SHARETRACE: CONTACT TRACING WITH THE ACTOR MODEL

by

RYAN TATTON

Submitted to the Department of Computer and Data Sciences

in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Science

at the

CASE WESTERN RESERVE UNIVERSITY

August 2023

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Ryan Tatton

candidate for the degree of

Master of Science in Computer and Information Science

Committee Chair

Erman Ayday, Ph.D.

Committee Members

Youngjin Yoo, Ph.D.

Harold Connamacher, Ph.D.

Michael Lewicki, Ph.D.

Date of Defense

TBD

We also certify that written approval has been obtained for any proprietary material contained therein.

Contents

List of Tables

List of Figures

Acknowledgments

I am sincerely thankful to Dr. Erman Ayday and Dr. Youngjin Yoo for their mentorship and collaboration on the ShareTrace project and the various xLab endeavors. As a result of their generous support, I have learned about the challenges of distributed computing and the exciting prospects of extending self-sovereign identity to data-driven applications. I believe that my experiences with Dr. Ayday and Dr. Yoo have significantly contributed to my first career opportunity at Amazon; and for that, I cannot thank them enough.

Research reported in this paper was partly supported by the National Science Foundation (NSF) under grant number NSF CCF 2200255 and Cisco Research University Funding grant number 2800379. This work made use of the High Performance Computing Resource in the Core Facility for Advanced Research Computing at Case Western Reserve University.

SHARETRACE: CONTACT TRACING WITH THE ACTOR MODEL

by

RYAN TATTON

Abstract

Proximity-based contact tracing relies on mobile-device interaction to estimate the spread of disease. ShareTrace is one such approach that improves the efficacy of tracking disease spread by considering direct and indirect forms of contact. In this work, we utilize the actor model to provide an efficient and scalable formulation of ShareTrace with asynchronous, concurrent message passing on a temporal contact network. We also introduce message reachability, an extension of temporal reachability that accounts for network topology and message-passing semantics. Our evaluation on both synthetic and real-world contact networks indicates that correct parameter values optimize for algorithmic accuracy and efficiency. In addition, we demonstrate that message reachability can accurately estimate the risk a user poses to their contacts.

Chapter 1

Introduction

Since the beginning of the COVID-19 pandemic, there has been a copious amount of research in mobile contact tracing solutions, most notably being the joint effort by Apple and Google [8]. External reviews and surveys provide extensive comparison of existing solutions through the lenses of privacy, security, ethics, adversarial models, data management, scalability, interoperability, and more. References [6] and [63] provide thorough reviews of existing mobile contact tracing solutions with discussion of the techniques, privacy, security, and adversarial models. The former offers additional detail on the system architecture (i.e., centralized, decentralized, and hybrid), data management, and user concerns of existing solutions. Other notable reviews with similar discussion include [21, 26, 59, 72, 81]. Reference [50] provides a formal framework for defining aspects of privacy for proximity-based contact tracing.

A number of online surveys have been conducted that examine user preferences of different aspects of contact tracing [7, 56, 75]. A common finding

across these surveys is that privacy and security continue to be of top concern for users, but contains some interesting nuance. For example, [7] surveyed over 10,000 individuals and found that there was over a 60-percent willingness to install a contact tracing mobile application. In a longitudinal study, [75] found that user preferences regarding privacy were stable over time. Moreover, they found that users had fewer privacy concerns for proximity-based contact tracing, in comparison to location-based contact tracing, but that there was security concerns for proximity-based tracking. Contrary to much of the developed techniques that emphasize a decentralized approach, [56] observed that mobile contact tracing applications that implement a centralized design are significantly more likely to be installed at the country level. Additionally, they found that individuals are generally more comfortable with their location data and identity information accessible to health-, state-, and federal-level authorities, compared to application developers, and the general public.

A common design element across all of the aforementioned contact tracing methodologies is that they only consider direct interactions between users. While there are privacy benefits to this approach, a major limitation is that they cannot utilize information about indirect contact to more effectively reduce the spread of disease. ShareTrace addresses this limitation by constructing a factor graph and estimating infection risk via a message-passing algorithm. As such, this work labels the ShareTrace algorithm as *risk propagation*. The first work on ShareTrace was a white paper that focused on the motivation, design, and engineering details. Exclusive to [9] is a discussion on privacy, network roaming, protocol interoperability, and the usage of geolocation data.

Furthermore, it includes detail on the system model and data flow. The second work on ShareTrace [10] formalizes the algorithmic details in a centralized setting and demonstrates its improved efficacy, compared to the framework developed by Apple and Google [8].

This work improves the efficiency of risk propagation and provides a concurrent, distributed, online, and non-iterative formulation using the actor model. To quantify the communication complexity of this new design, this work defines message reachability to account for the dynamics of message passing on a temporal network.

The evaluation of risk propagation in this work entails: (1) the efficiency of risk propagation on both synthetic and real-world contact networks; (2) the scalability of risk propagation on synthetic contact networks; and (3) the accuracy of message reachability on synthetic and real-world networks. To keep the scope of this work focused, we defer to [10] on the privacy and security aspects of ShareTrace.

While message passing has been studied under specific epidemiological models [44, 55], our formulation allows us to contextualize risk propagation as a novel usage of a contact network that does not require such assumptions to infer the transmission of disease. As a result, we introduce a form of reachability that can uniquely characterize the dynamics of message passing on a temporal graph. Our formulation of risk propagation aligns with its distributed extension, as introduced by [10], which has connections to the actor model of concurrent computing [**?** **?**] and the "think-like-a-vertex" model of graph algorithms [64].

1.1 Actor Model

Chapter 2

Actor-Based Contact Tracing

In prior ShareTrace work [10], risk propagation was described under the assumption that

After outlining the system model, the original description of risk propagation is presented as both context and motivation for this work. Following this discussion is the main part of the chapter, which frames risk propagation as an online algorithm and an application of the actor model. Lastly, message reachability is introduced as a generalization of temporal reachability that can estimate the complexity of message-passing on temporal networks.

Refer to Appendix C for the typographical conventions used in this chapter.

2.1 System Model

The system model is similar to that in previous work [9, 10]. **??** illustrates the corresponding data flow[1].

- Each user owns a *personal data store* (PDS), a form of cloud storage that empowers the user with ownership and access control over their data.

- Symptom scores are computed in a user's PDS to support integrating multiple streams of personal data [9]. While local symptom-score computation [9, 10] is more privacy-preserving, it is assumed that the user's PDS is a trusted entity.

- User device interactions serve as a proxy for proximal human interactions. This work does not assume a specific protocol, but does assume that the protocol can approximate the duration of contact with relative accuracy and that communication with the actors of those contacted users can be established in a privacy-preserving manner.

- No geolocation data is collected [9]. As a decentralized, proximity-based solution, it is not necessary to collect user geolocation data. See Appendix A.6 for a discussion of a geolocation-based design that was considered.

- Actor-based risk propagation is a distributed, online algorithm [22, pp. 791–818]. Previous work [9, 10] (see also Appendix A) formulates risk

[1] A *data-flow diagram* consists of data processors (circles), directed data flow (arrows), data stores (parallel lines), and external entities (rectangles) [32, pp. 437–438].

propagation as a centralized, offline algorithm that periodically aggregates all user data to estimate infection risk. To improve the privacy, scalability, and responsiveness of ShareTrace, this work designs risk propagation to avoid data aggregation and to estimate infection risk in near real-time.

2.2 Global Risk Propagation

In risk propagation, computing infection risk is an inference problem in which the objective is to estimate a user's *marginal posterior infection probability* (MPIP). The prior infection probability of a user is derived from their symptoms [65], so it is called a *symptom score*. Because the posterior infection probability accounts for direct and indirect contact with other users[2], it is called an *exposure score*. In general, a *risk score* s_t is a timestamped infection probability where $s \in [0, 1]$ and $t \in \mathbb{N}$ is the time of its computation.

Computing the full joint probability distribution is intractable as it scales exponentially with the number of users. To circumvent this challenge, risk propagation uses a variation of message passing on a factor graph. Formally, let $G = (V, F, E)$ be a factor graph where V is the set of variable vertices, F is the set of factor vertices, and E is the set of edges incident between them [49]. A *variable vertex* v_i is a random variable such that $v_i : \Omega \to \{0, 1\}$, where the

[2]While previous work on ShareTrace [10] defines a *contact* as a contiguous 15 minutes in which the devices of two users are proximal, the Centers for Disease Control and Prevention technically considers these 15 minutes *over a 24-hour time window* [20].

sample space is $\Omega = \{healthy, infected\}$ and

$$v_i(\omega) = \begin{cases} 0 & \text{if } \omega = healthy \\ 1 & \text{if } \omega = infected. \end{cases}$$

Thus, $p_t(v_i) = s_t$ is a risk score of user i. A *factor vertex* $f(v_i, v_j) = f_{ij}$ represents contact between users i, j such that f_{ij} is adjacent to v_i, v_j. Figure 2.1 depicts a factor graph that reflects the problem constraints. While the max-sum algorithm aims to maximize the *joint* distribution [13, pp. 411–415], risk propagation aims to maximize *individual* MPIPs [10].

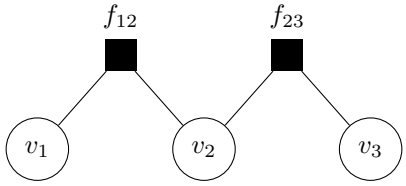

Figure 2.1: A factor graph of 3 variable vertices and 2 factor vertices.

2.3 Local Risk Propagation

The only purpose of a factor vertex is to compute and relay messages between variable vertices. Thus, one-mode projection onto the variable vertices can be applied such that variable vertices $v_i, v_j \in V$ are adjacent if the factor vertex $f_{ij} \in F$ exists [82]. Figure 2.3 shows the modified topology. To send a message to variable vertex v_i, variable vertex v_j applies the computation associated with the factor vertex f_{ij}. This modification differs from the distributed extension of risk propagation [10] in that we do not create duplicate factor vertices and

GLOBAL-RISK-PROPAGATION(S, C)

1: $(V, F, E) \leftarrow$ FACTOR-GRAPH(C)
2: $n \leftarrow 1$
3: $\delta_{n-1} \leftarrow \infty$
4: **for each** $v_i \in V$
5: $\quad R_i^{(n-1)} \leftarrow$ top K of S_i
6: **while** $n \leq N$ **and** $\delta_{n-1} > \delta_{min}$
7: $\quad$ **for each** $\{v_i, f_{ij}\} \in E$
8: $\quad\quad m_{v_i \to f_{ij}}^{(n)} \leftarrow R_i^{(n-1)} \setminus \{\, m_{f_{ij} \to v_i}^{(k)} \mid k \in [1\,..\,n-1] \,\}$
9: $\quad$ **for each** $\{v_i, f_{ij}\} \in E$
10: $\quad\quad m_{f_{ij} \to v_j}^{(n)} \leftarrow \max(\{0\} \cup \{\, \alpha \cdot s_t \mid s_t \in m_{v_i \to f_{ij}}^{(n)}, t \leq t_{ij} + \beta \,\})$
11: $\quad$ **for each** $v_i \in V$
12: $\quad\quad R_i^{(n)} \leftarrow$ top K of $\{\, m_{f_{ij} \to v_i}^{(k)} \mid k \in [1\,..\,n], f_{ij} \in N_i \,\}$
13: $\quad \delta_n \leftarrow 0$
14: $\quad$ **for each** $v_i \in V$
15: $\quad\quad \delta_n \leftarrow \delta_n + \max R_i^{(n)} - \max R_i^{(n-1)}$
16: $\quad n \leftarrow n + 1$
17: **return** $\{\, \max R_i^{(n)} \mid v_i \in V \,\}$

messages in each user's PDS. By storing the contact time between users on the edge incident to their variable vertices, this modified topology is identical to the *contact-sequence* representation of a *contact network*, a kind of *time-varying* or *temporal network* in which a vertex represents a person and an edge indicates that two persons came in contact:

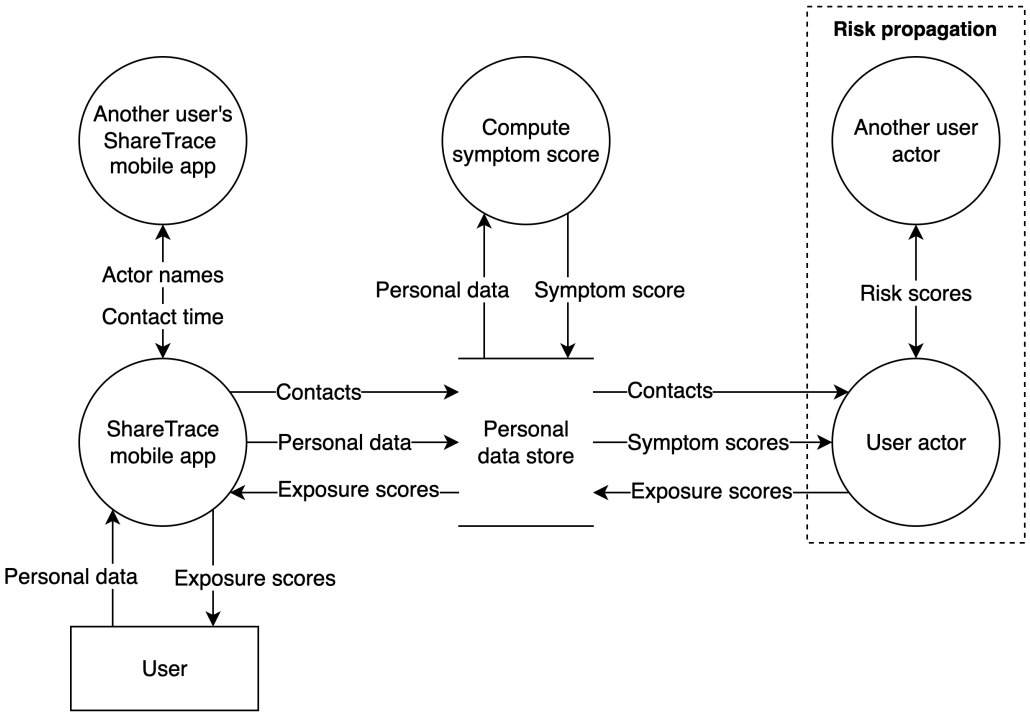

Figure 2.2: Distributed ShareTrace data flow. *Contacts* include the actor name and contact time of all users with which the user came into close proximity. *Personal data* includes the user's demographics, reported symptoms, and diagnosis. It may also include machine-generated biomarkers and electronic health record data [9].

$$\{\,(v_i, v_j, t) \mid v_i, v_j \in V; v_i \neq v_j; t \in \mathbb{N}\,\}, \tag{2.1}$$

where a triple (v_i, v_j, t) is called a *contact* [40]. Specific to risk propagation, t is the time at which users u and v *most recently* came in contact (see Section 2.3).

