Device owners may invite other users to post to their device via the web interface by email address. The Google identity associated with that email address gets added to the table when that user first logs in.

URL Handlers

Handler	**Description**
/	List of devices you own, and devices to which you may post.
/register_users	See who's authorized, and authorize new users. Click on a user to go to their /user page. (For OSS release, this should not show every user at once, but instead offer a pager and search bar).
/device	List of messages for a device, and authorized users. (Devices are automatically added by launching the app on the device and logging in with an authorized user).
/user	View a user's profile. The main functionality is selecting which devices owned by the logged in user the (other) user may post to.
/login	Users get kicked here on logout and auth failure. Maybe skip and go straight to login? Might be nice to have explanatory text though.

Blinking Light Selection

The light must be bright enough to be seen, and have brightness controllable by software. Software control of color is a plus.

The device must be able to charge while the light is in operation. This unfortunately rules out USB HID class controlled LEDs, like the BlinkStick, even though these devices may be

controlled by Android userland without rooting. USB HID also may require private APIs for an iOS verison: http://stackoverflow.com/questions/9373453/ios-usb-hid-programming.

So, Bluetooth controlled light is probably better. This can either be accomplished by reverse-engineering the protocol on an existing smart bulb (using the HCI dump in Android's debug options), or by controlling simpler bulbs from a controller like the Raspberry PI.

Appendix

Make curl work

```
curl --header "Content-Type: application/json"
localhost:8080/_ah/api/helloworld/v1/hellogreeting
```

Must be api not spi. Header might only matter for post.
Order is api_name/version/method_path