UniTrace

An Exposure Tracing System for Universities

By: Miriam Rittenberg, Benjamin Steffen, Nina Gerszberg

{mrittenb, bds, ninager}@mit.edu

WRAP Instructor: Laura McKee

May 11th 2021

Introduction

Many deadly diseases are primarily spread through exposure to an infected person. Since people may find out that they are infected after already being contagious for some period of time, and may continue to contact other people even after testing positive, a system that tracks which people have been exposed to which other people, and who tests positive when, can help control the spread of disease. The goal for this project is to describe UniTrace, a contact tracing application that supports a university in identifying infection exposures and assists those who have been infected or exposed.

Designing a well-functioning exposure tracing system is difficult because it requires high usage rates by people in an area, requires people to frequently check whatever method of communication the system uses, and requires some method of detecting whether users are nearby. However, our system can avoid many of these issues, since it will be used in a university setup where everyone is assumed to have a phone and to have the app installed at all times.

Goals

The primary goal of UniTrace is to reduce infection. To accomplish this objective, the system must be useful and people must want to use it. Subgoals that can help achieve this overall goal are ease of use, preservation of user privacy, and providing timely and accurate responses. While still important, we put less priority on performance and providing information to researchers.

Fast notification of exposure events is one of the most important priorities, since every hour a person spends potentially infected but unaware of their exposure is an hour they may be spending infecting even more people. Similarly, ease of use is important, so that the users always see the notifications, rather than tuning out frequent emails or rarely checking the app. Since users are required to use the system, privacy is also an important consideration.

UniTrace achieves fast notification by requiring mandatory reporting of positive tests, sending out information on the positive test to every user soon after the Central Server is notified of a positive test, and having a small enough user base that checking the key pairs against contacts takes very little time.

We plan to achieve ease of use by focusing on usability when designing the app and testing the understandability of the UI before and during deployment.

The system preserves as much privacy as possible for individuals who do not test positive, and some privacy for those who do test positive. Information about who you have contacted never leaves your phone, and if you do not test positive neither the central server nor any of the other users are capable of determining your contacts. If you do test positive, anyone who comes within a short distance of you while you are supposed to be quarantining will be notified that they are near a person who has tested positive and should be quarantining. However, even if you test positive, people you have not contacted and do not contact during your quarantine period will not learn about your contacts.

Because UniTrace will be used by a university with only 20,000 people, rather than a whole country, performance is not a major issue. However, we still try to use only as much storage and bandwidth as is necessary. Also, while we hope to help researchers, that is a lower priority than preventing spread.

System Assumptions

- Users are tested every other day
- Users will notice within a few hours if the app notifies them of an exposure event
- The longest users go without their phones running our app is ten minutes
- BLE signals do not transmit through walls
- The system is used only by about 20,000 people
- We can require people to quarantine
- Users will not turn off their phones to prevent the app from transmitting information

System Overview

UniTrace consists of several components, each owned by a different party. The Smartphones are personal devices owned by the students of the university. Each phone has a personal Generator, a cryptographically secure pseudorandom number generator (CSPRNG) that it uses to generate IDs. The Smartphones exchange those IDs with other nearby Smartphones via Bluetooth Low Energy (BLE). The Routers and Central Server are devices managed by the university. The Routers forward traffic to and from the Smartphones. The Central Server stores current and historical Generators for each student, along with their status (safe, isolation, quarantine) and contact statistics. The Central Server also receives test results from medical facilities and uses those tests to determine which students need to quarantine. Using the test results, the Central Server collects the Generators belonging to infected community members, and sends them to the Smartphones for contact analysis. Finally, the Central Server also sends each Smartphone a new Generator daily. The above information is summarized in Figure 1 below.

