**Thesis Outline**

**Introduction**

1. Digital contact tracing and COVID-19
2. ShareTrace previous work
3. Contributions of this work
   1. Connection to mobile crowdsensing and self-sovereign identity
   2. More efficient, scalable, online, asynchronous, concurrent message-passing formulation
      1. Actor model, temporal networks, and noniterative
      2. Connections to fixed-point computation
      3. Future work defined in previous ShareTrace work
   3. Message reachability
   4. Differential privacy
   5. Evaluation
      1. Parameter optimization for noniterative convergence and termination
      2. Implementation runtime performance and scalability
      3. Differential privacy analysis

**Actor-Based Contact Tracing**

1. Introduction
   1. Chapter outline
2. Iterative/global risk propagation
   1. Definitions
      1. Risk score
      2. Factor graph
      3. Variable message
      4. Factor message
   2. Algorithm
      1. **Transmission rate generalization.** Transmission rate is any nonincreasing function where is the actor space and is the risk score space.
   3. Refer to previous work on experimental results
   4. Transition to noniterative version: previous work’s future work
3. Noniterative risk propagation
   1. **Reducing communication complexity.** In a message-passing algorithm, this also reduces space and time complexity.
      1. Do not include contacts in the factor that are not at least duration
         1. Avoids if-statement to check contact duration
      2. Do not include contacts with no occurrences or exclusively expired occurrences
      3. Only retain the most recent time of contact
         1. Avoids for-loop
      4. Time complexity to compute a factor message for factor :
      5. Do not send null factor messages – implicit over explicit
         1. Rely on a default risk score and score TTL
   2. **Enabling distributed computing privacy preservation.** In a distributed or decentralized setting, the entire factor graph is unknown. Instead, only a user’s contacts and risk scores are known (i.e., the neighborhood of a variable vertex). Applying one-mode projection onto the variable vertices and associating contact time onto the edges incident to each pair of variable vertices produces a contact network where a vertex represents a user and an edge indicates that the users associated with the incident vertices came in contact. Each vertex is responsible for computing variable and factor messages. Variable messages represent state and factor messages are passed between vertices to update the state of a vertex. At this point, risk propagation may still run synchronously in a global or local fashion (implementation details). However, it is now possible to arbitrarily partition the network to preserve user privacy since.
      1. Refer to projected factor graph a contact network comprised of **nodes** or **actors** to emphasize computation and state; not just abstract data structure that defines the topology. Refer to the neighbors of a node or actor as **contacts** to emphasize the network semantics.
      2. IEEE ShareTrace “misses the mark” since it message-passing semantics are defined between nodes, but the message-passing description is defined between subnetworks of nodes. Practically, while subnetworks may be used in an implementation, it is clearer to define risk propagation as messages between nodes.
   3. **Asynchronous message passing.** In a distributed setting, fully asynchronous message-passing can be more time-efficient. However, we cannot rely on an iteration count or inter-iteration difference as termination conditions. To motivate a new termination condition, it is useful to view risk propagation as a fixed-point computation. That is, risk propagation begins with a prior risk distribution and, by computing variable and factor messages, converges to a posterior risk distribution (i.e., a fixed point). It is also useful to observe the *intent* of computing a message: to update the risk of some other vertex in the network. Thus, a vertex should not pass a message if will not update the risk of another vertex. We can define a temporal *send condition* for each contact that accounts for the time of contact, the time of the risk score, and the value of the previous message that was sent to the target vertex. By establishing a temporal threshold whose expiry is defined by the associated risk score that was sent, we can ensure that a finite number of messages are sent to a given target vertex (assuming a finite number of contacts).
      1. Exponential decay
      2. **Send threshold generalization**. Send threshold is a function f: **A** x **S** -> **R** > 0
      3. Liveness of the contact network proportional to number of messages sent
   4. Risk propagation using the actor model
      1. Define the actor system
      2. Define the message types and message handlers
   5. Risk score caching and synchronization delays
      1. Previous offline implementations do not need to worry about delays between device and PDS.
      2. Proof of correctness?
   6. Message reachability
   7. Propagation triggers
      1. New contact
      2. New symptom score