The usage of a temporal network in this work differs from its typical usage in epidemiology which focuses on modeling and analyzing the spreading dynamics of disease [23, 25, 48, 57, 71, 73, 83]. In contrast, this work uses a temporal network to infer a user's MPPI. As a result, Section 2.4 extends

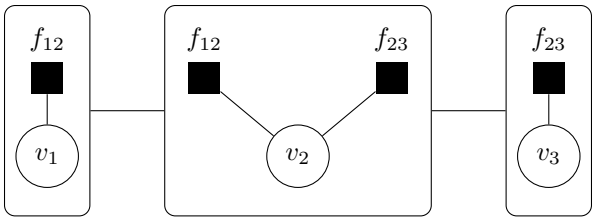

Figure 2.3: One-mode projection onto variable vertices of Figure 2.1. While the projected graph contains only variable vertices, the original factor vertices are also included to show how the original topology is modified.

temporal reachability to account for both the message-passing semantics and temporal dynamics of the network. As noted by [40], the transmission graph provided by [73] "cannot handle edges where one vertex manages to not catch the disease." Notably, the usage of a temporal network in this work allows for such cases by modeling the possibility of infection as a continuous outcome. Factor graphs are useful for decomposing complex probability distributions and allowing for efficient inference algorithms.

However, as with risk propagation, and generally any application of a factor graph in which the variable vertices represent entities of interest (i.e., of which the marginal probability of a variable is desired), applying one-mode projection is a .

2.3.1 ShareTrace Actor System

As a distributed algorithm, risk propagation is specified from the perspective of an *actor*, which is equivalent to the paradiagm of *think- like-a-vertex* graph-processing algorithms [64]. Some variation exists on exactly how actor behavior is defined [3, 5, 27]. Perhaps the simplest definition is that the *behav-*

ior of an actor is both its *interface* (i.e., the types of messages it can receive) and *state* (i.e., the internal data it uses to process messages) [27]. An *actor system*[3] is defined as the set of actors it contains and the set of unprocessed messages[4] in the actor mailboxes. An expanded definition of an actor system also includes a *local states function* that maps mail addresses to behaviors, the set of *receptionist actors* that can receive communication that is external to the actor system, and the set of *external actors* that exist outside of the actor system [3, 5]. Practically, a local states function is unnecessary to specify, so the narrower definition of an actor system is used. The remainder of this section describes the components of the ShareTrace actor system.

2.3.1.1 User Actor Behavior

Each user corresponds to an actor that participates in the message-passing protocol of risk propagation. Herein, the user of an actor will only be referred to as an *actor*. The following variant of the concurrent, object-oriented actor model is assumed to define actor behavior [4, 5].

- An actor follows the *active object pattern* [27, 54] and the *Isolated Turn Principle* [27]. Specifically, the state change of an actor is carried out by instance- variable assignment, instead of the canonical BECOME primitive that provides a functional construct for pipelining actor behavior replacement [3–5]. The interface of a user actor is fixed in risk propagation, so the more general semantics of BECOME is unnecessary.

[3]This is technically referred to as an *actor system configuration*.

[4]Formally, a *message* is called a *task* and is defined by a *tag*, a unique identifier; a *target*, the mail address to which the message is delivered; and a *communication*, the message content [3].

- The term "name" [3, 38] is preferred over "mail address" [3–5] to refer to the sender of a message. Generally, the mail address that is included in a message need not correspond to the actor that sent it. Risk propagation, however, requires this actor is identified in a risk-score message. Therefore, to emphasize this requirement, "name" is used to refer to both the identity of an actor and its mail address.

- An actor is allowed to include a loop with finite iteration in its behavior definition; this is traditionally prohibited in the actor model [3, 4].

- Because the following pseudocode to describe actor behavior mostly follows the conventions of [22] (see Appendix C), the behavior definition is implied by all procedures that take as input an actor.

The CREATE-ACTOR operation initializes an actor, which is equivalent to the *new expression* [3] or CREATE primitive [4, 5] with the exception that it only specifies the attributes (i.e., state) of an actor. As mentioned earlier, the behavior description of an actor is implied by the procedures that require an actor as input. Every actor A has the following attributes.

- $A.name$: an identifier that enables actors to communicate with it [3, 38].

- $A.contacts$: a dictionary (see Appendix C) that maps an actor name to a timestamp. That is, if user i of actor A_i comes in contact with user j of actor A_j, then the *contacts* of A_i contains the name of A_j along with the most recent contact time for users i and j. The converse holds for user j. This is an extension of the *acquaintance vector* [38] or *acquaintance list* [3, 5].

- $A.score$: the user's current exposure score. This attribute is either a default risk score (see DEFAULT-RISK-SCORE), a risk score sent by an actor of some contacted user[5], or a symptom score of the user.

- $A.cache$: a dictionary that maps a time interval to a risk score; used to tolerate synchronization delays between a user's device and actor (see Section 2.3.1.2).

CREATE-ACTOR

1: $A.name \leftarrow$ CREATE-NAME
2: $A.contacts \leftarrow \emptyset$
3: $A.score \leftarrow$ DEFAULT-RISK-SCORE(A)
4: $A.cache \leftarrow \emptyset$
5: **return** A

DEFAULT-RISK-SCORE(A)

1: $s.value \leftarrow 0$
2: $s.t \leftarrow 0$
3: $s.sender \leftarrow A.name$
4: **return** s

Risk scores and contacts have finite relevance, which is parametrized by a *liveness* or *relevance duration* $\Delta t_s, \Delta t_c > 0$, respectively. The relevance of risk scores and contacts is important, because it influences how actors pass messages. For example, actors do not send irrelevant risk scores or relevant risk scores to irrelevant contacts. The *time-to-live* (TTL) of a risk score or

[5]As discussed later (see HANDLE-CONTACT-MESSAGE in Section 2.3.1.1), it possible for an actor i to send a message to an actor j but not be a contact of actor j.

contact is the remaining duration of its relevance. In the following operations, $s.t$ denotes the time at which the risk score was originally computed, $c.t$ is the contact time, and GET-TIME returns the current time. The interface of a

SCORE-TTL(s)

1: **return** $\max\{0, \Delta t_s - (\text{GET-TIME} - s.t)\}$

CONTACT-TTL(c)

1: **return** $\max\{0, \Delta t_c - (\text{GET-TIME} - c.t)\}$

user actor is defined by two types of messages: contact messages and risk-score messages. A *contact message* c contains the name $c.name$ of the actor whose user was contacted and the contact time $c.t$. A *risk-score message* s is simply a risk score along with the actor's name $s.sender$ that sent it. A risk score previously defined as the ordered pair (r, t) (see Section 2.3) is represented as the attributes r and $s.t$. The following sections discuss how a contact message and risk-message are processed by an actor.

2.3.1.1.1 Handling Contact Messages There are two ways in which a user actor can receive a contact message. The first, technically correct approach is for a receptionist actor to mediate the communication between the user actor and the PDS so that the user actor can retrieve its user's contacts. The second approach is to relax this formality and allow the user actor to communicate with the PDS directly[6].

[6]If the PDS itself is an actor, then a push-oriented dataflow could be implemented, where the user actor receives contact messages (and symptom-score messages). This would be more efficient and timely than a pull-oriented dataflow in which the PDS is a passive data store that requires the user actor or receptionist to poll it for new data.

The HANDLE-CONTACT-MESSAGEoperation defines how a user actor processes a contact message. A contact is assumed to have finite relevance which is parametrized by the *contact time-to-live* $\Delta t_c > 0$. A contact whose contact time occurred no longer than Δt_c time ago is said to be *alive*. Thus, a user actor only adds a contact if it is alive. Regardless of whether the contact

HANDLE-CONTACT-MESSAGE(A, c)

```
1: if CONTACT-TTL(c) > 0
2:     c.key ← c.name
3:     c.threshold ← 0
4:     INSERT(A.contacts, c)
5:     START-CONTACTS-REFRESH-TIMER(A)
6:     SEND-CURRENT-OR-CACHED(A, c)
```

START-CONTACTS-REFRESH-TIMER(A)

```
1: oldest ← MINIMUM(A.contacts)
2: if oldest ≠ NIL
3:     x.key ← "contacts"
4:     START-TIMER(x, CONTACT-TTL(oldest))
```

HANDLE-CONTACTS-REFRESH-TIMER(x)

```
1: for each c ∈ A.contacts
2:     if CONTACT-TTL(c) ≤ 0
3:         DELETE(A.contacts, c)
4: START-CONTACTS-REFRESH-TIMER(A)
```

is alive, the user actor attempts to send a risk-score message that is derived

from its current exposure score or a cached risk score (see Section 2.3.1.2):

Like contacts, each risk score is assumed to have finite relevance

SEND-CURRENT-OR-CACHED(A, c)

1: **if** SHOULD-RECEIVE($c, A.score$)
2: SEND($A, c, A.score$)
3: **else**
4: $s \leftarrow$ CACHE-MAX($A.cache, c.t + \beta$)
5: **if** $s \neq$ NIL **and** SCORE-TTL(s) > 0
6: SEND(A, c, s)

SEND(A, c, s)

1: $s'.value \leftarrow \alpha \cdot s.value$
2: $s'.t \leftarrow s.t$
3: $s'.sender \leftarrow A.name$
4: SEND(c, s')
5: UPDATE-THRESHOLD(c, s')

SHOULD-RECEIVE(c, s)

1: **return** $c.threshold < s.value$ **and** $c.t + \beta \geq s.t$ **and** SCORE-TTL(s) $>$ 0 **and** $c.name \neq s.sender$

UPDATE-THRESHOLD(c, s)

1: $threshold \leftarrow \gamma \cdot s.value$
2: **if** $threshold > c.threshold$
3: $c.threshold \leftarrow threshold$
4: START-THRESHOLD-TIMER(c, s)

START-THRESHOLD-TIMER(c, s)

1: $x.key \leftarrow c.name +$ " threshold"
2: $x.name \leftarrow c.name$
3: START-TIMER$(x,$ SCORE-TTL$(s))$

HANDLE-THRESHOLD-TIMER(A, x)

1: $c \leftarrow$ SEARCH$(A.contacts, x.name)$
2: **if** $c \neq$ NIL
3: $\quad s \leftarrow$ CACHE-MAX$(A.cache, c.t + \beta)$
4: $\quad$ **if** $s \neq$ NIL **and** SCORE-TTL$(s) > 0$
5: $\quad\quad c.threshold \leftarrow \gamma \cdot s.value$
6: $\quad\quad$ START-THRESHOLD-TIMER(c, s)
7: $\quad$ **else**
8: $\quad\quad c.threshold \leftarrow 0$

that is parametrized by the *score time-to-live* $\Delta t_s > 0$ and evaluated by the operation IS-SCORE-ALIVE. To send an actor's current exposure score, the contact must be sufficiently recent. It is assumed that risk scores computed after a contact occurred have no effect on the user's exposure score.

To account for the disease incubation period, a delay in reporting symptoms, or a delay in establishing actor communication, a time buffer $\beta \geq 0$ is considered. That is, a risk score is not sent to a contact if IS-CONTACT-RECENT returns FALSE.

The TRANSMITTED operation is used to generate risk scores that are sent to other actors. It is assumed that contact only implies an incomplete transmission of risk between users. Thus, when sending a risk score to another actor, the value of the risk score is scaled by the *transmission rate* $\alpha \in (0, 1)$.

Notice that the time of the risk score is left unchanged; the act of sending a risk-score message is independent of when the risk score was first computed.

If line 1 of SEND-CURRENT-OR-CACHED evaluates to FALSE, then the actor attempts to retrieve the maximum cached risk-score message based on the buffered contact time. If such a message exists and is alive, a risk-score message is derived and sent to the contact. The SEND operation follows the semantics of the SEND-TO primitive [3, 4] or the *send command* [5].

2.3.1.1.2 Handling Risk-Score Messages Upon receiving a risk-score message, an actor executes the following operation. The UPDATE-ACTOR HANDLE-RISK-SCORE-MESSAGE(A, s)

```
1: CACHE-INSERT(A.cache, s)
2: if s.value > A.score.value
3:     previous ← A.score
4:     A.score ← s
5:     if previous ≠ DEFAULT-RISK-SCORE(A)
6:         START-SCORE-REFRESH-TIMER(A)
7: PROPAGATE(A, s)
```

operation is responsible for updating an actor's state, based on a received risk-score message. Specifically, it stores the message inside the actor's interval cache $A.cache$, assigns the actor a new exposure score and send coefficient (discussed below) if the received risk-score value exceeds that of the current exposure score, and removes expired contacts. In previous work [10], risk propagation assumes synchronous message passing, so the notion of an

START-SCORE-REFRESH-TIMER(A)

1: $x.key \leftarrow$ "score"

2: START-TIMER(x, SCORE-TTL($A.score$))

HANDLE-SCORE-REFRESH-TIMER(A, x)

1: $s \leftarrow$ CACHE-MAX(GET-TIME)

2: **if** $s =$ NIL

3: $\quad s \leftarrow$ DEFAULT-RISK-SCORE(A)

4: $A.score \leftarrow s$

5: **if** $s \neq$ DEFAULT-RISK-SCORE(A)

6: $\quad$ START-SCORE-REFRESH-TIMER(A)

iteration or inter-iteration difference threshold can be used as stopping conditions. However, as a streaming algorithm that relies on asynchronous message passing, such stopping criteria are unnatural. Instead, the following heuristic is applied and empirically optimized to minimize accuracy loss and maximize efficiency. Let $\gamma > 0$ be the *send coefficient* such that an actor only sends a risk-score message if its value exceeds the actor's *send threshold* $A.threshold$ (see line **??** in PROPAGATE).

Assuming a finite number of actors, any positive send coefficient γ guarantees that a risk-score message will be propagated a finite number of times. Because the value of a risk score that is sent to another actor is scaled by the transmission rate α, its value exponentially decreases as it propagates away from the source actor with a rate constant $\log \alpha$.

As with the SEND-CURRENT-OR-CACHED operation, a risk-score message must be alive and relatively recent for it to be propagated. As in previous

work [10], factor marginalization is achieved by not propagating the received message to the actor who sent it. The logic of PROPAGATE differs, however, in two ways. First, it is possible that no message is not propagated to a contact. The intent of sending a risk-score message is to update the exposure score of other actors. However, previous work [10] required that a "null" message with a risk-score value of 0 is sent. Sending such ineffective messages incurs additional communication overhead. The second difference is that only the *most recent* contact time is used to determine if a message should be propagated to a contact. Contact times determine what messages are relevant. Given two contact times t_1, t_2 such that $t_1 \leq t_2$, then any risk score with time $t \leq t_1 + \beta$ also satisfies $t \leq t_2 + \beta$. Thus, storing multiple contact times is unnecessary.

PROPAGATE(A, s)

1: **for each** $c \in A.contacts$
2: **if** SHOULD-RECEIVE(c, s)
3: SEND(A, c, s)

2.3.1.2 Risk Score Caching

For two actors to communicate, each must have the other actor in their contacts (see Section 2.3.1.1). Recall that an actor must retrieve these contacts from the user's PDS, which subsequently requires synchronization with the user's mobile device (see **??**). While the user's device can locally store contacts from proximal devices and symptoms of the user, an internet connection is needed to synchronize with the PDS and thus the user's actor. Therefore, it

is not only possible but a reality that the user's mobile device and actor will not always be synchronized.

In the best case, this "lag" may only be a few seconds; in the worst case, the user's device is offline for several days. If δ_i (δ_j) is the delay between when the device of user i (ref. j) records a contact with user j (ref. i) and when its actor receives the corresponding contact message, then the delay between when actors A_i and A_j can communicate bidirectionally and when the contact actually occurred is $\delta_{ij} = \max(\delta_i, \delta_j)$. Such dissonance between the "true" state of the world (i.e., when users actually came in contact) and that known to the network of actors could impact the accuracy of risk propagation, which assumes such delays are nonexistent. To address this issue, each actor maintains a cache of received risk-score messages such that it can still send a message that reflects its previous state to a contact that was significantly delayed.