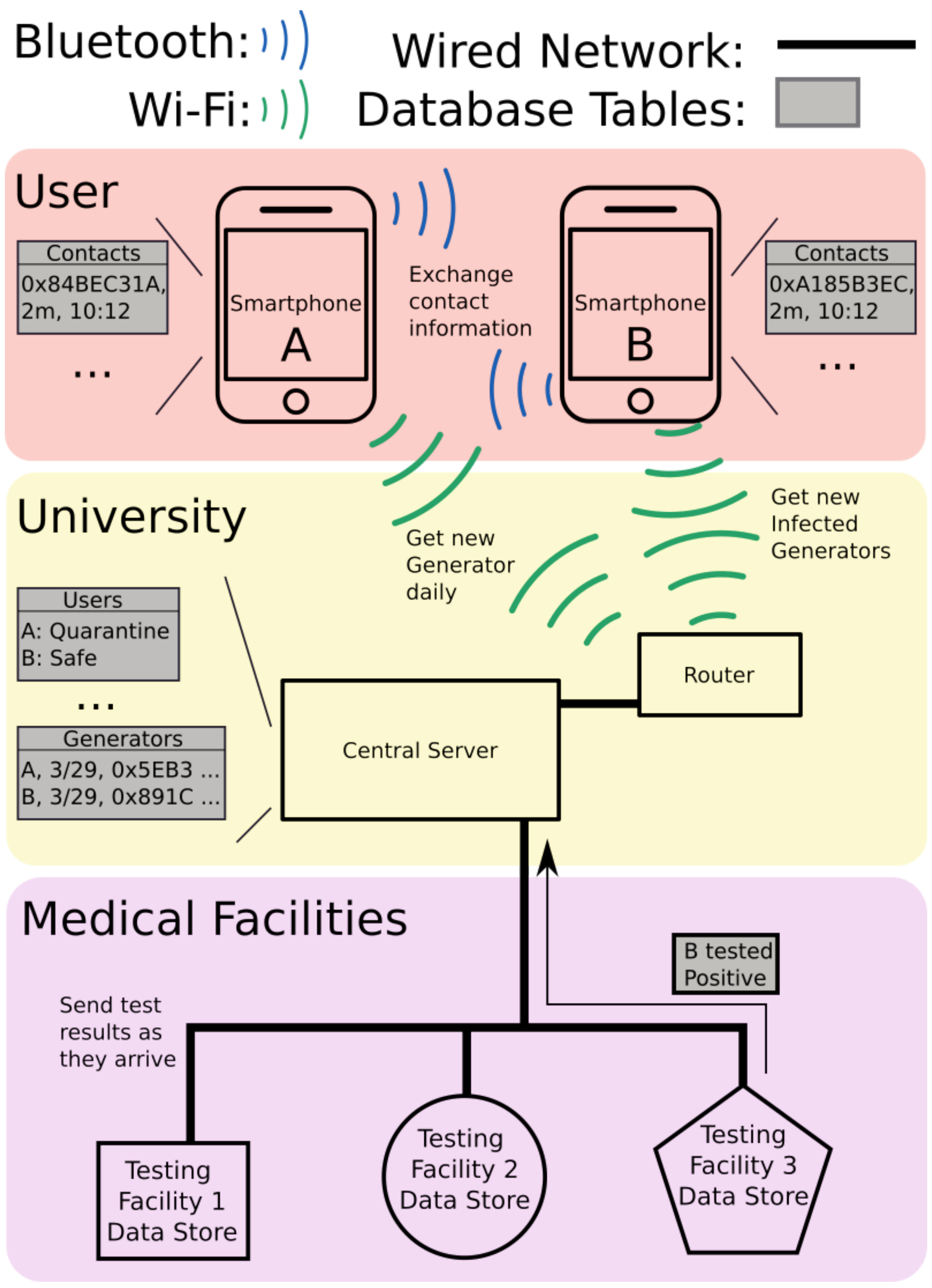

Figure 1: System overview
In this scenario, we have two users, A and B, who are in close contact with each other. They are in contact with the Central Server via the Router, which sends them the generators belonging to infected people that it has identified since their last contact, and also sends each Smartphone their new Generator daily. User B has taken a test at Testing Facility 3, which then informs the Central Server that B has tested positive.

System Description

The Smartphones

The Smartphones are handheld personal computing devices that the members of the university carry around with them as they go about their daily lives. We use a relational database for data storage. Relational databases are widely used and well understood, and have features providing strong reliability and durability of data storage. This reliability supports the system's ease of use by helping prevent the app from entering an inconsistent state in situations that cause it to suddenly shut down, such as the phone running out of battery or being dropped. Table 1 below lists the database tables stored on the smartphone.

Table Name	Contents
Contacts	Timestamp, Signal Strength, Sender ID
Generators	Date of Use, Generator State
Crashes	Timestamp, Sender ID, Exposure Number

Table 1: Smartphone Database

Every 15 minutes, the Smartphones use their Generator to create a new ID which they use in the exchange of contact information. Because the Generator is a CSPRNG, it is computationally infeasible to match different IDs to the same person. The use of a hash function supports our goal of privacy, as it makes deanonymizing an ID difficult.

