SmokeSignals: A Distributed Key-Value Store on a Mobile Network

Department of Computer and Information Science
Senior Design 2015-2016
Advised by Boon Thau Loo

Charles Cobb charcobb@seas.upenn.edu
Meyer Kizner mkizner@seas.upenn.edu
Xiuruo Zhang xiuruo@seas.upenn.edu

Introduction

Internet-enabled smartphones are a ubiquitous part of modern life, but they depend critically on centralized infrastructure to access the network. This infrastructure is generally reliable, but has several shortcomings. First, large gatherings like sporting events or concerts often overwhelm the limited network capabilities in an area. Second, during a disaster such as a hurricane or terrorist attack, networks often fail completely when communication needs are most critical. Lastly, users may prefer a surveillance-resistant decentralized network for sensitive communications.

Our proposed solution is a decentralized peer-to-peer key-value store replicated across mobile devices. It should:

- synchronize with nearby devices when they are in range, on a best-effort basis
- propagate changes across the network even as nodes move and sever existing links, or leave the network entirely
- provide a general API allowing developers to easily build a variety of peer-to-peer applications using our software

Approach

Our project separated cleanly into three self-contained components: the key-value store proper, consisting of interactions with the underlying database as well as serialization; the synchronization mechanism, including connecting to peer devices via Bluetooth; and the user-facing applications which are built on top the other two components.
Key-value store

We decided during development that we would not attempt to design a persistent database from scratch. A database completely of our own design would require us to deal with many low-level details, perhaps including kernel-level APIs, which were not the focus of our project. Additionally, it is unlikely that we would be able to reproduce the performance and quality of an existing, open-source mobile database. After considering several possible alternatives, including SQLite, we settled on Couchbase Lite, a NoSQL, document-oriented database available on our target platform, Android.

Nevertheless, we were unable to use Couchbase as our entire persistence layer for several reasons:

1. we wanted to expose a much simpler API than the one provided by Couchbase, for ease of testing and development;
2. in the context of our project, client requests to update or delete keys can arrive out of order because of the nature of a decentralized mesh network. Therefore we needed a custom conflict resolution scheme to handle collided keys;
3. we had an additional requirement that all keys and values stored in the database be serializable across a network connection.

We therefore wrote our own key-value store driver, which was the only component making use of Couchbase directly, to expose our API to the other components and to perform serialization. We implemented standard database operations (create, retrieve, update and delete) on key-value pairs consisting of string keys and values that can be either a string or a binary object (raw byte arrays). In particular, each key-value pair is created and stored with a last-modified timestamp, a modifier unique identifier (UUID), as well as a “deleted” mark. Updates to a key would only be accepted if the timestamp included with the update is newer than the key’s current timestamp and in the case of a tie, we choose the version with a higher modifier UUID; for deletions, we never truly delete a pair from database, but instead, we mark the pair as “deleted” so that other devices in the network is aware of the deletion when syncing.

Synchronization

In order for our store to be useful, it must be continuously synchronized with nearby Bluetooth devices—even when an application using the store is running only in the background. To this end, we developed an Android service to perform this synchronization. The service is transparent to the user, and once it is invoked by the application, it keeps running in the background until the application terminates it.

The synchronization service acts as a Bluetooth client and server simultaneously. When the service starts, it creates a Bluetooth server socket which it listens on continuously in a dedicated thread. As the socket accepts connections, new threads are spawned to handle them.

At the same time, the service creates a separate thread to perform Bluetooth device discovery through the Android Bluetooth API. Because the discovery process is resource-intensive, we wait 30 seconds after one discovery cycle before beginning a new discovery round. The process itself takes about 10 to 15 seconds; it involves listening for a number of asynchronous events from the Bluetooth API. When a device is discovered, and we determine it is also running our
synchronization service, we attempt to connect to it as a client, which also involves creating a connection to be handled on a new thread.

An important part of the discovery process is the use of the Service Discovery Protocol (SDP), a part of the Bluetooth stack, to determine which discovered devices are actually running our synchronization service and ready to exchange data. This protocol allows potential peers to determine which processes are listening for Bluetooth connections by means of an asynchronous event. Our process is identified by a UUID generated for this purpose, which is shared across all devices and applications making use of our store.