An *interval cache* C is a dictionary that maps a finite time interval (key) to a data element (value). In a typical cache, the *time-to-live* (TTL) of an element is a fixed duration after which the element is removed or *evicted*. In an interval cache, however, the TTL of an element is determined by its associated interval and the current time. That is, an interval cache is like a series of sliding windows, where each window corresponds to an interval that can hold a single element. Thus, the TTL of an element is the duration between the start of its interval and the start of the earliest interval in the cache.

An interval cache maintains N contiguous, half-closed (start-inclusive) intervals, each of duration Δt. An interval cache contains N_a *look-ahead intervals*

and N_b *look-back intervals* such that $N = N_b + N_a$. Look- ahead (resp. look-back) intervals allow elements to be associated with intervals whose start times are later (resp. earlier) than the current time t. The *look-back duration* Δt_b and *look-ahead duration* Δt_a are defined as

$$\Delta t_b = N_b \cdot \Delta t$$
$$\Delta t_a = N_a \cdot \Delta t.$$

The *range* of the interval cache is $[C.start, C.end)$, where

$$C.start = C.refresh - \Delta t_b$$
$$C.end = C.refresh + \Delta t_a,$$

and $C.refresh$ is the time at which the cache was last refreshed.

An interval cache is a "live" data structure, so the range must be updated periodically to reflect the advancement of time. Furthermore, intervals and their associated elements that are no longer contained in the range must be evicted. This process of updating the range and evicting cached elements is called *refreshing the cache*. The *refresh period* $T > 0$ of an interval cache is the duration until the range is updated, based on the current time t. Depending on the interval duration Δt and the nature of the data that is being cached, the refresh period may be on the order of seconds or days. To recognize when a refresh is necessary, the cache maintains the attribute $C.refresh$, which is the time of the previous refresh. The operation CACHE-REFRESHupdates the range if at least T time has elapsed since the previous refresh and then removes

all expired elements.

CACHE-REFRESH(C)

```
1: t ← GET-TIME
2: if t − C.refresh > T
3:     C.start ← t − Δt_b
4:     C.end ← t + Δt_a
5:     C.refresh ← t
6:     for each x ∈ C
7:         if x.key < C.start
8:             DELETE(C, x)
```

The operation CACHE-INSERTrefreshes the cache, if necessary, and merges into the cache the element pointed to by x if its timestamp $x.t$ is in the range. Keys in the interval cache are interval start times and are lazily computed (line 2) to avoid storing intervals with no associated element. By not storing all intervals explicitly, the interval cache only achieves $O(N)$ space complexity when each interval has an associated element. The MERGEoperation (line 7) can be as trivial as replacing the existing value. For risk propagation, the interval cache associates with each interval the newest risk score of maximum value.

CACHE-KEY(C, x)

```
1: return C.start + ⌊(x.t − C.start)/Δt⌋ · Δt
```

The intention of sending a cached risk score to a contacted user actor is to account for the delay between when the contact occurred and when the actors establish communication. Therefore, the cached risk score that should be sent

CACHE-INSERT(C, x)

1: **if** $x.t \in [C.start, C.end)$
2: $\quad x.key \leftarrow$ CACHE-KEY(C, x)
3: $\quad x_{old} \leftarrow$ SEARCH(C, x)
4: $\quad$ **if** $x_{old} =$ NIL
5: $\quad\quad x_{new} \leftarrow x$
6: $\quad$ **else**
7: $\quad\quad x_{new} \leftarrow$ MERGE(x_{old}, x)
8: $\quad$ INSERT(C, x_{new})

is that which would have been the current exposure score at the time the users came into contact. That is, each user actor should send the maximum risk score whose cache interval ends at or before the time of contact, accounting for the time buffer β (see line 4 of SEND-CURRENT-OR-CACHEDin Section 2.3.1.1). The operation CACHE-MAXis used to carry out this query.

CACHE-MAX(C, t)

1: **return** MAXIMUM$(\{x \in C \mid x.key <$ CACHE-KEY$(t)\})$

An interval cache is implemented by augmenting a hash table [**?** , pp. 253–285] with the aforementioned attributes and parameters. By using a hash table, the interval cache offers $\Theta(1)$ average-case insert, search, and delete operations. Reference [**?** , pp. 348–354] implements an *interval tree* by augmenting a red-black tree. However, insert, delete, and search operations on a red-black tree require $\Theta(\log N)$ in the average case.

2.4 Message Reachability

A fundamental concept of reachability in temporal networks is the *time-respecting path*: a contiguous sequence of contacts with nondecreasing time. Thus, vertex v is *temporally reachable* from vertex u if there exists a time-respecting path from u to v, denoted $u \to v$ [67]. The following quantities are derived from the notion of a time-respecting path and help quantify reachability in a time-varying network [40].

- The *influence set* $I(v)$ *of vertex* v is the set of vertices that v can reach by a time-respecting path.
- The *source set* $S(v)$ *of vertex* v is the set of vertices that can reach v by a time-respecting path.
- The *reachability ratio* $f(G)$ *of a temporal network* G is the average influence-set cardinality of a vertex v.

Generally, a message-passing algorithm defines a set of constraints that determine when and what messages are sent from one vertex to another. Even if operating on a temporal network, those constraints may be more or less strict than requiring temporal reachability. As a dynamic process, message passing on a time-varying network requires a more general definition of reachability that can account for network topology *and* message-passing semantics [11].

Formally, the *message reachability from vertex* u *to vertex* v is the number

of edges along the *shortest path* P that satisfy the message passing constraints,

$$m(u,v) = \sum_{(i,j)\in P} f(u,i,j,v),$$

where

$$f(u,i,j,v) = \begin{cases} 1 & \text{if all constraints are satisfied} \\ 0 & \text{otherwise.} \end{cases}$$

Vertex v is *message reachable* from vertex u if there exists a shortest path such that $m(u,v) > 0$. The *message reachability of vertex* u is the maximum message reachability from vertex u:

$$m(u) = \max\{m(u,v) \mid v \in V\}. \tag{2.2}$$

The temporal reachability metrics previously defined can be extended to message reachability by only considering the message-reachable vertices:

$$I_m(u) = \{v \in V \mid m(u,v) > 0\}$$
$$S_m(v) = \{u \in V \mid m(u,v) > 0\}$$
$$f_m(G) = \frac{\sum_{v\in V} |I_m(v)|}{|V|}.$$

Let $\mathbf{M}$ be the $|V| \times |V|$ *message-reachability matrix* of the temporal network

G such that vertices are enumerated $1, 2, \ldots, |V|$ and for each $m_{ij} \in \mathbf{M}$,

$$m_{ij} = \begin{cases} 1 & \text{if } m(i,j) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Then the cardinality of the influence set for vertex i is the number of nonzero elements in the ith row of $\mathbf{M}$:

$$|I_m(i)| = \sum_{j=1}^{|V|} m_{ij}. \tag{2.3}$$

Similarly, the cardinality of the source set for vertex j is the number of nonzero elements in the jth column of $\mathbf{M}$:

$$|S_m(j)| = \sum_{i=1}^{|V|} m_{ij}. \tag{2.4}$$

For risk propagation, let $H(x)$ be the *Heaviside step function*,

$$H(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Then message reachability is defined as

$$m(u,v) = \sum_{(i,j) \in P} f_r(u,i) \cdot f_c(u,i,j) \tag{2.5}$$

where P is the set of edges along the shortest path $u \to v$ such that the actors

are enumerated $0, 1, \ldots, |P| - 1$ and

$$f_r(u, i) = H(\alpha^i \cdot r_u - \gamma \cdot r_i) \tag{2.6}$$

$$f_c(u, i, j) = H(t_{ij} - t_u + \beta) \tag{2.7}$$

are the value and contact-time constraints in the SHOULD-RECEIVE operation (see Section 2.3.1.1), where (r_i, t_i) is the current exposure score for actor i and t_{ij} is the most recent contact time between actors i and j.

The value of (2.5) can be found by associating with each symptom score a unique identifier. If each actor maintains a log of the risk scores it receives, then the set of actors that receive the symptom score or a propagated risk score thereof can be identified. This set of actors defines the induced subgraph on which to compute (2.5) using a shortest-path algorithm [42].

Regarding efficiency, (2.2) to (2.4) provide the means to quantify the communication overhead of a given message-passing algorithm on a temporal network. Moreover, because such metrics capture the temporality of message passing, they can better quantify complexity than traditional graph metrics.

By relaxing the constraint (2.7), it is possible to estimate (2.5) with (2.6). The *estimated message reachability of vertex u to vertex v*, denoted $\hat{m}(u, v)$, is defined as follows. Based on (2.6),

$$\alpha^{\hat{m}(u,v)} \cdot r_u \leq \gamma \cdot r_v,$$

where the left-hand side is the value of the propagated symptom score for actor u when $\hat{m}(u, v) = 1$, and the right-hand side is the value required by

some message-reachable actor v to propagate the message received by actor u or some intermediate actor. Solving for $\hat{m}(u, v)$,

$$\hat{m}(u, v) \leq f(u, v), \tag{2.8}$$

where

$$f(u, v) = \begin{cases} 0 & \text{if } r_u = 0 \\ |P| & \text{if } r_v = 0 \\ \log_\alpha \gamma + \log_\alpha r_v - \log_\alpha r_u & \text{otherwise.} \end{cases}$$

Equation (2.8) indicates that a lower send coefficient γ will generally result in higher message reachability, at the cost of sending possibly ineffective messages (i.e., risk scores that do not update the exposure score of another actor). Equation (2.8) also quantifies the effect of the transmission rate α. Unlike the send coefficient, however, the transmission rate is intended to be derived from epidemiology to quantify disease infectivity and should not be optimized to improve performance.

Given the multivariate nature of message reachability, it is helpful to visualize how it with various combinations of parameter values. Figure 2.4 includes several line plots of estimated message reachability $\hat{m}(u, v)$ with respect to the initial risk score magnitude of actor u.

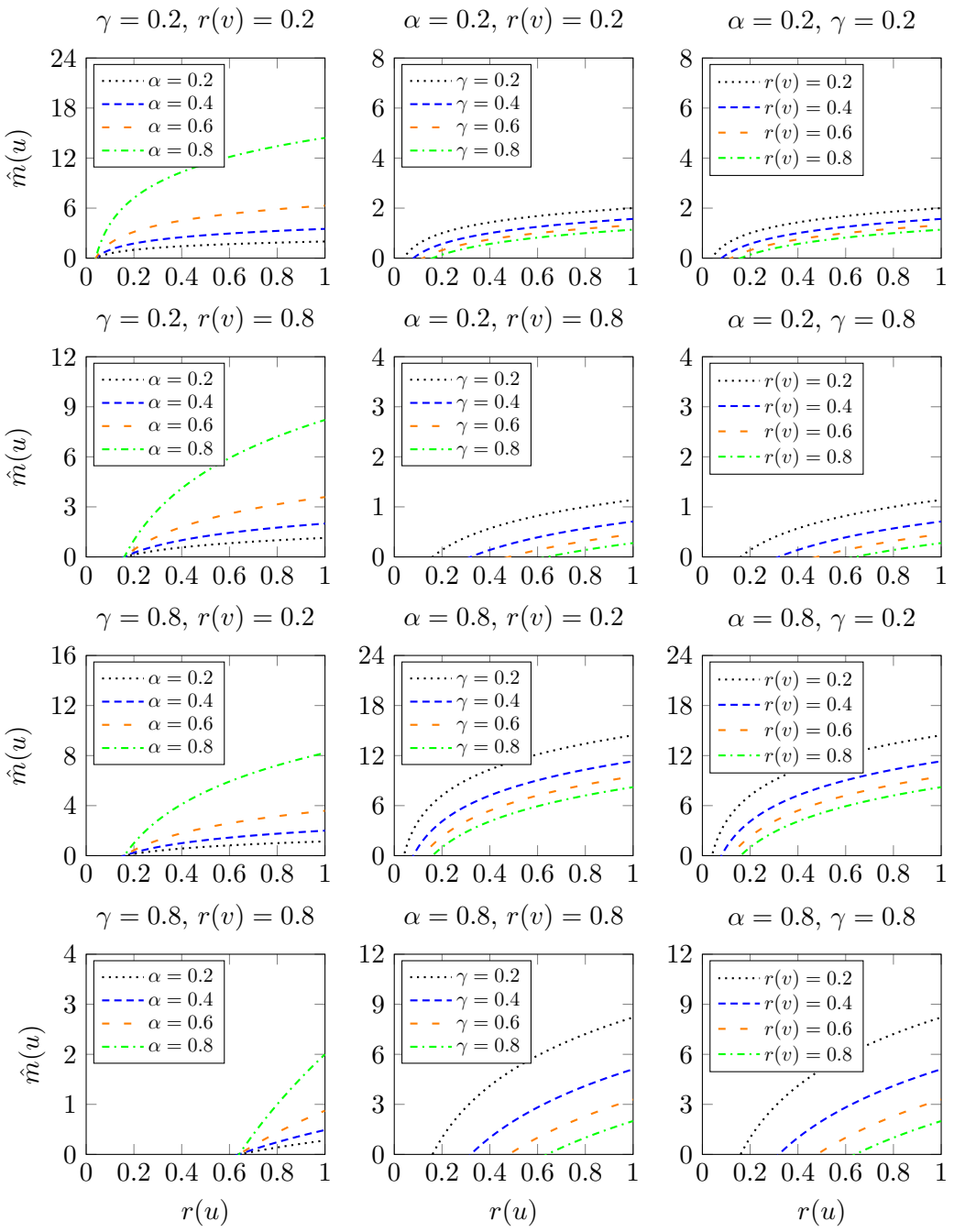

Figure 2.4: Estimated risk propagation message reachability $\hat{m}(u)$ for different values of the transmission rate α, the send tolerance γ, and the initial magnitude $r(v) = r_0(v)/\alpha$ of the destination user v with respect to the initial magnitude $r_u = r_0(u)/\alpha$ of the source user u.

2.5 Complexity

Risk propagation has worst-case time complexity $O(|C|)$ because the number of messages scales with the number of contacts. As noted in Section 2.4, however, this does not capture the temporal constraints of message passing on a time-varying contact network. This is a tighter bound than previous work [10] which states that the worst-case time complexity is $O(n^2)$, where n is the number of actors. The space complexity of risk propagation is also $O(|C|)$ because each actor must store their contacts and the messages received in their mailbox.

Chapter 3

Evaluation and Discussion

3.1 Evaluation

3.2 Discussion

3.2.1 Mobile Crowdsensing

Mobile crowdsensing (MCS) is a "sensing paradigm that empowers ordinary citizens to contribute data sensed or generated from their mobile devices" that is aggregated "in the cloud for crowd intelligence extraction and human-centric service delivery" [36]. Over the past decade, substantial research has been conducted on defining and classifying MCS applications [19, 36, and references therein]. While not discussed in previous work [9, 10], ShareTrace is a MCS application. The following characterization of ShareTrace assumes the four-layered architecture of a MCS application [19], which offers a comprehensive set of classification criteria that To offer a clear comparison, Table 3.1 follows

the same structure as [19]. Some aspects of the architecture, namely sampling frequency and sensor activity, are marked according to how ShareTrace is described in previous work, rather than how it would function to optimize for energy efficiency. More detail is provided below in this regard. When classifying ShareTrace as an MCS application, the following description is helpful:

> *ShareTrace is a decentralized, delay-tolerant contact-tracing application that estimates infection risk from proximal user interactions and user symptoms.*

3.2.1.1 Application Layer

3.2.1.1.1 Application Tasks

Task Scheduling *Proactive scheduling* allows users to decide when and where they contribute sensing data, while *reactive scheduling* requires that "a user receives a request, and upon acceptance, accomplishes a task" [19]. ShareTrace follows proactive scheduling where the sensing task is to detect proximal interactions with other users. Naturally, the scheduling of this task is at the discretion of the ShareTrace user.