Every 10 seconds, the Smartphone emits a BLE beacon which transmits the current ID to anyone in close proximity. Nearby Smartphones note this ID along with the signal strength of the message and the time it was received in their databases. We chose the interval of 10 seconds because it results in low storage requirements on the smartphone while also not dividing the 20 minute period that constitutes an exposure event into so few pieces that it becomes difficult to accurately determine.

Crash Response

Smartphones sometimes crash for up to ten minutes. When a Smartphone recovers from a crash, it does the following:

1. Determine how long the crash lasted for, say n seconds. The phone stores the time of its most recent broadcast and uses that to determine the length of the crash.
2. Send floor(n / 10) BLE messages at a rate of 4 per second. The Smartphone will send a maximum of 60 messages, which takes 15 seconds, so these messages will not cause a significant backlog.

3. Wait ten seconds. Then, for each ID from which it received a message within the ten seconds just before or after the crash, add an entry to the Crashes table with an Exposure Number equal to floor((n / 2) / 10). This is the number of messages that would have been received if the Smartphone remained in contact with that sender for half of the length of the crash.

There are a few important things to note about this response to crashes.

First, it is only an estimate of how much of the time during the crash was spent in contact with other people. This calculation could overestimate or underestimate the true length of exposure. We think that half of the time is a reasonable approximation, and since the maximum length of a crash is ten minutes, the most this estimate can be off by is five minutes, which is only a quarter of the cutoff for exposure.

Second, it is not symmetric. If two people are standing within range of each other when one person's phone recovers from an eight minute crash, and remain next to each other for at least another fifteen seconds, the crashed person's phone will estimate that the two people spent four additional minutes next to each other, while the other person's phone will estimate that the two people spent the entire eight minutes next to each other. On the other hand, if the overlap is at the start rather than the end of the crash, the crashed phone will calculate four minutes and the other person's will calculate zero. This scenario could mean, for example, that if person A tests positive then person B's phone will detect exposure, while if person B tests positive then person A's phone will not detect exposure. While this asymmetry is not ideal, avoiding it would require communication with the Central Server after crashes, and our decision to prioritize privacy leads us to keep as much of the contact calculations on the Smartphones themselves as possible. As with the above problem, the maximum difference caused by this algorithm is five minutes, so we believe that it will not be a large problem.

Generators and Determining Exposure

Every day at 4 a.m. with an additional random delay of up to 30 minutes, the Smartphone contacts the Central Server and downloads its new Generator for the day. The Smartphone loads this new Generator and uses it for all further IDs for the day. The system performs this action at 4 a.m. because most people are asleep at this time, causing the load on the network to be low. The early morning timing and the additional random delay spread the traffic to the server more evenly throughout time, increasing system performance.

Within every 30 minute block, the Smartphone queries the Central Server once to see if it has identified any new Generators as belonging to infected users (Infected Generators). The Central Server sends back any Infected Generators that it has identified since the Smartphone last contacted it so that the Smartphone can perform contact analysis using those Generators and its contact database. We chose 30 minutes as the interval to query the server because it represents an effective compromise between getting exposure results to users as quickly as possible while not overloading the Central Server with constant queries. The smartphone randomly chooses a time within the 30 minute block to spread what would be a large burst of traffic out over time, reducing the load on the network.

When a Smartphone receives Infected Generators, it runs the following algorithm:

1. Use the Generators belonging to infected people to generate all IDs representing such people.
2. Make a list of all messages from the Contact table that have an ID belonging to an infected person.
3. Count the total number of messages in this list.
4. Add the numbers corresponding to crashes and exposed IDs in the past 24 hours.
5. For each consecutive pair of messages associated with the same ID, if those messages were received
    a. 15-25 seconds apart: add 1
    b. 25-35 seconds apart: add 2
    c. 35-45 seconds apart: add 3
    d. 45-55 seconds apart: add 4
    e. 55-60 seconds apart: add 5
6. If the total count is at least 6 * 20 = 120, corresponding to an exposure time of at least 20 minutes, the user is deemed exposed.

Note that this algorithm treats all infected people as equivalent. 20 minutes of time with a single infected person counts the same as 10 minutes with two different infected people. We base these decisions on the CDC guidelines for determining COVID-19 exposure, but note that implementers can easily adjust this algorithm to use a number other than 120 for the cutoff point to handle diseases with different profiles.

When a message with an ID derived from an Infected Generator is received, the phone alerts the user that they have been in contact with a person who tested positive. The phones send these alerts in order to encourage compliance with quarantine, as they allow the people near the infected person to be aware that that person is breaking quarantine.