When a connection is established between the client end of one device’s service and the server end of another instance of the service, the two devices perform the synchronization procedure. In our current version, this procedure is straightforward: we simply send a copy of the local key-value store to the remote device. If we receive an update from another device which conflicts with one of our own, we use the modification timestamp in order to decide on a version to keep; if the timestamps are identical, the tie is broken deterministically using a device-specific UUID, which is also exchanged as part of the synchronization procedure.

Applications

We built two demo applications with different functionalities and use cases for this project. The first demo is a broadcasting application where users can choose a username and then input any messages to be broadcasted to everyone else using the application. In this demo, the key we store is the timestamp of the message and the username separated by a space; except in an extremely rare case where two users choose the exact same username and send a message at the exact same moment, this scheme can avoid any conflict in keys so that every broadcasted message has a unique key. The value is the message needed to be sent. The other demo is a photo sharing application similar to Dropbox. Users can upload photos from their photo albums to a shared folder and every image in this photo is automatically synced with all other devices in the background. In this scenario, we chose to store the image hash generated with the MD5 algorithm so that there will not be any duplicated images in the shared folder.

Results

Our infrastructure is designed for mesh networks so ideally we would like to evaluate our results on a large number of devices. However, due to the limited resources we had for this project, we could only get five working Android devices to test on and thus evaluated the metrics on a small sample size. Metrics like maximum distance propagated, or time required to transmit between two remote devices on this small sample would not be realistic for a large network, but still reflect the potentials of our concepts.

Performance between two devices
**Time required for the discovery phase** (including Bluetooth device discovery and SDP lookup): between 20s and 30s

**Time required for transmission phase** (purely textual data): under 1s

**Time required for transmission phase** (binary blobs, tested with images): 10s per 3Mb image

**Maximum distance two devices can communicate**: 100 feet

**Failure rate**: 1 out of every 4 cycles

---

**Overall performance among many devices:**

With the devices we could test with, we observed a clear trend of increasing speed and reliability as more phones are added to the network. This is indeed an expected merit of a decentralised mesh network -- contrary to traditional networks which suffer frequently from overloading, our network quality becomes better with more nodes, as a single failed link can be easily compensated by other alternative links. This characteristic, although on a small sample, makes us believe our project is successful as a proof-of-concept.

---

**Ethical and privacy considerations**

A vulnerability of our design lies in the fact that following the Bluetooth protocols, we require all participating devices must get permission from their users to pair with at least one other device over Bluetooth in order to receive and send data. However, Bluetooth does not tie the pairing permission to one particular service or application and thus the if two devices are connected as paired devices, data transfer can happen without any further permission from either user.

---

**Discussion**

**Difficulties Overcome**

We overcame considerable difficulties in exploring novel concepts that haven’t been well studied or widely attempted before. Bluetooth is a long existing technology, but turned out to be very unreliable in our project. We initially had troubles consistently establishing connections and discovering devices even though we followed every documentation of the Android Bluetooth API. After much investigations on our own, we were able to minimize failures to a reasonable level by meticulous timing and resource management.

One example of the complicated problems we solved to improve Bluetooth performance is concurrency. A key component of our infrastructure is discovering nearby Bluetooth devices running our service. Discovery is a process that lasts 10-15 seconds. During that time, no connections can be made to other devices, as discovery consumes all of the resources of the
Bluetooth Adapter. However, the function call that starts discovery is asynchronous, and also times out sometimes and never executes the callback. The function call that fetches the list of services a device is running (SDP) is much the same. So managing concurrency in this situation is hard; you have to somehow wait until discovery is finished, but discovery sometimes times out and never finishes. We ended up using semaphores with timeouts. We start discovery and try to acquire a permit for 15 seconds; if the semaphore isn’t acquired in that time frame, we stop discovery ourselves, and work with the devices we found, if any.

We also taught ourselves Android for this project, which none of us had experience with prior. Luckily, Android was not extremely difficult to learn, as we all used Java extensively.

Strengths

The main strength of our project is its generality and simplicity; one can easily build peer to peer Bluetooth applications, without writing any code or understanding the Bluetooth API at all. All Bluetooth communication is entirely hidden in the classes we created. An application developer simply needs to start the synchronization service (one line call), and then read from and write to the key value store. The only challenge the developer faces is how to encode the functionality of their application in terms of key value storage. In true distributed systems spirit, the key value store looks like it resides on one machine (phone), where in actuality, it is spread across many phones.