Task Assignment *Centralized assignment* assumes that "a central authority distributes tasks among users." Conversely, with *decentralized assignment*, "each participant becomes an authority and can either perform the task or forward it to other users. This approach is very useful when users are very interested in specific events or activities" [19]. The latter naturally aligns with

ShareTrace in which each user is responsible for their own interactions that are both temporally and spatially specific.

Task Execution With *single-task execution*, MCS applications assign one type of task to users, while *multi-task execution* assigns multiple types of tasks [19]. ShareTrace only involves the single task of sensing proximal interactions. Alternatively, ShareTrace could be defined more abstractly as sensing infection risk through interactions and user symtpoms. In this case, ShareTrace would follow a multi-tasked execution model, where the task of sensing user symptoms is achieved through user reporting or bodily sensors.

3.2.1.1.2 Application Users

User Recruitment *Volunteer-based recruitment* is when citizens can "join a sensing campaign for personal interests... or willingness to help the community," while *incentive-based recruitment* promotes participation and offers control over the recruitment rate... These strategies are not mutually exclusive and users can first volunteer and then be rewarded for quality execution of sensing tasks" [19]. ShareTrace assumes volunteer-based recruitment. However, as a decentralized application (dApp) that aligns with the principles of self-sovereignty, an incentive structure that rewards users with verifiable, high-quality data is plausible.

User Selection *User-centric selection* is when "contributions depend only on participants['] willingness to sense and report data to the central col-

lector, which is not responsible for choosing them." *Platform-centric selection* is "when the central authority directly decides data contributors... Platform centric decisions are taken according to the utility of data to accomplish the targets of the campaign" [19]. ShareTrace employs user-centric selection, because the purpose of the application is to passively sense the user's interactions and provide them with the knowledge of their infection risk.

User Type A *contributor* "reports data to the MCS platform with no interest in the results of the sensing campaign" and "are typically driven by incentives or by the desire to help the scientific or civil communities." A *consumer* joins "a service to obtain information about certain application scenario[s] and have a direct interest in the results of the sensing campaign" [19]. For ShareTrace, users can be either a consumer or a contributor but are likely biased toward the former.

3.2.1.2 Data Layer

3.2.1.2.1 Data Management

Data Storage *Centralized storage* involves data being "stored and maintained in a single location, which is usually a database made available in the cloud. This approach is typically employed when significant processing or data aggregation is required." *Distributed storage* "is typically employed for delay-tolerant applications, i.e., when users are allowed to deliver data with a delayed reporting" [19]. For sensing human interaction, ShareTrace relies on distributed storage in the form of Dataswift Personal Data Accounts. More-

over, ShareTrace is delay-tolerant, so distributed storage is most appropriate. However, for reporting the population-level risk distribution, it is likely that centralized storage would be used.

Data Format *Structured data* is standardized and readily analyzable. *Unstructured data*, however, requires significant processing before it can be used [19]. ShareTrace deals with structured data (e.g., user symptoms, actor URIs).

Data Dimensionality *Single-dimension data* typically occurs when a single sensor is used, while *multi-dimensional data* arises with the use of multiple sensors. ShareTrace data is one-dimensional because it only uses Bluetooth to sense nearby users.

3.2.1.2.2 Data Processing

Data Pre-processing *Raw data output* implies that no modification is made to the sensed data. *Filtering and denoising* entail "removing irrelevant and redundant data. In addition, they help to aggregate and make sense of data while reducing at the same time the volume to be stored" [19]. ShareTrace only retains the actor URIs that correspond to valid contacts (i.e., lasting at least 15 minutes).

Data Analytics *Machine learning* (ML) *and data mining analytics* are not real-time. They "aim to infer information, identify patterns, or predict future trends." On the contrary, *real-time analytics* consist of "examining

collected data as soon as it is produced by the contributors" [19]. ShareTrace aligns with the former category since it aims to infer the infection risk of users.

Data Post-processing *Statistical post-processing* "aims at inferring proportions given quantitative examples of the input data. *Prediction post-processing* aims to determine "future outcomes from a set of possibilities when given new input in the system" [19]. ShareTrace applies predictive post-processing via risk propagation.

3.2.1.3 Communication Layer

3.2.1.3.1 Communication Technology

Infrastructured Technology *Cellular* connectivity "is typically required from sensing campaign[s] that perform real-time monitoring and require data reception as soon as it is sensed." *WLAN* "is used mainly when sensing organizers do not specify any preferred reporting technologies or when the application domain permits to send data" at "a certain amount of time after the sensing process" [19]. Infrastructured technology is also referred to as the *infrastructured transmission paradigm* [60]. ShareTrace does not require cellular infrastructure, because it is delay-tolerant and thus only requires WLAN.

Infrastructure-less Technology *Infrastructure-less technologies* "consists of device-to-device (D2D) communications that do not require any infrastructure... but rather allow devices in the vicinity to communicate directly." Technologies include *WiFi-Direct*, *LTE-Direct*, and *Bluetooth* [19].

Infrastructure-less technology is also called the *opportunistic transmission paradigm* [60]. ShareTrace uses Bluetooth because of its energy efficiency and short range.

3.2.1.3.2 Data Reporting

Upload mode With *relay uploading*, "data is delivered as soon as collected." *Store-and-forward* "is typically used in delay-tolerant applications when campaigns do not need to receive data in real-time" [19]. Because ShareTrace is delay-tolerant, it uses store-and-forward uploading.

Metholodgy *Individualized sensing* is "when each user accomplishes the requested task individually and without interaction with other participants." *Collaborative sensing* is when "users communicate with each other, exchange data[,] and help themselves in accomplishing a task or delivering information to the central collector. Users are typically grouped and exchange data exploiting short-range communication technologies, such as WiFi-[D]irect or Bluetooth... Note that systems that create maps merging data from different users are considered individual because users do not interact between each other to contribute" [19]. The methodology of sensing is similar to the *sensing scale* which is typically dichotomized as *personal* [34, 53] (i.e., individualized) and *community* [34] or *group* [53] (i.e., collaborative). ShareTrace is inherently collaborative, relying on mobile devices to exchange actor URIs to estimate infection risk. Thus, collaborative sensing is used.

Timing *Timing* is based on whether devices "should sense in the same period or not." *Synchronous timing* "includes cases in which users start and accomplish at the same time the sensing task. For synchronization purposes, participants communicate with each other." *Asynchronous timing* occurs "when users perform sensing activity not in time synchronization with other users" [19]. ShareTrace requires synchronous timing, because contact sensing inherently requires synchronous communication between the involved devices.

3.2.1.4 Sensing Layer

3.2.1.4.1 Sensing Elements

Sensor Deployment *Dedicated deployment* involves the use of "non-embedded sensing elements," typically for a specific task. *Non-dedicated deployment* utilizes sensors that "do not require to be paired with other devices for data delivery but exploit the communication capabilities of mobile devices" [19]. ShareTrace relies on non-dedicated deployment since it relies on Bluetooth that is ubiquitous in modern-day mobile devices.

Sensor Activity *Always-on sensors* "are required to accomplish mobile devices['] basic functionalities, such as detection of rotation and acceleration... Activity recognition [i.e., context awareness]... is a very important feature that accelerometers enable." *On-demand sensors* "need to be switched on by users or exploiting an application running in the background. Typically, they serve more complex applications than always-on sensors and consume a higher amount fo energy" [19]. ShareTrace uses Bluetooth, which may be considered

on-demand. While energy efficient, users do control when it is enabled. Ideally, ShareTrace would also use always-on sensors to enable Bluetooth with context awareness (i.e., that the user is carrying or nearby the device).

Acquisition *Homogeneous acquisition* "involves only one type of data and it does not change from one user to another one," while *heterogeneous acquisition* "involves different data types usually sampled from several sensors" [19]. ShareTrace is homogeneous, because all users sense the same data from one type of sensor.

3.2.1.4.2 Data Sampling

Sampling Frequency *Continuous sensing* "indicates tasks that are accomplished regularly and independently [of] the context of the smartphone or the user['s] activities." *Event-based sensing* is "data collection [that] starts after a certain event has occurred. In this context, an event can be seen as an active action from a user or the central collector, but also a given context awareness" [19]. ShareTrace sensing is continuous but would ideally be event-based to conserve device energy.

Sensing Responsibility When the *mobile device* is responsible, "devices or users take sampling decisions locally and independently from the central authority... When devices take sampling decisions, it is often necessary to detect the context [of the] smartphones and wearable devices... The objective is to maximize the utility of data collection and minimize the cost of perform-

ing unnecessary operations." When the *central collector* is responsible, they make "decisions about sensing and communicate them to the mobile devices" [19]. Given the human-centric nature of the ShareTrace sensing task, mobile devices are responsible.

User involvement *Participatory involvement* "requires active actions from users, who are explicitly asked to perform specific tasks. They are responsible to consciously meet the application requests by deciding when, what, where, and how to perform sensing tasks." *Opportunistic involvement* means that "users do not have direct involvement, but only declare their interest in joining a campaign and providing their sensors as a service. Upon a simple handshake mechanism between the user and the MCS platform, a MCS thread is generated on the mobile device (e.g., in the form of a mobile app), and the decisions of what, where, when, and how to perform the sensing are delegated to the corresponding thread. After having accepted the sensing process, the user is totally unconscious with no tasks to perform and data collection is fully automated... The smartphone itself is context-aware and makes decisions to sense and store data, automatically determining when its context matches the requirements of an application. Therefore, coupling opportunistic MCS systems with context-awareness is a crucial requirement" [19]. Earlier works on MCS refer to user involvement as the *sensing paradigm* [34, 53, 60]. ShareTrace is opportunistic, ideally with context-awareness.

Application	Task	Scheduling	Proactive Reactive	•
		Assignment	Centralized Decentralized	 •
		Execution	Single task Multi-tasking	•
	User	Recruitment	Voluntary Incentivized	•
		Selection	Platform-centric User-centric	 •
		Type	Consumer Contributor	• •
Data	Management	Storage	Centralized Distributed	 •
		Format	Structured Unstructured	•
		Dimension	Single dimension Multi-dimensional	•
	Processing	Pre-processing	Raw data Filtering and denoising	 •
		Analytics	ML and data mining Real-time	•
		Post-processing	Statistical Prediction	 •
Communication	Technologies	Infrastructured	Cellular WLAN	• •
		Infrastructure-less	LTE-Direct WiFi-Direct Bluetooth	 •
	Reporting	Upload mode	Relay Store and forward	•
		Methodology	Individual Collaborative	 •
		Timing	Synchronous Asynchronous	•
Sensing	Elements	Deployment	Dedicated Non-dedicated	 •
		Activity	Always-on On-demand	• ○
		Acquisition	Homogeneous Heterogeneous	•
	Sampling	Frequency	Continuous Event-based	• ○
		Responsibility	Mobile device Central collector	•
		User involvement	Participatory Opportunistic	 •

Table 3.1: ShareTrace classification using the four-layered architecture of a mobile crowdsensing application [19]. Always (•); with context-awareness (○).

3.2.2 Self-Sovereignty

Chapter 4

Evaluation

4.1 Experimental Design

Risk propagation requires a partitioning or clustering algorithm, as described in Algorithm **??**. We configured the METIS graph partitioning algorithm [45] to use k-way partitioning with a load imbalance factor of 0.2, to attempt contiguous partitions that have minimal inter-partition connectivity, to apply 10 iterations of refinement during each stage of the uncoarsening process, and to use the best of 3 cuts.

4.1.1 Synthetic Graphs

We evaluate the scalability and efficiency of risk propagation on three types of graphs: a random geometric graph (RGG) [24], a benchmark graph (LFRG) [52], and a clustered scale-free graph (CSFG) [39]. Together, these graphs demonstrate some aspects of community structure [30] which allows us to more

accurately measure the performance of risk propagation. When constructing a RGG, we set the radius to $r(n) = \min\left(1, 0.25^{\log_{10}(n)-1}\right)$, where n is the number of users. This allows us to scale the size of the graph while maintaining reasonable density. We use the following parameter values to create LFRGs: mixing parameter $\mu = 0.1$, degree power-law exponent $\gamma = 3$, community size power-law exponent $\beta = 2$, degree bounds $(k_{\min}, k_{\max}) = (3, 50)$, and community size bounds $(s_{\min}, s_{\max}) = (10, 100)$. Our choices align with the suggestions by [52] in that $\gamma \in \mathbb{R}_{[2,3]}$, $\beta \in \mathbb{R}_{[1,2]}$, $k_{\min} < s_{\min}$, and $k_{\max} < s_{\max}$. To build CSFGs, we add $m = 2$ edges for each new user and use a triad formulation probability of $P_t = 0.95$. For all graphs, we remove self-loops and isolated users.

The following defines our data generation process. Let p be the probability of a user being "high risk" (i.e., $r \geq 0.5$) Then, with probability $p = 0.2$, we sample $L + 1$ values from the uniform distribution $\mathbb{U}_{[0.5,1)}$. Otherwise, we sample from $\mathbb{U}_{[0,0.5)}$. This assumes symptom scores and exposure scores are computed daily and includes the present day. We generate the times of these risk scores by sampling a time offset $t_{\text{off}} \sim \mathbb{U}_{[0\text{s};86{,}400\text{s}]}$ for each user such that $t_d = t_{\text{now}} + t_{\text{off}} - d$ days, where $d \in \mathbb{N}_{[0,L]}$. To generate a contact times, we follow the same procedure for risk scores, except that we randomly sample one of the $L + 1$ times and use that as the contact time.

We evaluate various transmission rates and send tolerances:

$$(\gamma, \alpha) \in \{0.1, 0.2, \ldots, 1\} \times \{0.1, 0.2, \ldots, 0.9\}.$$

For all γ, α, we set $n = 5,000$ and $K = 2$.

To measure the scalability of risk propagation, we consider $n \in \mathbb{N}_{[10^2,10^4]}$ users in increments of 100 and collect 10 iterations for each n. The number of actors we use depends on n such that $K(n) = 1$ if $n < 10^3$ and $K(n) = 2$ otherwise. Increasing K for our choice of n did not offer improved performance due to the communication overhead.

4.1.2 Real-World Graphs

We analyze the efficiency of risk propagation on three real-world contact networks that were collected through the SocioPatterns collaboration. Specifically, we use contact data in the setting of a high school (Thiers13) [31], a workplace (InVS15), and a scientific conference (SFHH) [35]. Because of limited availability of large-scale contact networks, we do not use real-world contact networks to measure the scalability of risk propagation.

To ensure that all risk scores are initially propagated, we shift all contact times forward by t_{now} and use $(t_{\text{now}} - 1 \text{ day})$ when generating risk scores times. In this way, we ensure the most recent risk score is still older than the first contact time. Risk score values are generated in the same manner as described in Section 4.1.1 with the exception that we only generate one score. Lastly, we perform 10 iterations over each data set to obtain an average performance.