Communication With Central Server

The Smartphone also generates anonymized statistics on the collected data that are sent to the Central Server. The app generates four statistics:

1. How many total messages the user received that were from infected users
2. How many total messages the user received that were from uninfected users
3. A count of how many unique infected sender IDs the phone received
4. The standard deviation of the number of messages from each sender ID

These statistics were chosen because they preserve the user's privacy while also being useful to researchers. These statistics do not include data that can be used to positively identify who the user was in contact with except in degenerate cases such as only one person in the community being infected. We believe these statistics will be useful because they allow researchers to estimate things such as to what degree the infected and uninfected populations of campus are interacting, and whether the disease is primarily spread by a few bad actors who blatantly disobey isolation directives, or by large groups of people who only slightly disobey.

The Smartphones and the Central Server send 4 main messages to each other over the internet. These messages are listed below in Table 2, along with a description of what data they hold notated in JSON. The Infected Generators Request message deserves special attention, as it contains an index into the Central Server's list of Infected Generators that allows it to send only the Infected Generators that the Smartphone doesn't already know about, reducing load on the network.

New Generator
Central Server -> Smartphone
 	
{
  "date": <timestamp>,
  "generator_state": <byte string>
}
	Infected Generators Request
	Smartphone -> Central Server
	
	{
	  "up_to": <integer>
	}

Statistics
Smartphone -> Central Server
 	
{
  "id": <byte string>,
  "total_infected_message_count": <integer>,
  "total_uninfected_message_count": <integer>,
  "unique_infected_id_count": <integer>,
  "message_count_per_id_stddev": <integer>
}
	Infected Generators Response
	Central Server -> Smartphone
	
	{
	  "generators": [<byte string>]
	}

Table 2: Messages between Smartphones and Central Server

For sending the messages over the network, we use TCP as the transport layer protocol because of its reliability features. TCP allows us to simplify our system by not worrying about the need to potentially retransmit packets because they got lost or damaged in transit. For the serialization of these messages, we use the Compact Binary Object Notation (CBOR) format, specified in RFC 8949. This format allows encoding a superset of JSON values, importantly including the ability to serialize byte strings directly without using, for example, Base64 to turn the data into text. This feature is particularly important for UniTrace, as it transmits a large amount of binary data as part of its Infected Generators Response message, so using a binary format allows for reduced communication overhead.

The Central Server

The Central Server is a well-equipped datacenter-class computer that the university owns. The university has given exclusive use of the computer to the contact tracing system. The Central Server stores general information on the students, such as their name, phone number, living situation, and classes in a separate database. We explicitly consider only the information directly relevant to contact tracing, such as their current status (safe, isolation, quarantine) and the date any transitions between different statuses occurred, and their Generators. The Central

Server also receives and stores the contact statistics the Smartphones generate. For similar reasons as the Smartphone, we again use a relational database to store the tracing information. In addition to the previously discussed properties of reliability and durability, much work has gone into making relational database scale well in response to highly concurrent access. This scalability is important for UniTrace as there will be many Smartphones accessing this data throughout the day, so having the database scale well is a priority. The database tables are shown below in Table 3.

Table Name	Contents
Users	User ID, Current Status (Safe, Isolating, Quarantine)
Transitions	User ID, Timestamp, Old Status, New Status, Cause (Exposure tracing, Living group infected, In-class infected, Positive test)
Generators	User ID, Date of Use, Generator State
Statistics	User ID, Date, Total Infected Message Count, Total Uninfected Message Count, Unique Infected ID Count, Standard Deviation of Message Count per Infected ID

Table 3: Central Server Database

In order to support user privacy, the Central Server stores little more than what is required by university policy. In particular, the Central Server does not store enough data to deduce exactly who different users have been in contact with.

The Central Server does, however, store enough information to get a rough estimate of how much time the users have spent around other people. To encourage compliance with isolation and quarantine directives, this data can be used by the university to warn users who are breaking their quarantine or isolation.

The Central Server also interfaces with various medical testing facilities to ingest any relevant test results the facility has performed and integrate them into the system. The medical facility needs to provide only positive test results along with a timestamp of when the test was taken and the minimum amount of information necessary to uniquely identify a student. This information is sufficient for the system to update any relevant user statuses. To support user privacy, the system does not permanently store any information from the test results, save for any transitions between user statuses as a result of the test.

