Stroboscopik

By Namratha Vedire, Lewen Yu, Siddhartha Banerjee

Stroboscopik is an Android project that generates “synchronized and interesting” light patterns across the screens of multiple Android devices.

I. Vision
The target audience of this project are concert-goers who wish to wave a bright, flashing object in synchrony to the music. Traditionally, people in concerts have used (cigarette) lighters to move with the music, but more recently, cellphones have begun to functionally displace lighters as the light source of choice. We wish to capitalize on this paradigm shift in concert audiences, and by intelligently varying the flashing across all devices, we want to elevate the mood of the event.

I.a. Intelligence Criteria
In order to produce interesting and meaningful results, we think that the intelligence of flashing should depend on 2 main factors – the relative Locations of the devices, and the Music that is being played at the event. Although the production of unambiguous patterns throughout the crowd requires fine-grained localization of the phones, we decided to defer creating a solution to that problem in a later iteration of this project because of the complexities and challenges involved with very precise location calculations. Instead, in this iteration of Stroboscopik, we have focused on developing an intelligent flashing algorithm as a function of the Music and coarse-grained Localization.

II. Architecture
The App has a hybrid client-server architecture as shown in Figure 1. The smart phones form local clusters through Bluetooth in order to implement coarse-grained localization; while the server acts as a music conductor in order to ensure “intelligent” flashing based on the music being played at the event.

II.a. Server Overview
In order to produce an emergent visualization of the music across a crowd, the server uses its knowledge of the music being played to determine the main “beats”. In doing so, the server obtains a set of frequencies that clients should flash at. It then distributes these frequencies across clusters of clients in the audience to produce desired “interesting” visualizations.
II.a.1. Server API

The Server has been implemented on a Django Web Framework served by a PostgreSQL database. This web framework is meant to provide users (ideally concert organizers) with the ability to create events, specify the songs at an event, and even view the clusters that are formed at an event. Although at this moment the website is not a fully functional website, it does provide primary server functionality through its API. On a high level, these API methods are the following:

1. **determine_freq (input)**
   - This method takes a temporal description of the music (as a .wav), and returns a list of main frequencies in the music.
     
     **input** -> Temporal description of music (perhaps as .wav)
     
     **Return** -> List of main frequencies that clients can flash at

2. **distribute_freq (freq, clusters)**
   - This method takes a list of frequencies and a list of clusters, and then sends out GCM messages to devices in each cluster based on the frequency assigned to the cluster.
     
     **freq** -> List of frequencies available
     
     **clusters** -> List of clusters being distributed over frequencies
     
     **Return** -> True if sending successful, else error details

3. **view_clusters([cluster_ids])**
   - This method provides a list of clusters along with their corresponding status in terms of number of devices, the current frequency of flashing, etc. If the optional argument cluster_ids is provided, then the status reports are restricted only to those cluster_ids that are passed to the function.
     
     **cluster_ids** -> Optional argument for getting status of only a few clusters
     
     **Return** -> Status report in the form of a dictionary

II.a.2. Determining Frequencies

The aim of the server is to determine a set of distinct frequencies that the clients can flash at. Since flashing in sync is similar to “tapping” with the beats in the music, the problem of determining sync can be boiled down to a problem of determining the various beat frequencies in a particular piece of music. Of course some pieces of music (most notably classical music) change their music signatures from one phrase of the song to the next; but for simplicity’s sake, we decided to ignore such variations and instead focused on the global trends in the music.

The method of calculating beat frequencies was very straightforward:

1. Divide a sample of music into 32 frequency sub-bands in order to account for differences in beats between various different instruments.
2. Calculate the beats present in each of the sub-bands by comparing instantaneous energy to average energy of the music: a beat is a noticeable increase in instantaneous energy compared to the ambient energy of the music.
3. Derive the frequencies of the beats by performing an FFT on the beat graphs and then selecting the most prominent frequency

The output of this algorithm after stages 2 and 3 can be seen in Figure 2.
As a final point of clarification, although the algorithm is very straightforward, it is computationally intensive. Hence, the server pre-computes the frequencies when a song is loaded, and stores the set of frequencies associated with the song in the database for later use.

II.a.3. Cluster Control
As the API states, the server also provides an interface to control the clusters. In order to accomplish this, the server takes advantage of GCM. When a cluster is assigned a frequency (either manually or through computed song data), the server updates the frequency of the devices in each cluster through GCM. This allows for both low power communication, as well as easy implementation of broadcast messages.

II.b. Client Overview
The clients (at this moment) are Android devices that are capable of both Bluetooth and Internet (particularly GCM) communication. During the development process, we determined that the client App should have 3 main responsibilities:

1. **Cluster Formation**
   The App is responsible for coarse-grained localization. It forms clusters with nearby neighbors through Bluetooth, and in the event of a lack of neighbors, it creates its own cluster.

2. **Cluster Management and Synchronization**
   Under this responsibility, the app ensures that the client device is always part of a cluster, and it also ensures that the client can seamlessly transition from one cluster to the next when it loses contact with the old cluster. Finally, and most importantly, the app ensures that all clients within a cluster are flashing within an acceptable phase difference of each other.