**Evaluation**

* Risk propagation parameters
* Dataset generation and sampling procedure
* Event, metrics, and settings logging
* Implementation details: JGraphT, Akka, fastuil data structures
* Graph parameters (n = 1000)
  + G(n, m) random
    - New edges m = 0.01N, 0.05N, 0.1N
      * N = n(n - 1) / 2
  + Random regular
    - Degree d = 3, 4, 5
  + Barabasi Albert (requires m0 >= m)
    - Number of initial nodes m0 = 3
    - Number of new edges m = 1, 2, 3
  + Watts Strogatz
    - Nearest neighbors k = 2, 4, 6
    - Rewiring probability p = 0.2, 0.5, 0.8
* Distributions (risk score value, risk score timestamps, and contact timestamps
  + Normal(μ, σ^2):
    - Standard: (0, 1)
    - Narrow: (0, 0.2)
    - Wide: (0, 5)
  + Beta(α, β):
    - Modest right skew: (5, 2)
    - Modest left skew: (2, 5)
    - Extreme right skew: (10, 0.1)
    - Extreme left skew: (0.1, 10)
  + Uniform(0, 1)
* Research questions
  + Parameter experiments
    - What value of send coefficient optimizes for completeness and efficiency?
    - How do different values of transmission rate affect completeness and efficiency?
  + Topology
    - Do random contact networks behave differently than complex contact networks?
      * Experiments:
        + Random graphs: G(n, M) and regular
        + Complex: Watts, Barabosi, scale-free
      * Significance: If random graphs behave similarly to complex graphs, then they can be used to verify hypotheses about contact tracing that use risk propagation.
    - Do synthetic contact networks behave differently than real-world contact networks?
      * Significance: If synthetic contact networks behave similarly to real-world networks, then they can be used to verify hypotheses about contact tracing that use risk propagation.
      * Experiments:
        + Synthetic: use previous research question results
        + Real-world: all 6 SocioPatterns networks
  + Distributions
    - How does the risk score value distribution affect runtime and number of messages?
    - How does the risk score timestamp distribution affect runtime and number of messages?
    - How does the contact timestamp distribution affect runtime and number of messages?
  + Scalability
    - How well does risk propagation scale? (verify linear scaling in the number of contacts)
    - Experiments:
      * Random and complex synthetic graphs with varying topologies and sizes
* Discussion
  + Evaluation results
  + Mobile crowdsensing application
  + Trust Over IP stack / SSI
    - Include IP document

**Conclusion**

* Summary
* Implications
* Future work

**Appendices**

1. Previous work
   1. Giraph
   2. Subnetwork actors
      1. Differences to actor-based design:
         1. No partitioning: actors are scheduled
         2. No “initial score” – current score is either a symptom score or a propagated risk score
         3. Graph changes as the (online) algorithm runs, i.e., not a snapshot
            1. Spread is [non-conserved](https://en.wikipedia.org/wiki/Network_theory#Spread)
         4. No longer weights based on risk score time, just TTL (see Algorithm 3)
   3. Location-based contact tracing
2. Conventions

**Notes**

**Pedantic topics**

* Contact: mobile devices vs users bodies – “users came into contact” implies their devices
* Agent vs actor
* Hyperlocal interaction graph vs contact network
  + The latter is preferred

**Akka**

* [Actor](https://doc.akka.io/docs/akka/current/general/actors.html)
  + Upon receiving a message, an actor can…
  + Send a finite number of messages to Actors it knows
  + Create a finite number of new Actors
  + Designate the behavior to be applied to the next message
* [Mailbox](https://doc.akka.io/docs/akka/current/general/actors.html#mailbox)
  + Sending multiple messages to the same target from the same actor, on the other hand, will enqueue them in the same order
  + Default mailbox implementation: FIFO
* [Java Memory Model](https://doc.akka.io/docs/akka/current/general/jmm.html)
  + Akka guarantees the following two “happens before” rules:
    - **The actor send rule**: the send of the message to an actor happens before the receive of that message by the same actor.
    - **The actor subsequent processing rule**: processing of one message happens before processing of the next message by the same actor.
    - In layman’s terms this means that changes to internal fields of the actor are visible when the next message is processed by that actor. So fields in your actor need not be volatile or equivalent.
* [Message Delivery Reliability](https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html)
  + These are the rules for message sends (i.e., tell operations):
    - At-most-once delivery (common in other actor model frameworks)
    - Message ordering per sender-receiver pair (specific to Akka)
      * The rule more specifically is that for a given pair of actors, messages sent directly from the first to the second will not be received out-of-order. The word directly emphasizes that this guarantee only applies when sending with the tell operator to the final destination, not when employing mediators or other message dissemination features (unless stated otherwise).
* [Supervision and Monitoring](https://doc.akka.io/docs/akka/current/general/supervision.html)
  + **Message**: If an exception is thrown while a message is being processed (i.e. taken out of its mailbox and handed over to the current behavior), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don’t want a system to livelock (so consuming a lot of cpu cycles without making progress).
  + **Mailbox**: If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox will be there as well.
  + **Actor**: If code within an actor throws an exception, that actor is suspended and the supervision process is started. Depending on the supervisor’s decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated.
  + Restarting does not affect the ActorRef, so other actors can still communicate with the restarted actor ([source](https://doc.akka.io/docs/akka/current/general/supervision.html#what-restarting-means)).
  + Akka supports persistence via [durable state](https://doc.akka.io/docs/akka/current/typed/index-persistence-durable-state.html) and [event sourcing](https://doc.akka.io/docs/akka/current/typed/index-persistence.html) to ensure restarted actors can be restored to their previous state.