When the Central Server receives a report of a positive test, it maps the test to a student, then changes their status to ‘quarantine.’ While UniTrace is designed to be a contact tracing solution for many potential diseases, we follow current research on COVID-19 that suggests that people may be contagious up to 48 hours before a positive test (Harvard Health Publishing, 2020). In accordance with this research, the system grabs the user’s current Generator, along with their

Generators for the previous two days and puts them into the list of Infected Generators. This threshold can be easily adjusted to fit the profile of different diseases. Every time the Central Server generates a new set of Generators for every user, it will add the Generator of any user currently marked as ‘quarantine’ to the list of Infected Generators. This list of Infected Generators gets sent out to the Smartphones as they contact the Central Server throughout the day.

In addition, the Central Server checks the user’s classes and living group and sends a message to the Smartphones of all people who share a class or living group with an infected student notifying them that they have been exposed. If it does not receive confirmation of receiving the message from a Smartphone, it attempts to contact that Smartphone again ten minutes later.

The Routers

The routers serve only their normal function of forwarding traffic along the network. We did not find any usages of this module that would help the system function while also preserving user privacy.

Events

To summarize the above, here is how UniTrace responds to the different events that can occur in the system:

New User joins UniTrace

To join UniTrace, users must download the contact tracing app and log into the app using their university login. After that, the central server generates a Generator, saves it, and sends it to the user’s smartphone. Users will be able to easily access and use this app on their smartphones as per our “ease of use” goal above.

User passes by another app user

Every smartphone generates a BLE beacon once every ten seconds containing the current ID. When a user’s phone receives a beacon from another app user, it notes the ID and time and saves this data locally. User privacy is ensured here as all interaction data is stored via ID’s on smartphones as opposed to a central server.

User has a positive test

First, the testing center notifies the central server that one of its users had a positive test. Next the server looks through the database of Generators and finds all Generators associated with the infected user from the past two days. The smartphones query the server about every hour to get the new list of infected Generators. Users therefore receive timely information about exposures.

User’s smartphone receives a list of infected Generators

The smartphone generates a list of IDs that each Generator can create. If IDs associated with one of the Infected Generators are seen for at least twenty minutes, the user is notified of an exposure event. Comparing the received list of infected ID’s to the ID’s that the user interacted with ensures that the user will be informed of an exposure event in a timely and accurate fashion.

New Day

Every day, the server generates and distributes a new Generator for each user.

Evaluation

Given that UniTrace is a large system designed to accommodate up to 20,000 people, it is important to ensure that the system can still run well under reasonable hardware constraints. Additionally, since we incorporate user smartphones into our design which may have varying capabilities, we must carefully analyze the workloads on the smartphones. We aim for the UniTrace app to have relatively low requirements, as having high requirements presents an equity issue: those without new and expensive smartphones could not keep themselves healthy as effectively as those who do.

Throughout the evaluation, we make the assumption that storing a piece of data in a database will at most triple its size. We believe this assumption is reasonable because in the simulation of the contact database detailed below in the Processing section, the data is only slightly more than doubled in size (181MB). This is also a fairly pessimistic assumption because the simulation constructs an index on the sender ID, which is very large compared to the other pieces of data, causing the index to nearly double the size of the database.

We also make the assumption that transmitting a piece of data over a network will at most double its size. We believe this assumption is reasonable because the serialization format we chose, CBOR, is a binary format. At the cost of human-readability, binary formats allow for lower overhead in size due to directly encoding values in binary form rather than text.

We also assume the sizes of several data elements. In order to preserve user privacy, the Generator must be a cryptographically-secure pseudorandom number generator. The ChaCha20 stream cipher can be used to create such a number generator and has an internal state with a size of 512 bytes. We assume that this number is a reasonable maximum value for the size of a Generator. We assume that 64 bytes is sufficient for storing a User ID or an ID from a BLE broadcast. We assume that all other elements, such as timestamps and counts for the various statistics the system calculates, can be held in 8 bytes.

Storage - Smartphone

First, we consider the storage requirements on the smartphone. As stated in the DP, BLE messages are 70 bytes long (DP Page 12), equalling 140 bytes per message on disk. We assume that in the worst case, a person will be in contact with 10 people for the whole day. Since our system sends a message every 10 seconds, this scenario results in 86,400 messages per day. The university policy mandates that contact data be stored for 2 weeks (DP Page 12), resulting in 1,209,600 total messages. Multiplying this number by the size of the message in bytes, we find that we expect to use a maximum of 254MB of storage for storing contact data on the smartphones.

Next, we consider the storage of Infected Generators. With the addition of the timestamp, we estimate that each Infected Generator stored would take 1560 bytes on disk. We further assume that in the worst case, half of the population of the campus is infected at the same time. Given these assumptions, the smartphone will receive 10,000 Infected Generators daily. We assume that retaining the Infected Generators for a month is sufficient for tracing contact. With our assumption about on-disk storage overhead, we calculate that a maximum of about 468MB would be used for storage of Infected Generators.