3. **Flashing and UI**
   The App is also responsible for maintaining the flashing frequency of the device and for ensuring an engaging User Experience for the end-user. To this end, there was a strong emphasis on making the UI as simple as possible without compromising on the functionality.

II.b.1. App Lifecycle
In order to handle its different responsibilities, the App’s lifecycle can be summarized by the figure in Figure 3. In essence, the life-cycle is as follows:
1. Every device starts out as an Orphan – a disconnected device that is attached to no cluster.
2. The Orphans use Bluetooth to scan their neighborhood for clusters for a determined amount of time.
3. If the device finds a cluster near itself, it associates itself with that cluster becoming what is known as a “Subnode”.
4. As a Subnode, the device continues to ensure that it is within range of its cluster. If at any point of time the device determines that it has lost its connection with its cluster, it becomes an Orphan again and begins a new search for nearby clusters.
5. In the event that the device does not find a cluster nearby, it elevates itself to the status of a “Supernode”.
6. As a Supernode, it is the device’s responsibility to inform the server of the formation of a new cluster. It is also the device’s responsibility to broadcast its new cluster to other potential devices through Bluetooth.

When the App is shut down, the devices return to being Orphans and the entire cycle starts over from the beginning.

II.b.1.i. Orphans
The Orphan phase of a device’s life-cycle is a critical period; in particular, the timeout to declare a Supernode is important. The timeout must be both long enough to allow the device to discover nearby clusters, as well as short enough to not give the appearance of an application in stall. Also, in the unlikely event that multiple devices try to form a cluster at the same time, some devices must switch states before others so that not every device is a Supernode.

Hence to solve this “Startup Problem”, the clients use a random timeout that is seeded on each device’s UUID. The randomized algorithm also favors some devices over others in order to ensure that some devices become Supernodes much more quickly than others.
II.b.2. Device Clusters
Clusters are an integral part of this project’s functionality. Hence, a bulk of the application code has been written to deal with this problem. The main challenges tackled by this portion of the application code are:

1. Seamless cluster formation and break-up
2. Fast and efficient intra-cluster communication
3. Synchronization of devices within a cluster

In fact, we chose Bluetooth as the primary medium of communication within a cluster precisely because of the need for fast and efficient intra-cluster communication.

II.b.2.i Cluster Formation
In order to speed up the time of cluster formation, the Android application on a device changes that device’s display name to a string that has a format specific to Stroboscopik. This new string contains important intra-cluster information such as the Cluster ID, the frequency details of a cluster, as well as a time stamp of the device flashes.

When a new device is looking for a cluster to join, it searches for devices in its vicinity that have Bluetooth device names that are specific to Stroboscopik. Upon finding one such device, the new device parses out the relevant information about the cluster, and effectively becomes part of the cluster from that point onwards. The device also notifies the Application Server of its new cluster, but that is just for book-keeping purposes.

Finally, a device is considered to be out of range of its cluster when a scan of neighboring Bluetooth devices fails to reveal a device with the same cluster information in its name as that of the device.

II.b.2.ii Fast and Efficient Intra-Cluster Communication
The communication of information through the device name is a novel and extremely fast method of sending information from one device to another. It works particularly well with Stroboscopik’s requirements because:

1. It avoids the overheads associated with setting up and tearing down Bluetooth pairings.
2. It is a form of broadcast messaging. Stroboscopik does not need peer-to-peer messaging within a cluster.
3. A Bluetooth piconet is restricted to 8 devices alone. By avoiding the creation of explicit network connections, we can support clusters of much more than 8 devices
4. The Bluetooth adapter is only scanning for neighboring devices, and we avoid the battery and power overheads of point-to-point communication.

II.b.2.iii Synchronization within a Cluster
Synchronization is done in a two-fold manner. Firstly, all nodes provide others with a timestamp of flashes so that other devices can compute the time of next flash to synchronize the phase. Secondly, the nodes periodically query the Server for time information to keep their clocks synced.
III. Testing Results and Performance Estimates

The Application is mostly functional, and we have managed to create clusters that have flashed in sync. However, due to the usage of only 4 Bluetooth capable devices, exhaustive testing of this application was not possible. But, the observations that we have made during testing have been revealing:

1. Possible range of frequencies for the strobe function is 1Hz-20Hz (based on empirical data from screen refresh rates). However, it appears that even at frequencies of around 16Hz, it is difficult for the human eye to “see” beats.

2. While running the app multiple times, we found that GCM was unreliable. In the sense that it did not quite push the messages from the server to the device in real time and sometimes, skipped pushing completely.

3. The devices did appear to be in sync, although it is unclear if this was due to our efforts of trying to sync them, or because of the human eye’s inability to detect phase differences in frequencies upwards of 5Hz.

4. The bulk of the device power was consumed in the act of flashing the screen. The control for this test was a device that simply set to strobe without using any Bluetooth or Wi-Fi. Empirical comparisons revealed that the control device lost power at a rate that was almost comparable to a device that was running the complete Stroboscopik application.

5. Having multiple devices flash in various colors, and at different frequencies in sync to the music, did indeed produce the mesmerizing effect that we’d hoped to achieve.