4.2 Results

4.2.1 Efficiency

Prior to measuring scalability and real-world performance, we observed the effects of send tolerance and transmission rate on the efficiency of risk propagation. As ground truth, we used the maximum update count for a given transmission rate. Fig. 4.1 indicates that a send tolerance of $\gamma = 0.6$ permits 99% of the possible updates. Beyond $\gamma = 0.6$, however, the transmission rate has considerable impact, regardless of the graph. As noted in Section 2.4, send tolerance quantifies the trade-off between completeness and efficiency. Thus, $\gamma = 0.6$ optimizes for both criteria.

Unlike the update count, Fig. 4.1 shows a more variable relationship with respect to runtime and message count. While, in general, transmission rate (send tolerance) has a direct (resp. inverse) relationship with runtime and message count, the graph topology seems to have an impact on this fact. Namely, the LFRG displayed less variability across send tolerance and transmission rate than the RGG and CSFG, which is the cause for the large interquartile ranges. Therefore, it is useful to consider the lower quartile Q_1, the median Q_2, and the upper quartile Q_3. For $\alpha = 0.8$ and $\gamma = 0.6$, risk propagation is more efficient with $(Q_1, Q_2, Q_3) = (0.13, 0.13, 0.46)$ normalized runtime and $(Q_1, Q_2, Q_3) = (0.13, 0.15, 0.44)$ normalized message count.

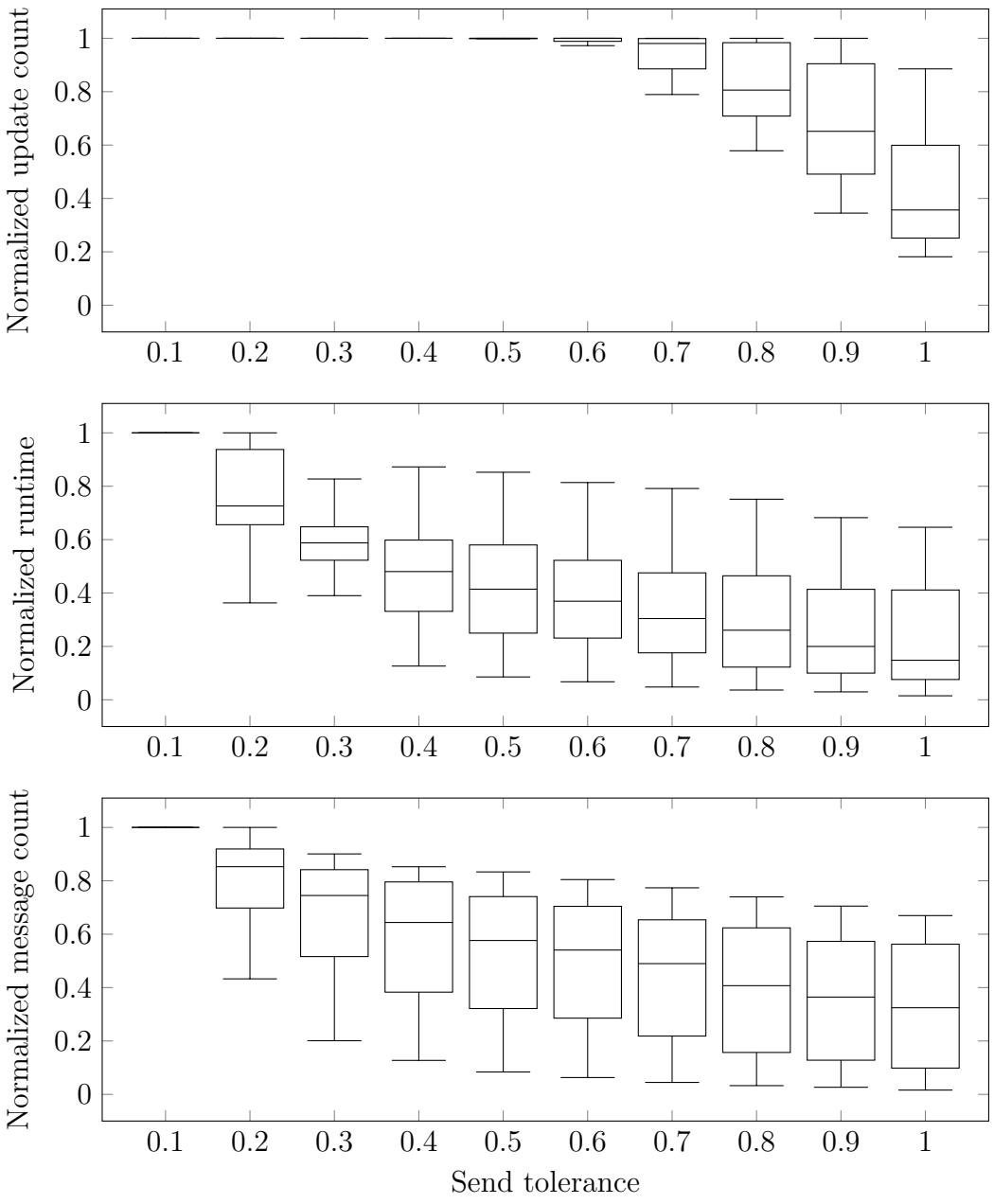

Figure 4.1: Effects of send tolerance on efficiency. All dependent variables are normalized across graphs and transmission rates.

4.2.2 Message Reachability

To validate the accuracy of (2.8), we collected values of (2.5) and (2.8) for real-world and synthetic graphs. For the latter set of graphs, we observed reachability while sweeping across values of γ and α.

To measure the accuracy of (2.8), let the *message reachability ratio* (MRR) be defined as

$$\text{mrr}(u) := \frac{m(u)}{\hat{m}(u)}. \tag{4.1}$$

Overall, (2.8) is a good estimator of (2.5). Across all synthetic graphs, (2.8) modestly underestimated (2.5) with quartiles $(Q_1, Q_2, Q_3) = (0.71, 0.84, 0.98)$ for the (4.1). For $\alpha = 0.8$ and $\gamma = 0.6$, the quartiles of (4.1) were $(Q_1, Q_2, Q_3) = (0.52, 0.77, 1.12)$ and $(Q_1, Q_2, Q_3) = (0.79, 0.84, 0.93)$, respectively. Table 4.1 provides mean values of (4.1) for both synthetic and real-world graphs. Fig. 4.2 indicates that moderate values of γ tend to result in a more stable MRR, with lower (higher) γ underestimating (resp. overestimating) (2.5). With regard to transmission rate, (4.1) tends to decrease with increasing α, but also exhibits larger interquartile ranges.

Because (2.8) does not account for the temporality constraints (2.7) and (**??**), it does not perfectly estimate (2.5). With lower γ and higher α, (2.8) suggests higher MR. However, because a message is only passed under certain conditions (see Algorithm **??**), this causes (2.8) to overestimate (2.5). While (2.8) theoretically is an upper bound on (2.5), it is possible for (2.8) to underestimate (2.5) if the specified value of $r_0(v)$ overestimates the true value of $r_0(v)$. When computing (4.1) for Fig. 4.2, we used the mean $r_0(v)$ across all

users v, so $\text{mrr}(u) > 1$ in some cases.

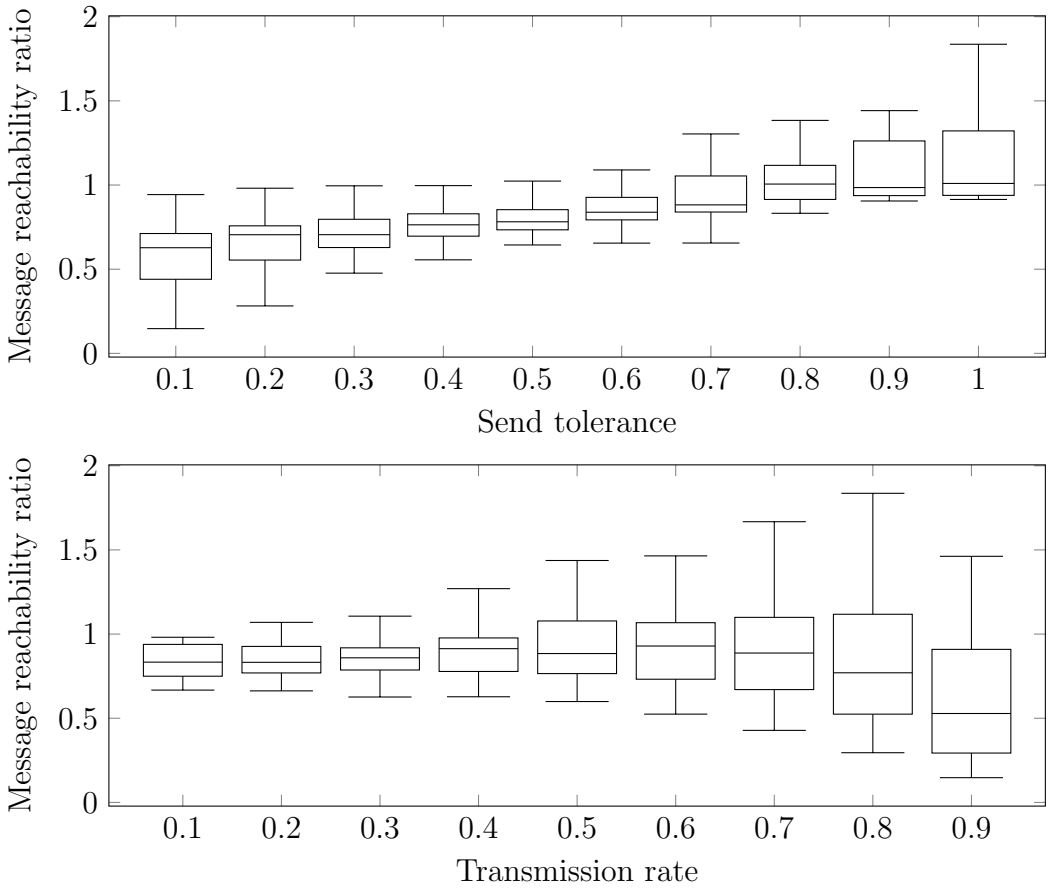

Figure 4.2: Effects of send tolerance and transmission rate on the message reachability ratio. Independent variables are grouped across graphs.

4.2.3 Scalability

Fig. 4.3 describes the runtime behavior of risk propagation. The runtime of CSFGs requires further investigation. A linear regression fit explains ($R^2 = 0.52$) the runtime of LFRGs and RGGs with a slope $m = (1.1 \pm 0.1) \cdot 10^{-3}$ s/contact and intercept $b = 4.3 \pm 1.6$s ($\pm 1.96 \cdot \text{SE}$).

Setting	$\mathrm{mrr}(u) \pm 1.96 \cdot \mathrm{SE}$
Synthetic	
LFR	0.88 ± 0.14
RGG	0.74 ± 0.12
CSFG	0.90 ± 0.14
	$\mathbf{0.85 \pm 0.08}$
Real-world	
Thiers13	0.58 ± 0.01
InVS15	0.63 ± 0.01
SFHH	0.60 ± 0.01
	$\mathbf{0.60 \pm 0.01}$

Table 4.1: Message reachability ratio for synthetic and real-world graphs ($\alpha = 0.8$, $\gamma = 0.6$). Synthetic (real-world) ratios are averaged across parameter combinations (resp. runs).

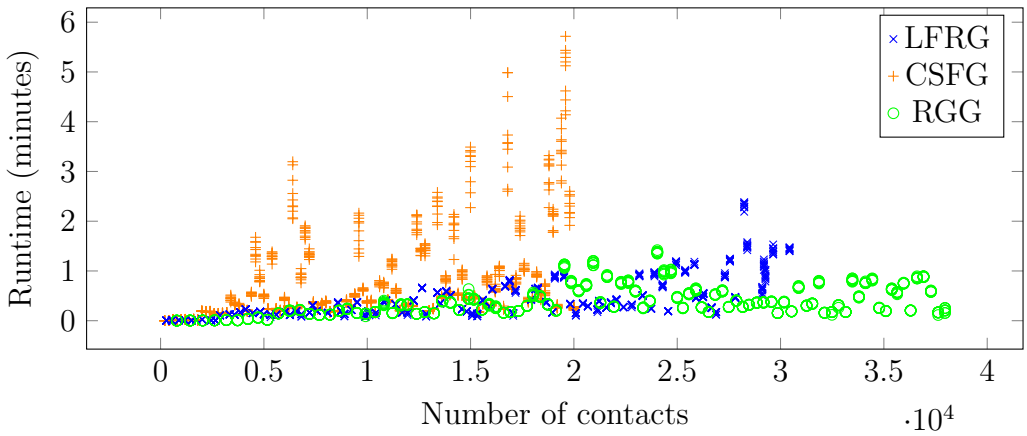

Figure 4.3: Runtime of risk propagation on synthetic graphs containing 100–10,000 users and approximately 200–38,000 contacts.

Chapter 5

Conclusions

Applications like ShareTrace are fundamentally collaborative in that users exchange data amongst each other to achieve an objective or gain personal utility. Maintaining personal data ownership and privacy in this collaborative setting while ensuring architectural scalability and security is an ongoing challenge in the fields of machine learning and cloud computing [18, 41, 43, 76]. Our formulation of risk propagation offers scalability and efficiency and is thus a viable candidate for real-world usage to estimate the spread of infectious diseases. Moreover, message reachability provides researchers and system designers the ability to quantify both the risk of an individual and the effects parameter values have on the efficiency and accuracy of risk propagation.

In future work, we intend to consider mechanisms of establishing decentralized, verifiable communication channels [2] as a means to satisfy the collaborative requirements of user-centric applications, such as ShareTrace. Moreover, we shall consider how privacy-preserving mechanisms, such as differential pri-

vacy [28], may be utilized in such a setting to minimize the personal risks of widespread data sharing.

5.1 Future Work

5.1.1 Add PDA functionality

While our current implementation only accounts for user-reported symptoms, but allows for rich integration of other user data streams, such as wearable and mobile health tracking applications, machine-generated biomarkers (e.g., temperature, coughing, heart rate, oxygen saturation level), and electronic health records. This additional information would allow us to provide advanced and personalized recommendations to the user.

It is important to note that because ShareTrace uses location-based contact tracing, it is not currently interoperable with proximity-based approaches. However, users of ShareTrace gain additional personalized risk assessments based on their symptoms and existing conditions. As part of future work, ShareTrace will offer in-app opt-in options for users to share their risks with government agencies, healthcare providers, their employer, and research organizations. Using the HAT Microserver, users can legally and functionally control how their data is shared with these organizations.

5.1.2 Extend to distributed architecture

PDAs currently function as passive data stores. That is, they are not able to communicate with other PDAs. However, given that PDAs can communicate

actively, we can formulate risk propagation as a distributed algorithm in which we partition the factor graph amongst all the PDAs. Each PDA would contain a variable node and neighboring factor nodes. To minimize the amount of user information that is transferred between PDAs, we can utilize differential privacy [**?**]. In a distributed setting, the message-passing aspect of risk propagation would follow the tenets of reactive streams [1, 14]. With such an architecture, we allow for true scalability and further privacy preservation.

5.1.3 Validate on diverse cohorts

Given that a vaccine is now available for the COVID-19 virus, it is unlikely that ShareTrace, at least as a contact tracing application, will be of immediate relevance. However, to fully understand the effectiveness of our approach, we would need to conduct validation studies on diverse cohorts, such as university students, health care workers, and essential workers. This is particularly important because of the intricacies of human behavioral dynamics, which can be difficult to simulate.