Finally, we consider the storage of crash data. Making our assumption about storage overhead on disk, each crash should take at most 240 bytes of storage. We assume that each phone crashes at most twice a day. As crash data is part of the contact data, we must store it for 2 weeks. In total, the storage requirements for crash data amounts to 6.7KB.

In total, we expect each smartphone to need to store about 716MB of data. This amount is below the limit of 1GB given in the DP (DP Page 12), so lack of storage space on the smartphone is unlikely to be an issue.

Storage - Central Server

The Central Server stores several tables of data, which we will examine in order. As a reminder, we again make the assumption that storage in a database at most triples the storage requirements.

The *Users* table contains a User ID and a status, totalling to 72 bytes. Since the system will serve a population of 20,000, this amount of storage per user results in a total of 4.3MB for this table.

The *Transitions* table contains a User ID, a timestamp, the user's previous and new statuses, and the cause of the transition, totalling to 88 bytes. We assume that in the worst case, every user will undergo a status change daily. This scenario results in 5.28MB of data per day. This data needs to be kept for 180 days (DP Page 12), resulting in a total of 951MB for this table.

The *Generators* table contains a User ID, a timestamp, and the state of a Generator, totalling to 584 bytes. The system creates a new Generator daily for each user, totalling to 35MB per day. Each generator is kept for 2 weeks, as it is a part of contact logging information, resulting in a total of 491MB for this table.

The *Statistics* table contains a User ID, a timestamp, and 4 statistics, totalling to 104 bytes. Each user sends statistics every day, resulting in 6.24MB per day. Given the long-term research use for this data and lack of privacy concerns, we assume that we will store this data for 5 years. These assumptions result in a total of 11.23GB of data.

Over all tables, we expect to store about 11.73GB of data. This number is well within the 12TB limit posed by the DP spec (DP Page 11).

Bandwidth

UniTrace relies on a connection to the Central Server for delivering timely reports of contact with infected persons. This section makes all of the same assumptions about the size of various pieces of data as the previous section. As stated in the introduction, we assume that transmitting a piece of data requires no more bandwidth than double the size of the data. Throughout this section, we consider the worst case scenario where every smartphone is contacted through a single choke point router, limiting the network speed to 2.5Gb/s (DP Page 11).

We consider separately the three types of messages that the system sends.

Every day, Smartphones contact the Central Server for their new Generator. Because the Generator state is 512 bytes, we assume that it takes no more than 1024 bytes to send it over the network. Each one of the 20,000 users needs a new Generator daily, so this message takes 20.48MB daily. In our worst case scenario, we calculate that it would take 0.065 seconds to distribute all of the new Generators to every member of campus. In practice, this number is so small that network latency and the TCP handshake would need to be considered to get an accurate number, but we regardless hope this number illustrates that the network is capable of handling this message type easily.

In addition to a new Generator, each Smartphone sends the Central Server its contact statistics for the past day. These statistics total to 72 bytes, and we assume it takes 144 bytes over the network. In our worst case scenario, for every member of campus to send their statistics at the same time, we estimate that it would take 0.0011 seconds to send. Again, this number is too small to be considered an accurate representation of the time in practice, but it does again show that the given bandwidth is certainly sufficient for handling this message.

Throughout the day, Smartphones also contact the Central Server for updates on the Infected Generators. For this, we assume that in the worst case, half of the campus is infected. We again estimate that each Generator takes 1024 bytes to send over the network. If half of the campus population is infected, this comes to a total of 10.24MB of data. We need to distribute all of this data to every Smartphone, which results in 204.8GB of data to send in total. Making the same worst case assumption about the network, this data would take approximately 11 minutes to send if it all needed to be sent out at once. However, UniTrace has the Smartphones stagger their requests randomly over 30-minute blocks of time. With this in mind, we expect the network to be at about 33% utilization just from UniTrace data in this worst case scenario, thus we expect that no Smartphones will experience a delay in getting the newest list of Infected Generators.

Processing

Evaluating the performance of software numerically is extremely difficult given the nearly endless list of complex performance optimizations both on the hardware and software layers. Thus, for evaluating the processing requirements we place on the Smartphone, we construct a simulation of an expected workload on a low-power microcomputer we expect to perform similarly to the average smartphone.