Our project also displays great fault tolerance. Phones can join or leave the network at will, without the system having to stop at all. Of course, the system works better when there are more phones, but an application can read or write from the store whether there are phones in range or not. An application can use the store even if the synchronization service itself fails. And the synchronization service will continue trying to find phones regardless of whether there are phones in range. Our infrastructure does not depend on any other phone being present.

Our infrastructure displays great availability. Calls to the key value store never fail. The store always just returns whatever it has. The synchronization service is responsible for keeping the store as up to date as possible, but there are no consistency guarantees. This is unavoidable, as the latency of the synchronization service is prohibitive. Discovery, SDP and the actual exchanging of data take about 30 seconds. We can’t block for 30 seconds on a function call just to make sure we have up to date data. And without executing the whole synchronization process on our end, we have no way of knowing whether nearby phones have new updates that haven’t yet been broadcast. Lastly, network partitions are extremely common as phones can go totally out of range. In this instance, it wouldn’t make sense to have the key value store not function. Again, it simply returns whatever it has.

Our project demonstrates the generality of a key value store. We built two significantly different applications: broadcast messaging and image “dropbox”. One could also build point to point messaging, RSS feed or even a social media platform, all using a key value store.

Our project also demonstrates the advantages of a de-centralised mobile mesh network over traditional networks relying on central infrastructure. Our infrastructure displays remarkable fault tolerance and availability, while exhibiting reasonable latency.
Limitations

Besides the privacy concerns discussed in the previous section, our current solution also has three other major limitations that we need to address in the future.

**Reliability**

In our testing, we experienced a relatively high failure rate of the Bluetooth Service Discovery Protocol (SDP), which is essential to discovering and connecting compatible devices. The failures are not explained in the documentation from Android [1] and impose a limit of reliability of the syncing layer.

We also experienced difficulties in the discovery phase as well. Our newest phone, a Nexus 5x running Android 6.0 (newest version), could never discover another phone. Periodically, some of the phones that usually discovered other phones would be unable to discover all or some phones that were in range.

Most of these issues come from the intrinsic unreliability of the Bluetooth technology which was not originally designed for long-lasting continuous connection and we believe our current limitation of reliability can be easily overcome if newer technology like peer-to-peer Wifi becomes more prevalent.

**Scalability**

Our implementation has space concerns. When we delete a record, we need to make a record of the deletion, so the deletion can be propagated to other phones in the network. However, this means the number of records never decreases. In addition to this, every phone stores a copy of the entire database. This makes our infrastructure poorly suited for applications that want to use a lot of data.

Our implementation has latency concerns. When we discover a phone, we send the entire database, instead of just sending the updates that have occurred since we last communicated with that phone. Images can take up to a second each to send, so these latencies can quickly add up.

We also have latency concerns regarding the propagation of updates. The discover, send, sleep cycle can take up to one minute. When waiting for an update to propagate across n links, this could take up to n minutes. For a developer designing a messaging application using our infrastructure, this latency becomes prohibitive.

**Multiple Applications**

Android services are singletons. So, if two applications are using our synchronization service, they are actually using the same service. Currently, all applications using our API would use the same database and service. This poses a security concern, as well as a latency concern. If an image storing application and a messaging application are both using the synchronization service, images are going to be broadcast even if the image storing application stops using the synchronization service for a time. This would slow down the lower latency messaging application.
Future work

Given the problems encountered with Bluetooth, we would port our infrastructure over to peer to peer wifi to make it more reliable as well as secure.

We would address our scalability concerns. First, instead of sending the entire database, we would only send the key value pairs that are actually needed. We would need to construct a protocol for phones to determine which pairs are needed. We could also develop some type of cache, where key value pairs that are updated and/or read from often are broadcast more often, while less accessed key value pairs are broadcast only periodically. We could also compress data to make data exchange quicker, as well as to save space locally.

Also, we could expand our API to support notifications when a key value pair is updated.

Lastly, we could do more testing on larger mesh networks to evaluate the latency of transmitting data across many links, as well as the feasibility of messaging applications.

References