Appendix A

Previous Designs and Implementations

Prior to my work on ShareTrace, I had no experience developing an algorithm that (at least hypothetically) required scalability. Ultimately, I implemented five designs of risk propagation, each providing insight that allowed for iterative improvement in performance. The following provides motivation and context for each design, along with rationale for pursuing an alternative approach.

A.1 Thinking Like a Vertex

The first iteration of risk propagation utilized Apache Giraph [78], an open-source version of the iterative graph-processing library, Pregel [62], which is based on the bulk synchronous parallel model for distributed computing [80]. Giraph follows the think-like-a-vertex paradiagm for graph-processing algorithms in which the algorithm is specified from the perspective of a vertex.

As a highly scalable framework for graph-based algorithms, Giraph was a natural choice for implementing risk propagation. Figure A.1 describes the full compute architecture.

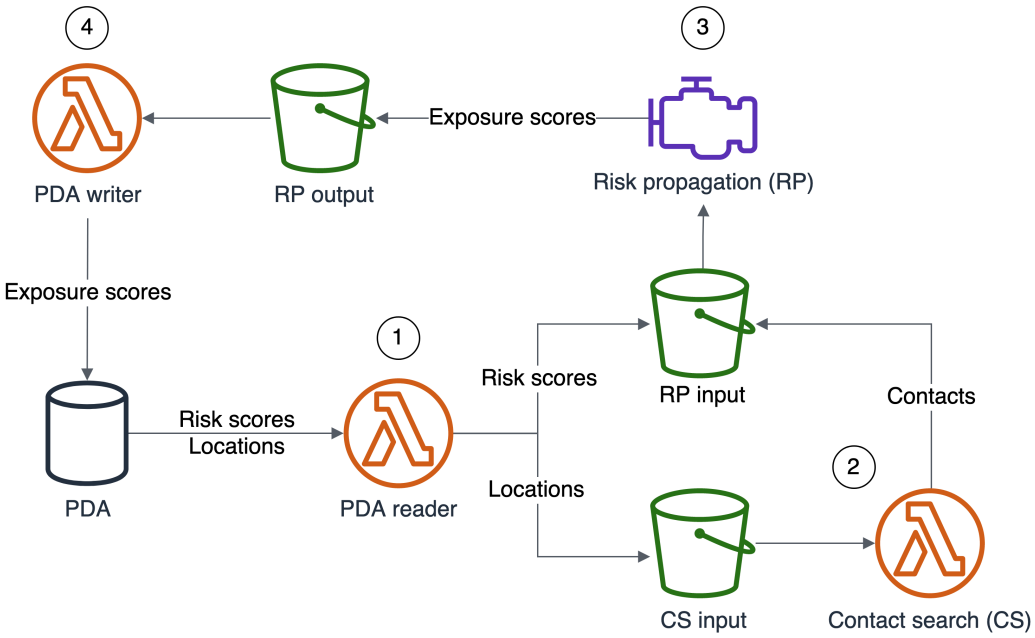

Figure A.1: ShareShareTrace batch-processing architecture. (1) An AWS Lambda function retrieves the risk scores (symptom scores and previously computed exposure scores) and location histories from all user PDAs. Risk scores are transformed into vertex inputs for risk propagation and stored in an Amazon Simple Service (S3) bucket. Location histories are stored in a separate S3 bucket for contact extraction. (2) An AWS Lambda function executes contact search on the location histories to find all contacts, maps them to edge inputs for risk propagation, and stores them in the same S3 bucket that holds the vertex inputs. (3) Amazon Elastic MapReduce (EMR) runs risk propagation as a batch job and outputs the computed exposure scores into an S3 bucket. (4) An AWS Lambda function writes the exposure scores to the PDA of their respective user.

The functionality implemented by all AWS Lambda functions was intended to follow a fan-out design in which one Lambda function would be invoked and

then distribute the work amongst one or more other Lambda functions. For example, the PDA reader Lambda function would retrieve the list of all HATs and then partition that list among several other Lambda functions to retrieve user data in parallel.

The Giraph-based implementation is a literal interpretation of the original ShareTrace algorithm [10]. That is, it assumes a factor graph in which a factor vertex contains the contact times between two users and a variable vertex contains the maximum risk score it has received from a neighboring factor vertex. The algorithm haults when either a given number of iterations has elapsed or the total change in variable vertex values drops below a given threshold.

Several factors prompted the search for an alternative design to Giraph:

1. *Design complexity.* For a relatively straightforward data flow, the architecture in Figure A.1 corresponds to over 4,000 lines of source code. In retrospect, a more suitable approach than manually configured Lambda functions would have been a managed batch-processing or workflow orchestration service, such as AWS Batch or AWS Step Function. Additionally, the low-level design of the Giraph implementation was unnecessarily complex. One-mode projection that is used in Sections A.4 and A.5 would have avoided the complexity of multiple vertex types. Regardless, the implementation was overengineered.

2. *Dependency management incompatability.* A major cause for redesigning the implementation was the dependency version conflicts between Giraph and the libraries used for the ShareTrace implementation. In spite of the

several attempts (e.g., using different library versions, using different versions of Giraph, and forcing specific transitive dependency versions) to resolve these conflicts, a lack of personal development experience and stalled progress prompted me pursue alternatives to Giraph.

3. *Persistent data storage external to the PDA*. One of the core tenets of Dataswift is that the user fully controls the access to their data. However, as shown in Figure A.1, user data is stored in S3 buckets. While it is possible to encrypt S3 objects at rest and automatically delete objects after a certain duration, data persistence to any extent is neither ideal nor desired for a privacy-preserving contact tracing solution.

A.2 Subgraph Actors

Attempting to mimic the design in Section A.1, I implemented risk propagation using the Python library, Ray [79] that "provides a simple, universal API for building distributed applications." While it claims to support actor-based programming, Ray provides relatively basic support compared to Akka, which is used in the current implementation (see Section A.5). Essentially, Ray offers an extension of multiprocessing in which each actor is bound to a process. For risk propagation, I partitioned the factor graph among multiple actors such that each actor maintained a subset of variable vertices *or* factor vertices. The graph topology was stored in shared memory so that it all actors could efficiently access it. The lifetime of this design was brief for the following reasons:

1. *Poor performance.* Interprocess communication is relatively more expensive than intraprocess communication. As a result, by partitioning the vertices such that actors only contained one type of vertex in the bipartite graph, all messages passed during risk propagation were sent to different processes. Unsurprisingly, this manifested in slow runtimes.

2. *Design complexity.* Not using a framework, like Giraph, meant that this implementation required more low-level code to implement actor functionality and message-passing. Regardless of the performance, the overall design of this implementation was poorly organized and overthought.

A.3 Monitor-Worker-Driver (MWD) Framework

Based on the poor runtime performance and complexity of the approach taken in Section A.2, I speculated that centralizing the mutable aspects of risk propagation (i.e., the current value of each variable vertex) would decrease runtime and reduce the implementation complexity. With this is mind, I designed the monitor-worker-driver (MWD) framework, which draws inspiration from the tree of actors design pattern [CITE]. Figure A.2 describes the framework. Listing A.1 provides a partial implementation of the monitor and driver.

For risk propagation, a `RiskMonitor` monitor was implemented. It terminates when any of the following conditions are satisifed. The default behavior is to terminate once the monitor queue is empty.

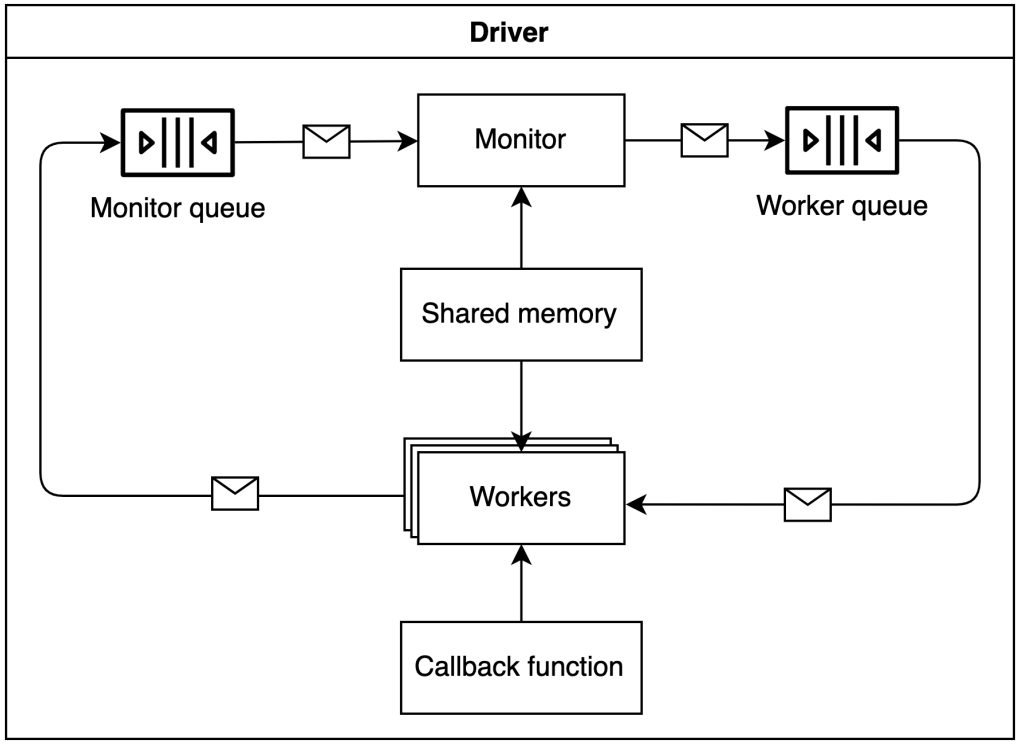

Figure A.2: Monitor-worker-driver framework. The *monitor* is a stateful actor that is responsible for any mutable state of the program. A *worker* (e.g., threads, processes, etc.) is a stateless entity that consumes monitor-produced messages from the *worker queue*. Following the strategy design pattern [33], worker behavior is defined by a *callback function* which may use the attributes of a message to decide on how to process it. Any side effects of processing a message from the worker queue is encapsulated in a message and added to the *monitor queue*. The monitor then consumes messages from the monitor queue, updates the state of the program, and produces messages in response for the workers to consume. The use of queues follows the mediator design pattern [33] in that the monitor and workers communicate indirectly. Any immutable state of the program can be efficiently accessed by both the monitor and the workers in *shared memory*. The *driver* is responsible for initiating the monitor and workers, waiting for the monitor to terminate message-passing, and returning the program output.

```
from abc from ABC
from typing import Any, TypeVar

Q, M = TypeVar("Q"), TypeVar("M")

class BaseMonitor(ABC):
    __slots__ = ()

    def call(self, mqueue: Q, wqueue: Q) -> Any:
        while not self.should_terminate():
            # Get a message from the monitor queue.
            msg = self.get(mqueue)
            # Only process the message if necessary.
            if self.should_process(msg):
                # Optionally modify the message.
                msg = self.transform(msg)
                # Update the state of the monitor.
                self.update(msg)
                # Put the message in the worker queue.
                self.put(wqueue, msg)

class BaseDriver(ABC):
    __slots__ = "monitor", "mqueue", "wqueue", "callback", "n_workers
    "

    def call(self, inputs: Any) -> Any:
        # Perform any necessary setup before starting.
        self.setup(inputs)
        return self.monitor.call(self.mqueue, self.wqueue)
```

Listing A.1: Monitor-worker framework code. The `BaseMonitor` is responsible for observing the message-passing between workers and maintaining any state of the program. The `BaseDriver` defines the rest of the message-passing program by first completing any required setup, and then returning the result from the monitor. It is composed of a worker queue instance, a monitor queue instance, a worker callback function, and the number of workers to instantiate. The type variables `Q` and `M` are used to indicate the types of the queue and message, respectively. The `BaseMonitor.call(Q, Q)` method follows the template design pattern in that it specifies the composition of multiple abstract methods and leaves their implementation to subclasses [33].

- The monitor has received `max_msgs` messages.
- A duration of `max_duration` has elapsed.
- No variable vertex has updated after `n_msgs_early_stop` messages.
- No messages have been received `n_retries` times after `timeout` time.

Only messages that are likely to induce an update to the state of the monitor are processed. The `send_threshold` parameter allows us to vary the strictness of this likelihood. For a message sent by a variable node, the monitor only allows it if the value, scaled by `send_threshold`, is greater than the current value of the variable vertex. This does not guarantee that it will invoke an update after the receiving factor vertex processes it, but it does prevent messages that would obviously not cause a variable vertex to update its value. Similarly, for a factor message, the monitor only allows it if the value, scaled by `send_threshold`, is greater than the current value of the receiving variable vertex. Unlike variable messages, this guarantees against sending ineffective messages. Even if no other terminating condition is specified, the nature of `filter(M)` will gradually cause message passing to terminate. No transformation is applied to messages. The `update(M)` method contains the logic corresponding to the terminating conditions specified earlier. The monitor also updates the value of a variable vertex if the message value is greater than its current value.

The `RiskPropagation` driver exectutes risk propagation. Its `setup(Any)` method (1) creates the factor graph and stores it in shared memory, (2) sets the initial state of the monitor to be the maximum risk score of each variable

vertex, and (3) adds all risk scores to the monitor queue. The `call(Any)` method (1) calls `setup(Any)`, (2) initializes `n_workers` workers and the monitor, and (3) returns the state of the monitor, which is the exposure score of variable vertex.

Compared to the approach in Section A.2, this implementation provides a cleaner design and less communication overhead. However, what prompted me to consider (yet another) an alternative implementation was its scalability. As evidenced by Listing A.1, the MWD framework is just a way of organizing an algorithm around the monitor while loop. Because the monitor processes messages serially, it is a bottleneck for algorithms in which the workers perform fine-grained tasks. Indeed, the Ray documentation notes that the parallelization of small tasks is an antipattern because the interprocess communication cost exceeds the benefit of multiprocessing [CITE]. Unfortunately, the functionality of a variable vertex and factor vertex is too fine-grained, so the scalability of the MWD framework is no better than a serial implementation.

