

CASE WESTERNRESERVE 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.

Chapter 1

Evaluation

1.1 Reference Implementation

A reference implementation of ?? is available on GitHub¹. Actors are implemented using the Akka toolkit², which offers high performance for large-scale actor systems. Experimental results indicate that the reference implementation can reliably handle contact networks with 1 million individuals and 10 million contacts, which makes it ideal for small-scale experiments. In addition to using the Akka toolkit, several other optimizations are implemented:

- To reduce the size of event logs and result files, individual actor identifiers follow zero-based numbering and event records are serialized using the Ion format³ with shortened field names.

¹<https://github.com/cwru-xlab/sharetrace-akka>

²<https://doc.akka.io/docs/akka/2.8.5/typed/index.html>

³<https://amazon-ion.github.io/ion-docs>

- To reduce memory usage, FastUtil⁴ data structures are used, including a specialized integer-based JGraphT⁵ graph implementation [5]. Also, singletons [3], primitive data types, and reference equality are preferred where feasible and do not impact readability.
- To reduce runtime and increase throughput, logging is performed asynchronously with Logback⁶ and the LMAX Disruptor⁷.

Figure 1.1 shows the dependencies among the application components. Contextualizing this implementation with prior implementations of the driver-monitor-worker (DMW) framework (see ??), **RiskPropagation** is the driver, **Monitor** is the monitor, and **User** is the worker. The key difference between this implementation and previous implementations of the DMW framework is that the workers are stateful, which is necessary for decentralization.

Main → **Runner** → **RiskPropagation** → **Monitor** → **User** → **Contact**

Figure 1.1: Arrow diagram of the reference implementation.

?? describes the behavior of **User** and **Contact**. In order to evaluate **RiskPropagation**, each **User** also logs the following types of **UserEvent**:

- **ContactEvent**: logged when the **User** receives an unexpired **Contact-Message**; contains the **User** identifier, the **Contact** identifier, and the contact time.

⁴<https://fastutil.di.unimi.it>

⁵<https://jgrapht.org>

⁶<https://logback.qos.ch/index.html>

⁷<https://lmax-exchange.github.io/disruptor>

- **ReceiveEvent**: logged when the **User** receives an unexpired **RiskScoreMessage**; contains the **User** identifier, the **Contact** identifier, and the **RiskScoreMessage**.
- **UpdateEvent**: logged when the **User** updates its exposure score; contains the **User** identifier, the previous **RiskScoreMessage**, and the current **RiskScoreMessage**.
- **LastEvent**: logged when the **User** receives a **PostStop** Akka signal⁸ after the **Monitor** has stopped; contains the **User** identifier and the time of logging the last event, besides **LastEvent**; used to detect the end time of message passing.

For reachability analysis, **RiskScoreMessage** contains the identifier of the **User** that propagated the message and the identifier of the **User** that first sent the message.

Monitor is an actor that is responsible for transforming the **ContactNetwork** into a collection of **User** actors and terminating when no **UpdateEvent** has occurred for a period of time. As with **User** actors, the **Monitor** logs several types of **LifecycleEvent**, the meanings of which should be self-explanatory:

- | | |
|---------------------------|------------------------------|
| • CreateUsersStart | • SendRiskScoresStart |
| • CreateUsersEnd | • SendRiskScoresEnd |

⁸<https://doc.akka.io/docs/akka/current/typed/actor-lifecycle.html#stopping-actors>

- `SendContactsStart`
- `RiskPropagationStart`
- `SendContactsEnd`
- `RiskPropagationEnd`

`RiskPropagation` logs execution properties, creates an Akka `ActorSystem` that creates a `Monitor` actor and sends it a `RunMessage`, and then waits until the `ActorSystem` terminates. Each execution of `RiskPropagation` is associated with a unique key that is included in each event record as mapped diagnostic context (MDC)⁹.

The `Runner` specifies how `RiskPropagation` is created and invoked, usually through some combination of statically defined behavior and runtime configuration.

Finally, `Main` is the entry point into the application. It is responsible for parsing `Context`, `Parameters`, and `Runner` from configuration and invoking `Runner` with `Context` and `Parameters` inputs. `Context` makes application-wide information accessible, such as the system time and user time¹⁰, a pseudorandom number generator, `Runner` configuration, and loggers. `Parameters`, as the name suggests, is a collection of parameters that modify the behavior of the `Monitor`, `User`, and `Contact`.

In order to analyze the logs that were generated during the execution of `RiskPropagation`, they are transformed into a tabular dataset as follows:

1. Load the execution properties for all executions of `RiskPropagation`

⁹<https://logback.qos.ch/manual/mdc.html>

¹⁰System time is always the wall-clock time and is included in each logged event record. User time is configurable to either be the wall-clock time or fixed at the reference time. The latter ensures that no `RiskScoreMessage` and `ContactMessage` expires across executions of `RiskPropagation`.

that are associated with the same configuration.