UniTrace uses a relational database to store all of its data. For this simulation, we choose to use SQLite. Among its many other use cases, it is well-known to be suited for embedded use-cases due to its small size and durable data storage, and as such is widely used in smartphone apps for storing data.

While we anticipate that most of the processing work will be done within the database engine, the speed of the language that the app is written in can also have an impact on the performance of the app overall. For this simulation, we choose to use Python to interact with the database. Common smartphone app programming languages such as Java and Kotlin on the Android side, and Objective-C and Swift on the Apple side, are either bytecode-interpreted or compiled to native code. Python is similar to Java and Kotlin in that it is bytecode interpreted, however given the highly dynamic nature of Python, we expect that it will be slower than all common smartphone app programming languages.

Finally, for the computing platform to perform the simulation on, we choose to use a Raspberry Pi 3 Model B. Raspberry Pis are low-power computing devices designed for educational use. They are quite cheap (A Raspberry Pi 3 currently sells for approximately 40 US dollars), and as such, they are not particularly performant. We thus expect that a Raspberry Pi microcomputer can serve as an appropriate proxy for the average smartphone for testing processing requirements.

In the simulation, we construct a simple database containing one table having three columns: a timestamp, a signal strength, and a sender ID. We additionally create an index on the sender ID in order to increase the performance of queries on this column. As detailed above in the Storage section, we expect each Smartphone to receive 86,400 messages per day. The Smartphone needs to save these messages for 14 days, resulting in 1,209,600 total messages in the database. The simulation inserts that many messages, each having a random timestamp, signal strength, and sender ID. With the sample database generated, we measure the performance of looking up a series of sender IDs in the database. Each Generator produces a total of 96 IDs throughout the day, so the simulation creates that many random IDs and queries the database for any messages from one of these IDs.

After running the simulation we find that checking an Infected Generator against the contact database in our simulation takes a mean of 5.77ms, with a standard deviation of 0.17ms. As UniTrace contacts the Central Server once in every block of 30 minutes, we believe that the time to check Infected Generators against the database will not pose an obstacle to the successful operation of the system, nor cause annoyance to the users.

Upholding Design Properties

The design of UniTrace seeks to create three desirable system properties: timeliness of results, reliability, and privacy.

UniTrace contacts the Central Server once in every 30-minute block to retrieve a new list of Infected Generators. This means that, assuming a working network and Central Server, a user will receive notice that they should isolate no more than an hour after a positive test enters the system. In the section above dealing with network bandwidth, we show that network congestion is unlikely to be a problem holding up the delivery of Infected Generators.

The core functionality of UniTrace relies only minimally on modules other than the Smartphones. A temporarily nonfunctional network does not cause the system to completely stop functioning or destroy data. It only impedes the notification of Infected Generators and the sending and receipt of new daily Generators and statistics. Contact between individuals is still recorded as usual, and the Smartphone is still able to check any newly received IDs against previously any Infected Generators previously downloaded from the Central Server.

Finally, UniTrace also preserves a high degree of privacy for the users of the system. Each user only knows the IDs of the people who passed by them. Users cannot reverse these IDs to names of people, nor can they link any IDs to another broadcasted by the same person. In addition, while the operators of the Central Server are able to determine which Generators belong to a certain person, they cannot determine which users have come in contact with which other users. This is because the Smartphones never transmit contact data to the Central Server. This careful separation of information ensures that who a user comes into contact with remains private. Moreover, UniTrace does not collect or store any kind of location data whatsoever, which ensures that users cannot have their location tracked via UniTrace.

Response to Changing Conditions

UniTrace is flexible to changing conditions owing to how it determines contact entirely on the Smartphone. As an example, if new research is published stating that 30 minutes should be the threshold for exposure instead of 20 minutes, a simple app update is sufficient to change the algorithm in light of this new information. The system handles such changes gracefully, as users who haven't upgraded the app simply have slightly less accurate contact determination without affecting other users at all.

Community Members Without Cellphones

While cell phone ownership is almost universal, there are many valid reasons a person might choose to not own a cell phone, such as cost and privacy concerns. The explicitly public nature of a contact tracing system necessitates consideration of minority groups. Community members without a cell phone would not be able to participate in contact tracing, as there is nothing to broadcast BLE signals or contact the Central Server to receive Infected Generators. If the number of users without a cell phone is small, it is likely that the system's functionality would not be significantly impaired.