A.4 Subnetwork Actors with Projection

A.5 Thinking Like a Vertex with Actors

Improvements: 1. Distributed; a drawback of all but the first approach 2. Decentralized; a drawback of all previous approaches 3. Robust; allows for delays in the changes of the contact graph with caching 4. Scalable and performant

Drawbacks: 1. Concurrent execution is difficult to reason about

A.6 Location-Based Contact Tracing

- Motivation: Google/Apple API prevents exporting Bluetooth EphIDs
- Other location-based contact tracing approaches

A.6.1 System Model

The system model is very similar to previous work [9, 10] and designs. The only difference is that user geolocation data is collected instead of Bluetooth ephemeral identifiers. Figure A.3 shows the modified dataflow[1]. *Geohash-*

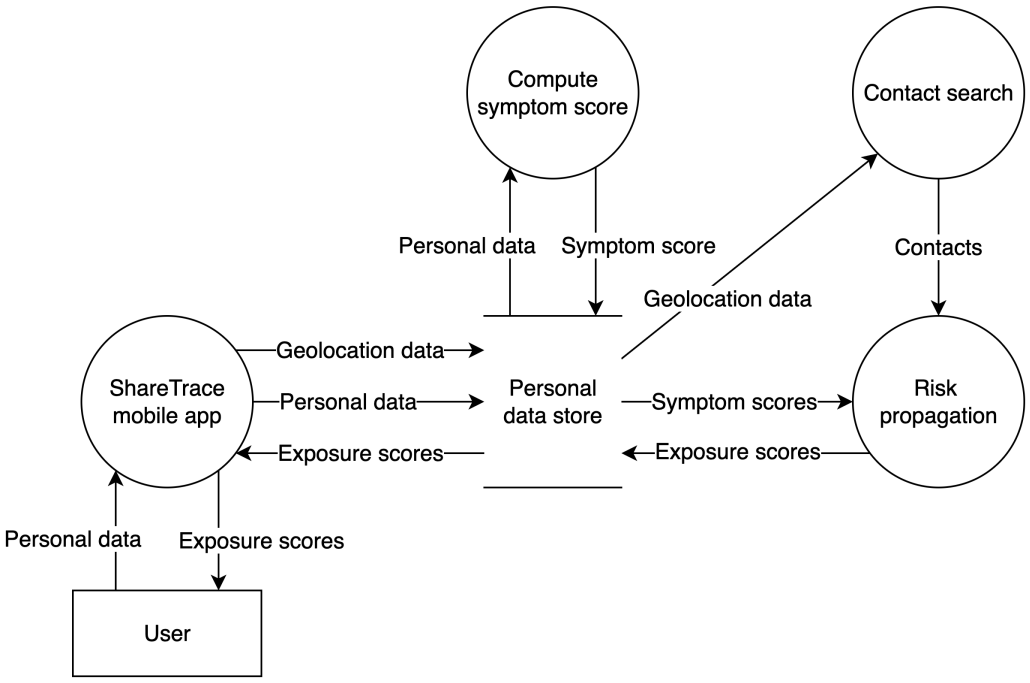

Figure A.3: Locationbased ShareTrace dataflow. This requires that risk propagation is executed in a centralized setting since all user geolocation data is needed to construct the contact network.

[1]See footnote 1 (p. 6) for the definition of a dataflow diagram.

ing is a publicdomain encoding system that maps *geographic coordinates* (i.e., latitudelongitude ordered pairs [74, p. 5]) to alphanumeric strings called *geohashes*, where the length of a geohash is correlated with its geospatial precision [68]. To offer some basic privacy, a user's precise geolocation history is obfuscated ondevice by encoding geographic coordinates as geohashes with 8character precision which corresponds to a region of 730m^2.

A.6.2 Contact Search

A *contact* follows the definition of (2.1), where the contact time indicates the most recent time at which two users were proximal for a contiguous duration of at least $\delta \in$. Each user has a *geolocation history*

$$H = \left[(_i, \ell_i) \mid \ \forall i \in_{[1,H.length]} (_i <_{i+1})\right],$$

a temporally ordered sequence of timestamped geolocations. It is assumed that

1. geolocation histories are not recorded on a fixed schedule, and
2. a user remains at a geolocation until the next geolocation is recorded.

By assumption 1, any two geolocation histories can be "out of alignment" such that they are of different length with interleaving timestamps. Geolocation histories G, H can be *aligned* by *padding* such that

$$n = G.length = H.length,$$

and *temporally interpolating* such that

$$\forall i \in_{[1,n]} (G[i] = H[i]).$$

By assumption 2, it is most appropriate to use *previous interpolation*: given timestamped geolocations $(_i, \ell_i), (_j, \ell_j)$ such that $_i <_j$, all intermediate geolocations are defined as $(_k, \ell_i)$ for all $_k \in [_{i,j})$. In practice, time is a discrete variable that is recorded with fixed precision (e.g., seconds). Let $T \in$ be the *time period* between two consecutive timestamps $_{i,i+1}$ such that $_{i+1} =_i +T$. Then previous interpolation between timestamps $_{i,j}$ such that $_j >_i$ results in $T \cdot (_j -_i - 1)$ intermediate geolocations.

Finding the most recent contact between two users from their aligned geolocation histories is similar to finding the last $k$length common substring between two strings, where each symbol represents a timestamped geolocation. The difference lies in how the start and end of the contact time interval is defined. By assumption 2, the start (end) of a contact time interval is defined as the earlier (ref. later) timestamp of the two first (ref. last) timestamped geolocations in the sequence where the two histories differ. Figure A.4 provides a visual example.

A.6.2.1 Naive Contact Search

A naive approach to finding all contacts amongst a set of geolocation histories $\mathcal{H}$ is to compare all unordered pairs. For a given pair of aligned geolocation histories, the idea is to maintain a pointer to the previous and current index in

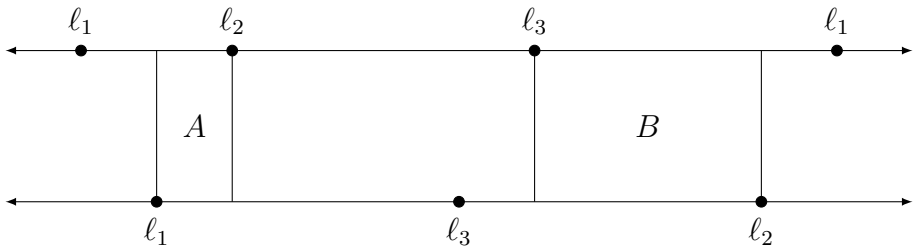

Figure A.4: Contact search with two geolocation histories. Each line denotes time, increasing from left to right. A point ℓ_i is a geolocation and occurs relative in time with respect to the placement of other points. Region B defines the contact interval as it is of sufficient duration and occurs after A.

each history, advancing the pair of pointers whose geolocation occurs later in time. Once a common geolocation is found, all pointers are advanced together until the geolocations differ. If the sequence is δcontiguous, where a sequence of timestamped geolocations S is $\delta\textit{contiguous}$ if $S.\textit{length} \geq \delta$ and

$$\forall i \in_{[1,S.\textit{length}]} (S[i+1] = S[i] + T),$$

then it is recorded. The latest such sequence is used to define the contact between the two users. Because only the most recent time of contact is of interest, the procedure can be improved by iterating in reverse and then terminating once a sequence is found. Regardless, this approach takes $\Theta(n^2)$ time, where $n = |\mathcal{H}|$, because all $\frac{n(n-1)}{2}$ unique pairs must be considered.

The NaiveContactSearch operation implements the above procedure. The operation MostRecentContact considers geolocations[2] $\ell_i, \ell_j \in \mathbb{L}$ $\epsilon\textit{proximal}$ (i.e., approximately equal) if $d(\ell_i, \ell_j) \leq \epsilon$, according to some *metric* $d : \mathbb{L}\times\mathbb{L} \rightarrow$ [46, p. 118] and distance $\epsilon \in$. Because geolocations are encoded as geohashes,

[2]The symbol "$\mathbb{L}$" denotes the space of all geolocations.

they must be decoded into geographic coordinates to perform this proximity calculation. Moreover, because geohashing discretizes the coordinate system into a grid of geographic regions, the number of possible geolocations is finite. Thus, the operation returns a δcontiguous, ϵproximal contact c if such a contact exists between the geolocation histories $G, H \in \mathcal{H}$. The AlignHistories operation pads each geolocation history such that the padded values (e.g., $\pm\infty$) do not result in false contact between users.

Naive-Contact-Search($\mathcal{H}$)

1: $\leftarrow \emptyset$
2: **for each** $(G, H) \in$ Unique-Pairs($\mathcal{H}$)
3: $\quad (G, H) \leftarrow$ Align-Histories(G, H)
4: $\quad c \leftarrow$ Most-Recent-Contact(G, H, ϵ, δ)
5: $\quad$ **if** cNIL
6: $\quad\quad \leftarrow \cup\{c\}$
7: **return**

A.6.2.2 Indexed Contact Search

While the **for** loop in NaiveContactSearch is *embarrassingly parallel* [37, p. 14], the naive approach is neither scalable nor efficient. It can be improved by observing that it is necessary, but not sufficient, that a pair of ϵproximal geolocations exists between two geolocation histories for a contact to exist. Therefore, the geolocation histories $\mathcal{H}$ can be indexed into a spatial data structure $\mathcal{I}$ [61, 66, 69] and then only consider the geolocationhistory pairs that share at least one ϵproximal geolocation pair. This approach is

described by the IndexedContactSearch operation.

Line 2 executes a fixedradius nearneighbors search (FR-NNS) [12, 15] for each geolocation in the spatial index $\mathcal{I}$. Formally, given a set of geolocations $\mathcal{L} \subseteq \mathbb{L}$, a metric d, and a distance ϵ, the *fixedradius nearneighbors* of a geolocation $\ell \in \mathcal{L}$ is defined as the subset of ϵ-proximal geolocations [15],

$$\mathcal{N}(\ell) = \{\ell' \in \mathcal{L} \mid d(\ell, \ell') \leq \epsilon\}$$

Note that the set of neighbors $\mathcal{N}(i)$ of user i corresponds to the geolocations that are ϵproximal to *any* of the geolocations in their geolocation history H_i,

$$\mathcal{N}(i) = \bigcup_{\ell \in H_i} \mathcal{N}(\ell).$$

On line 3, the operation Unique-Users maps these near-neighbors back to the associated users, removing any duplicates that may arise from mapping multiple geolocations to the same user. Finally, line 4 maps the set of users $\mathcal{U}$ back to their geolocation histories and runs Naive-Contact-Search on the resultant subset.

Indexed-Contact-Search($\mathcal{H}$)

1: $\mathcal{I} \leftarrow$ Spatially-Index($\mathcal{H}$)
2: $\mathcal{N} \leftarrow$ Fixed-Radius-Near-Neighbors($\mathcal{I}, \epsilon$)
3: $\mathcal{U} \leftarrow$ Unique-Users($\mathcal{N}, \mathcal{H}$)
4: **return** Naive-Contact-Search($\{H_i \in \mathcal{H} \mid i \in \mathcal{U}\}$)

To carry out FRNNS, one approach is to use a *ball tree*, a complete binary tree that associates with each node a hypersphere that contains a subset of

the data [47, 51, 70]. Any metric can be used to perform FRNNS on a ball tree. However, because geolocation is represented as geographic coordinates, metrics that assume a Cartesian coordinate system may be unsuitable. One of the simplest geometric models of the Earth is that of a sphere. Given two geographic coordinates, the problem of finding the length of the geodesic[3] between them is known as the *inverse geodetic problem* [77]. Assuming a spherical Earth, the solution to the inverse problem is to find the length of the segment that joins the two points on a great circle[4].

Let $\theta = \frac{d}{r}$ be the *central angle*, where $d \in$ is the distance between the two points along the great circle of a sphere with radius $r \in$ (see Figure A.5). The *haversine*, or the half "versed" (i.e., reversed) sine, of a central angle θ is

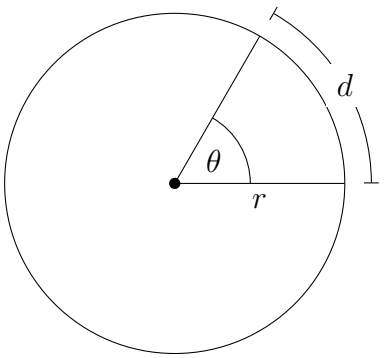

Figure A.5: Central angle of a great circle.

defined as

$$\operatorname{hav}\theta = \frac{\operatorname{vers}\theta}{2} = \frac{1-\cos\theta}{2} = \sin^2\frac{\theta}{2}. \tag{A.1}$$

The greatcircle distance d between two points can be found by inversing A.1

[3]The *geodesic* is the shortest segment between two points on an ellipsoid [58, p. 204].
[4]The *great circle* is the crosssection of a sphere that contains its center [58, p. 165]

and solving,

$$d(\ell_i, \ell_j) = 2 \cdot \arcsin \sqrt{\sin^2 \frac{\phi_i - \phi_j}{2} + \cos \phi_i \cdot \cos \phi_j \cdot \sin^2 \frac{\lambda_i - \lambda_j}{2}},$$

where $\ell_i = (\phi_i, \lambda_i)$ is a latitudelongitude coordinate in radians [16, pp. 157–162].

The choice of the greatcircle distance was primarily driven by its readily available usage in the scikitlearn [17] implementation of a ball tree. If such an approach for discovering contacts were to be used in practice, more advanced *geodetic datum* [58, pp. 71–130] could be used to provide better geospatial accuracy. Moreover, by projecting geodetic coordinates onto the plane, metrics that assume a Cartesian coordinate system could be used instead [58, pp. 265–326].

The running time of SpatiallyIndex is $\Theta\left(|\mathcal{H}| \cdot \log |\mathcal{H}|\right)$ [70]. Assuming the ball tree is balanced[5], the running time of FixedRadiusNearNeighbors to find the ϵproximal neighbors of all geolocations in the spatial index $\mathcal{I}$ is $\Theta(k\,|\mathcal{I}| \cdot \log |\mathcal{I}|)$, where k is the dimensionality of a tree element (i.e., $k = 2$ for geographic coordinates). The running time of UniqueUsers is $\Theta(|\mathcal{N}|)$. While the running time of NaiveContactSearch is $\Theta\left(|\mathcal{U}|^2\right)$, it is likely that $|\mathcal{U}| << |\mathcal{H}|$. Regardless of the input,

$$|\mathcal{H}| \geq |\mathcal{I}| \geq |\mathcal{U}|$$

since $|\mathcal{H}| = |\mathcal{I}|$ only if all geolocations in $\mathcal{H}$ are distinct, and $|\mathcal{I}| = |\mathcal{U}|$ only

[5]The balltree implementation provided by scikitlearn [17] ensures the tree is balanced.

if each user has exactly one geolocation that is distinct from all other users. Dependent upon the input, however, is $|\mathcal{N}|$. In the worst case, $|\mathcal{N}| \in O\left(|\mathcal{I}|^2\right)$ if each geolocation is ϵproximal to all other geolocations. This implies that the overall worstcase running time of IndexedContactSearch is $O(|\mathcal{H}|^2)$. In practice, the running time depends on the geohash precision as well as the geospatial density and mobility behavior of the user population.

A.6.3 Issues

- Privacy and downstream accuracy of risk propagation
- GPS accuracy and power consumption
- Scalability
- Unnecessary overhead; prefer decentralized proximitybased approach

A.6.4 Evaluation

- Data generation
- Naive vs. indexed running time

Appendix B

Data Structures

Let a *dynamic set* S be a mutable collection of distinct elements, and a *dictionary* be a dynamic set that supports insertion, deletion, and membership querying. Each element of S is represented as an object x with attributes such that $x.key$ is a unique identifier for the object x [22, p. 249]. The operations SEARCH, INSERT, DELETE, MINIMUM, and MAXIMUM are mostly consistent with [22, p. 250].

- SEARCH(S, k) returns a pointer x to an element in the set S such that $x.key = k$, or NIL if no such element belongs to S.
- INSERT(S, x) adds the element pointed to by x to the set S.
- DELETE(S, x) removes the element pointed to by x from the set S.
- MINIMUM(S) and MAXIMUM(S) return a pointer x to the minimum and maximum element, respectively, of the totally ordered set S, or NIL if S is empty. Reference [22] only considers the *key* attribute of the

pointers when finding the minimum or maximum element of S. This work, however, will allow other attributes to be used.

Appendix C

Typographical Conventions

C.1 Mathematics

Mathematical typesetting follows the guidance of [29].