2. Process the stream of event records with one `EventHandler` per execution of `RiskPropagation`.
3. Collect the results from each `EventHandler` and store them in a file.
4. To analyze different configurations of `RiskPropagation`, load multiple result files and augment the results of each `RiskPropagation` execution with its execution properties.
5. Flatten the resulting data structure and store the tabular dataset.

For evaluation, the following event handlers were implemented:

- **Reachability:** aggregates `ReceiveEvents` that involve a distinct sender and receiver to determine the influence set cardinality, source set cardinality, and message reachability of each `User`.
- **Runtimes:** aggregates `LifecycleEvents` and `LastEvents` to determine the runtime of creating `Users`, sending `ContactMessages`, sending `RiskScoreMessages`, message passing, and the overall execution of `RiskPropagation`. Message passing runtime is the time elapsed from the start of sending `RiskScoreMessages` until the last `LastEvent`.
- **UserEventCounts:** aggregates `UserEvents` to determine the frequency of each subtype for each `User`.
- **UserUpdates:** aggregates `UpdateEvents` to determine the new exposure score of each `User` and the change in value.

1.1.1 Experimental Design

The following research questions were the focus of evaluation:

1. How do the send coefficient and tolerance affect the accuracy and efficiency of risk propagation?
2. Do the distributions of risk scores and contact times affect the runtime of risk propagation?
3. How does the contact network topology affect the runtime of risk propagation?

Several graphs

Barabasi-Albert graphs [1] are parametrized by the order n , the initial order n_0 , and the increase in size m_0 upon each incremental increase in order. The latter two parameters are determined by solving (1.1), where $\text{frac}(x)$ is the fractional part of a real number x .

$$\begin{aligned}
 & \arg \min_{n_0, m_0} \quad \text{frac}(m_0) \\
 & \text{subject to} \quad n_0 \in [1 \dots n - 1], \\
 & \quad \quad \quad m_0 \in [1 \dots n_0], \\
 & \quad \quad \quad m_0 = \frac{2m - n_0(n_0 - 1)}{2(n - n_0)}
 \end{aligned} \tag{1.1}$$

Erdős-Rényi $G_{n,m}$ random graphs [2] are parametrized by the order n and the size m . Random regular graphs [4] are parametrized by the order n and, using the degree sum formula, the degree $d = \lfloor \frac{2m}{n} \rfloor$. Lastly, Watts-Strogatz graphs

Parameter	Default value
Transmission rate, α	0.8
Send coefficient, γ	1
Tolerance, ϵ	0
Time buffer, β	2 days
Risk score expiry, T_s	14 days
Contact expiry, T_c	14 days
Flush timeout	3 seconds
Idle timeout	1 minute
Seed	12 345

Table 1.1: Default parameter values for evaluation.

[6] are parametrized by the order n , the rewiring probability p and the number of nearest neighbors $k = d + (d \bmod 2)$, which must be even.

Common:

- Fixed user time
- Sampling procedure to generate dataset values: Given the probability density function f_X and the cumulative distribution function F_X of a random variable X , sample a value $x \sim f_X$ and evaluate $F_X(x)$.

Parameter experiments:

- Graph parameters: $n = 10^4$, $m = 5 \cdot 10^4$
- Distributions: All combinations of uniform and standard normal
- Parameters: $\gamma \in \{0.1x \mid x \in [8 \dots 20]\}$, $\epsilon \in \{0.001x \mid x \in [1 \dots 10]\}$
- Contact networks: 5 with distinct risk scores and contact times

Runtime baseline experiment:

- Graph parameters: $n = 10^4$, $m = 10^5$
- Distributions: All combinations of uniform and standard normal
- Contact networks: 1 burn-in + 5 contact networks with distinct risk scores and contact times

Runtime experiment:

- Graph parameters: $n \in \{10^5 x \mid x \in [1 \dots 10]\} \times m \in \{10^6 x \mid x \in [1 \dots 10]\}$
- Distributions: only uniform
- Contact networks: 1 burn-in + 5 contact networks with distinct risk scores and contact times

Bibliography

- [1] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks”. In: *Science* 286.5439 (1999), pp. 509–512. DOI: 10.1126/science.286.5439.509 (cit. on p. 7).
- [2] Paul Erdős and Alfréd Rényi. “On random graphs I.” In: *Publicationes Mathematicae* 6.3–4 (1959), pp. 290–297. DOI: 10.5486/pmd.1959.6.3-4.12 (cit. on p. 7).
- [3] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995 (cit. on p. 3).
- [4] Jeong Han Kim and Van H. Vu. “Generating random regular graphs”. In: *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*. 2003, pp. 213–222. DOI: 10.1145/780542.780576 (cit. on p. 7).
- [5] Dimitrios Michail et al. “JGraphT—A Java library for graph data structures and algorithms”. In: *ACM Transactions on Mathematical Software* 46.2 (2020), pp. 1–29. DOI: 10.1145/3381449 (cit. on p. 3).

- [6] Duncan J. Watts and Steven H. Strogatz. “Collective dynamics of ‘small-world’ networks”. In: *Nature* 393.6684 (1998), pp. 440–442 (cit. on p. 8).