However, if the number of users without a cell phone is not insignificant, then the system could provide much less accurate and reliable results to the users that do have them, and additionally not assist users without a cell phone in keeping healthy. A potential solution to this is enabled by UniTrace's relatively low requirements: using a microcomputer to run the contact tracing system on a standalone device. This neatly solves both the cost and privacy concerns. There are small microcomputers with BLE and Wi-Fi support such as the Raspberry Pi 4 that cost under $40 individually and would be sufficient to run the contact tracing algorithms. Such microcomputers could be cheaply fitted to run the contact tracing algorithm like a smartphone would. They also run primarily free and open-source software, thus easing any privacy concerns. We believe this option would appeal to most people who don't own a cell phone, and ensures equitable access to resources for staying healthy among the campus community.

Conclusion

UniTrace will help universities track and reduce the spread of infectious disease. Our system preserves as much user privacy as possible, without sacrificing the contact tracing functionality. By checking for positive tests every half hour, UniTrace does not spend too much time checking and finding no updates when community infection rates are low, but still notifies people quickly when a positive test does occur. When test rates are high, the system can still handle checking every Infected Generator against its contacts, since a university has a relatively small number of people.

Author Contributions

We discussed our design and made all the design decisions together as a group on zoom calls. Below describes how we divided writing up these decisions in this paper.

Nina made the events section. She helped out with the evaluation section, author contribution section, works cited section, and the introduction. She also helped respond to some of the feedback given on the DPPR by modifying this paper accordingly. Nina also wrote up the acknowledgements, helped with the editing, and did a lot of the formatting (such as the title page).

Miriam wrote part of the introduction and most of the goals and assumptions section. She added the descriptions of the response to crashes and of the more detailed exposure determination algorithm. She also helped with the editing.

Ben wrote most of the system description section and the evaluation section. He created the figure shown in the system overview section, and wrote the simulation of the contacts database. He also helped with editing and formatting the paper.

Acknowledgements

We would like to thank Katrina LaCurts and the 6.033 course staff for all their wonderful work during this unusual year. We further would like to thank our writing instructor Laura McKee, and our technical reviewer Mohammad Alizadeh for their valuable feedback.

Works Cited

CDC. (2020, February 11). *Coronavirus disease 2019 (COVID-19)*. Centers for Disease Control and Prevention. Retrieved March 29, 2021, from https://www.cdc.gov/coronavirus/2019-ncov/hcp/guidance-risk-assesment-hcp.html

Harvard Health Publishing. (2020, March). *If you've been exposed to the coronavirus*. Harvard Health. Retrieved March 29, 2021, from https://www.health.harvard.edu/diseases-and-conditions/if-youve-been-exposed-to-the-coronavirus

Ayala, Inmaculada & Pinilla, Mercedes & Fuentes, Lidia. (2019). An Energy Efficiency Study of Web-Based Communication in Android Phones. Scientific Programming. 2019. 1-19. 10.1155/2019/8235458.
https://www.hindawi.com/journals/sp/2019/8235458/

Appendix A: Database Test Code

```
import sqlite3
import random
import string
import timeit
import statistics

IDS_PER_GENERATOR = 96

MESSAGES_PER_DAY = 24 * 60 * 60
RETENTION_DAYS = 14
MAX_CONTACTS = MESSAGES_PER_DAY * RETENTION_DAYS

con = sqlite3.connect('contacts.db')
cur = con.cursor()

cur.execute('''CREATE TABLE contacts (timestamp INTEGER, signal_strength REAL, sender_id VARCHAR(64));''')

cur.execute('''CREATE INDEX sender_id_index ON contacts(sender_id);''')

for _ in range(MAX_CONTACTS):
        timestamp = random.randrange(2**32)
        signal_strength = random.randrange(100)
        sender_id = ''.join(random.choice(string.ascii_letters) for _ in range(64))
        cur.execute('''INSERT INTO contacts VALUES (?, ?, ?);''', (timestamp, signal_strength, sender_id))

con.commit()
print('database generated')

cur.execute('vacuum;')
con.commit()
print('vacuum completed')


def infected_generator_received():
        generator_ids = [random.randrange(2**32) for _ in range(IDS_PER_GENERATOR)]
        contacted = {}
        for generator_id in generator_ids:
        for _ in cur.execute('''SELECT * FROM contacts WHERE sender_id = ?;''', (generator_id,)):
        contacted.setdefault(generator_id, 0)
        contacted[generator_id] += 1

times = timeit.repeat(infected_generator_received, repeat=10000, number=1)

print(f'mean   = {statistics.mean(times)}')
print(f'median = {statistics.median(times)}')
print(f'stddev = {statistics.stdev(times)}')

con.close()
```