C.2 Pseudocode

The pseudocode conventions used in this work mostly follow [22, pp. 21–24].

- Indentation indicates block structure.
- Looping and conditional constructs have similar interpretations to those in standard programming languages.
- Composite data types are represented as *objects*. Accessing an *atttribute* a of an object o is denoted $o.a$. A variable representing an object is a *pointer* or *reference* to the data representing the object. The special value NIL refers to the absence of an object.

- Parameters are passed to a procedure *by value*. That is, the "procedure receives its own copy of the parameters, and if it assigns a value to a parameters, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not" [22, p. 23]. Thus, object attribute assignment "is visible if the calling procedure has a pointer to the same object" [22, p. 24].

- A **return** statement "immediately transfers control back to the point of call in the calling procedure" [22, p. 24].

- Boolean operators **and** and **or** are *short circuiting*.

The following conventions are specific to this work.

- Object attributes may be defined *dynamically* in a procedure.

- Variables are local to the given procedure, but parameters are global.

- The "$\leftarrow$" symbol is used to denote assignment, instead of "=".

- The "=" symbol is used to denote equality, instead of "==", which is consistent with the use of "$\neq$" to denote inequality.

- The "$\in$" symbol is used in **for** loops when iterating over a collection.

- Set-builder notation $\{x \in X \mid \textsc{Predicate}(x)\}$ is used to create a subset of a collection X in place of constructing an explicit data structure.

Bibliography

[1] Reactive streams github repository, 2021. Retrieved 31 May 2021.

[2] Will Abramson, Adam James Hall, Pavlos Papadopoulos, Nikolaos Pitropakis, and William J. Buchanan. A distributed trust framework for privacy-preserving machine learning. In Stefanos Gritzalis, Edgar R. Weippl, Gabriele Kotsis, A. Min Tjoa, and Ismail Khalil, editors, *Trust-Bus 2020: Trust, Priv., Secur. Digit. Bus.*, volume 12395 of *Lect. Notes Comput. Sci.*, 2020.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT, Cambridge, MA, 1985.

[4] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[5] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 19–41, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[6] Nadeem Ahmed, Regio A. Michelin, Wanli Xue, Sushmita Ruj, Robert Malaney, Salil S. Kanhere, Aruna Seneviratne, Wen Hu, Helge Janicke, and Sanjay Jha. A survey of COVID-19 contact tracing apps. *IEEE Access*, 8, 2020.

[7] Samuel Altmann, Luke Milsom, Hannah Zillessen, Raffaele Blasone, Frederic Gerdon, Ruben Bach, Frauke Kreuter, Daniele Nosenzo, Séverine Toussaert, and Johannes Abeler. Acceptability of app-based contact tracing for COVID-19: Cross-country survey evidence, May 2020.

[8] Apple Inc. and Google LLC. Privacy-preserving contact tracing, 2021.

[9] Erman Ayday, Fred Collopy, Taehyun Hwang, Glenn Parry, Justo Elias Karell, James Kingston, Irene Ng, Aiden Owens, Brian Ray, Shirley Reynolds, Jenny Tran, Shawn Yeager, Youngjin Yoo, and Gretchen Young. Sharetrace: A smart privacy-preserving contact tracing solution by architectural design during an epidemic. Technical report, Case Western Reserve University, 2020.

[10] Erman Ayday, Youngjin Yoo, and Anisa Halimi. ShareTrace: An iterative message passing algorithm for efficient and effective disease risk assessment on an interaction graph. In *Proc. 12th ACM Con. Bioinformatics, Comput. Biology, Health Inform.*, BCB 2021, 2021.

[11] Alain Barrat and Ciro Cattuto. Temporal networks of face-to-face human interactions. In Petter Holme and Jari Saramäki, editors, *Temporal Netw.*, Underst. Complex Syst. Springer, 2013.

[12] Jon Louis Bentley. A survey of techniques for fixed radius near neighbor searching. Technical report, Stanford University, 1975.

[13] Christopher M. Bishop. Pattern recognition and machine learning. In M. I. Jordan, Robert Nowak, and Bernhard Schoelkopf, editors, *Inf. Sci. Stat.* Springer, 2006.

[14] Jonas Bonér, Dave Farley, Roland Kuhn, and Marton Thompson. The reactive manifesto, 2014. Retrieved 31 May 2021.

[15] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584, 1995.

[16] Glen Van Brummelen. *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry.* Princeton University Press, Princeton, NJ, 2013.

[17] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[18] Ignacio Cano, Dhruv Mahajan, Giovanni Matteo Fumarola, Arvind Krishnamurthy, Markus Weimer, and Carlo Curino. Towards geo-distributed machine learning. *IEEE Database Eng. Bull.*, 40, 2015.

[19] Andrea Capponi, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry. A survey on mobile crowdsensing systems: Challenges, solutions, and opportunities. *IEEE Commun. Surv. Tut.*, 21(3):2419–2465, 2019.

[20] Centers for Disease Control and Prevention. Quarantine and isolation, 2021. `https://www.cdc.gov/coronavirus/2019-ncov/your-health/quarantine-isolation.html`.

[21] Hyunghoon Cho, Daphne Ippolito, and Yun William Yu. Contact tracing mobile apps for COVID-19: Privacy considerations and related trade-offs. *arXiv*, 2020.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, fourth edition, 2022.

[23] Meggan E. Craft. Infectious disease transmission and contact networks in wildlife and livestock. *Phil. Trans. R. Soc. B*, 370, 2015.

[24] Jesper Dall and Michael Christensen. Random geometric graphs. *Phys. Rev. E*, 66, 2002.

[25] Leon Danon, Ashley P. Ford, Thomas House, Chris P. Jewell, Gareth O. Roberts, Joshua V. Ross, and Matthew C. Vernon. Networks and the epidemiology of infectious disease. *Interdiscip. Perspect. Infect. Dis.*, 2011, 2011.

[26] Aaqib Bashir Dar, Auqib Hamid Lone, Saniya Zahoor, Afshan Amin Khan, and Roohie Naaz. Applicability of mobile contact tracing in fighting pandemic (COVID-19): Issues, challenges and solutions. *Comput. Sci. Rev.*, 38, 2020.

[27] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: A taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, pages 31–40, New York, NY, 2016. Association for Computing Machinery.

[28] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9, 2014.

[29] Chris Dyer, Kevin Gimpel, and Noah A. Smith. A short guide to typesetting math in NLP papers. `http://demo.clab.cs.cmu.edu/cdyer/short-guide-typesetting.pdf`.

[30] Santo Fortunato. Community detection in graphs. *Phys. Rep.*, 486, 2010.

[31] Julie Fournet and Alain Barrat. Contact patterns among high school students. *PLoS ONE*, 9, 2014.

[32] Susan Fowler and Victor Stanwick. *Web Application Design Handbook: Best Practices for Web-Based Software*. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[33] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady

Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, sep 1994.

[34] Raghu K. Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges. *IEEE Commun. Mag.*, 49(11):32–39, 2011.

[35] Mathieu G'enois and Alain Barrat. Can co-location be used as a proxy for face-to-face contacts? *EPJ Data Sci.*, 7, 2018.

[36] Bin Guo, Zhu Wang, Zhiwen Yu, Yu Wang, Neil Y. Yen, Runhe Huang, and Xingshe Zhou. Mobile crowd sensing and computing: The review of an emerging human-powered sensing paradigm. *ACM Comput. Surv.*, 48(1):1–31, 2015.

[37] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Waltham, MA, 2012.

[38] Carl Hewitt and Henry Baker. Laws for communicating parallel processes. Technical report, Massachusetts Institute of Technology, 1977. `http://hdl.handle.net/1721.1/41962`.

[39] Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Phys. Rev. E*, 65, 2002.

[40] Petter Holme and Jari Saramäki. Temporal networks. *Phys. Rep.*, 519, 2012.

[41] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-

distributed machine learning approaching LAN speeds. In *14th USENIX Symp. Networked Syst. Des. Implement. (NSDI 17)*, 2017.

[42] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24, 1977.

[43] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proc. 2017 Symp. Cloud Comput.*, SoCC '17, 2017.

[44] Brian Karrer and M. E. J. Newman. Message passing approach for general epidemic models. *Phys. Rev. E*, 82, 2010.

[45] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20, 1998.

[46] John L. Kelley. *General Topology.* Springer, New York, NY, 1975.

[47] Ashraf M. Kibriya and Eibe Frank.

[48] Andreas Koher, Hartmut H. K. Lentz, James P. Gleeson, and Philipp Hövel. Contact-based model for epidemic spreading on temporal networks. *Phys. Rev. X*, 9, 2019.

[49] Frank R. Kschischang, Brendan J. Frey, and Hans A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory*, 47, 2001.

[50] Christiane Kuhn, Martin Beck, and Thorsten Strufe. Covid notions: Towards formal definitions – and documented understanding – of privacy

goals and claimed protection in proximity-tracing services. *Online Soc. Netw. Media*, 22, 2021.

[51] Neeraj Kumar, Li Zhang, and Shree Nayar.

[52] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78, 2008.

[53] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. A survey of mobile phone sensing. *IEEE Commun. Mag.*, 48(9):140–150, 2010.

[54] R. Greg Lavender and Douglas C. Schmidt. Active object – an object behavioral pattern for concurrent programming, 1996. `https://csis.pace.edu/~marchese/CS865/Papers/Act-Obj.pdf`.

[55] Bo Li and David Saad. Impact of presymptomatic transmission on epidemic spreading in contact networks: A dynamic message-passing analysis. *Phys. Rev. E*, 103, 2021.

[56] Tianshi Li, Jennifer King, Jackie Yang, Yuvraj Agarwal, Jason I. Hong, Cori Faklaris, and Laura Dabbish. Decentralized is not risk-free: Understanding public perceptions of privacy-utility trade-offs in covid-19 contact-tracing apps, May 2020.

[57] Andrey Y. Lokhov, Marc Mézard, Hiroki Ohta, and Lenka Zdeborová. Inferring the origin of an epidemic with a dynamic message-passing algorithm. *Phys. Rev. E*, 90, 2014.

[58] Zhiping Lu, Yunying Qu, and Shubo Qiao. *Geodesy: Introduction to Geodetic Datum and Geodetic Systems*. Springer, Heidelberg, Germany, 2014.

[59] Federica Lucivero, Nina Hallowell, Stephanie Johnson, Barbara Prainsack, Gabrielle Samuel, and Tamar Sharon. COVID-19 and contact tracing apps: Ethical challenges for a social experiment on a global scale. *J. Bioeth. Inq.*, 17, 2020.

[60] Huadong Ma, Dong Zhao, and Peiyan Yuan. Opportunities in mobile crowd sensing. *IEEE Commun. Mag.*, 52(8):29–35, 2014.

[61] Ahmed R. Mahmood, Sri Punni, and Walid G. Aref. Spatio-temporal access methods: a survey (2010 - 2017). *Geoinformatica*, 23:1–36, 2019.

[62] Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, New York, NY, USA, jun 2010. ACM.

[63] Tania Martin, Georgios Karopoulos, José Hernández-Ramos, Georgios Kambourakis, and Igor Nai Fovino. Demystifying COVID-19 digital contact tracing: A survey on frameworks and mobile apps. *Wirel. Commun. Mob. Comput.*, 2020, 2020.

[64] Robert McCune, Tim Weninger, and Greg Madey. Thinking like a vertex:

A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surveys*, 48, 2015.

[65] Cristina Menni, Ana M Valdes, Maxim B Freidin, Carole H Sudre, Long H Nguyen, David A Drew, Sajaysurya Ganesh, Thomas Varsavsky, M Jorge Cardoso, Julia S El-Sayed Moustafa, Alessia Visconti, Pirro Hysi, Ruth C E Bowyer, Massimo Mangino, Mario Falchi, Jonathan Wolf, Sebastien Ourselin, Andrew T Chan, Claire J Steves, and Tim D Spector. Real-time tracking of self-reported symptoms to predict potential COVID-19. *Nat. Med.*, 26, 2020.

[66] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26:1–7, 2003.

[67] James Moody. The importance of relationship timing for diffusion. *Soc. Forces.*, 81, 2002.

[68] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, International Business Machines, 1966.

[69] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. Spatio-temporal access methods: Part 2 (2003 - 2010). *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 33:46–55, 2010.

[70] Stephen M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

[71] Romualdo Pastor-Satorras, Claudio Castellano, Piet Van Mieghem, and Alessandro Vespignani. Epidemic processes in complex networks. *Rev. of Mod. Phys.*, 87, 2015.

[72] Ramesh Raskar, Isabel Schunemann, Rachel Barbar, Kristen Vilcans, Jim Gray, Praneeth Vepakomma, Suraj Kapa, Andrea Nuzzo, Rajiv Gupta, Alex Berke, Dazza Greenwood, Christian Keegan, Shriank Kanaparti, Robson Beaudry, David Stansbury, Beatriz Botero Arcila, Rishank Kanaparti, Vitor Pamplona, Francesco M Benedetti, Alina Clough, Riddhiman Das, Kaushal Jain, Khahlil Louisy, Greg Nadeau, Vitor Pamplona, Steve Penrod, Yasaman Rajaee, Abhishek Singh, Greg Storm, and John Werner. Apps gone rogue: Maintaining personal privacy in an epidemic. *arXiv*, 2020.

[73] Christopher S. Riolo, James S. Koopman, and Stephen E. Chick. Methods and measures for the description of epidemiologic contact networks. *J. Urban Health*, 78, 2001.

[74] Jan Van Sickle. *Basic GIS coordinates*. CRC Press, Boca Raton, FL, 2004.

[75] Lucy Simko, Jack Lucas Chang, Maggie Jiang, Ryan Calo, Franziska Roesner, and Tadayoshi Kohno. COVID-19 Contact Tracing and Privacy: A Longitudinal Study of Public Opinion. *arXiv*, Dec 2020.

[76] Arjun Singhvi, Sujata Banerjee, Yotam Harchol, Aditya Akella, Mark Peek, and Pontus Rydin. Granular computing and network intensive applications: Friends or foes? In *Proc. 16th ACM Workshop Hot Top. Netw.*, HotNets-XVI, 2017.

[77] Lars E. Sjöberg and Masoud Shirazian. Solving the direct and inverse geodetic problems on the ellipsoid by numerical integration. *Journel of Surveying Engineering*, 138:1–36, 2012.

[78] The Apache Software Foundation. Apache Giraph documentation, 2020.

[79] The Ray Team. Ray documentation, 2021. Retrieved 31 May 2021.

[80] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[81] Haohuang Wen, Qingchuan Zhao, Zhiqiang Lin, Dong Xuan, and Ness Shroff. A study of the privacy of COVID-19 contact tracing apps. In Noseong Park, Kun Sun, Sara Foresti, Kevin Butler, and Nitesh Saxena, editors, *Secur. Priv. Commun. Netw.*, volume 335 of *Lect. Notes Inst. Comput. Sci., Soc. Inform. Telecomm. Eng.* Springer Int. Publ., 2020.

[82] Tao Zhou, Jie Ren, Matú š Medo, and Yi-Cheng Zhang. Bipartite network projection and personal recommendation. *Phys. Rev. E*, 76, 2007.

[83] Lorenzo Zino and Ming Cao. Analysis, prediction, and control of epidemics: A survey from scalar to dynamic network models. *IEEE Circuits Syst. Mag.*, 21, 2021.