Chapter 1

Evaluation

1.1 Reference Implementation

A reference implementation of ?? is available on GitHub[1]. Actors are implemented using the Akka toolkit[2], 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 format[3] 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, FastUtil[4] data structures are used, including a specialized integer-based JGraphT[5] graph implementation (Michail et al., 2020). Also, singletons (Gamma et al., 1995), 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 Logback[6] and the LMAX Disruptor[7].

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 `ContactMessage`; contains the `User` identifier, the `Contact` identifier, and the

[4] `https://fastutil.di.unimi.it`
[5] `https://jgrapht.org`
[6] `https://logback.qos.ch/index.html`
[7] `https://lmax-exchange.github.io/disruptor`

contact time.

- 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 signal[8] 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 from which the message originated.

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
- CreateUsersEnd
- SendRiskScoresStart
- SendRiskScoresEnd

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

- SendContactsStart
- SendContactsEnd
- RiskPropagationStart
- 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.

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 time[9], 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`.

An experiment typically composed of multiple configuration files in order to vary an aspect of the

An execution is defined by one or more invocations of risk propagation that are associated with the same configuration file.

Multiple invocations may be needed to compute statistics when the data generation process is stochastic.

[9] System 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 is ensures that no `RiskScoreMessage` and `ContactMessage` expires across executions of `RiskPropagation`.

An execution may involve multiple invocations of risk propagation in order to collect multiple results from the same configuration. This is particularly relevant when the data generation process involves

Analysis sequence:

1. Load execution properties.
2. Stream the event log and process each record by one or more `EventHandler`s.
3. Put the results from each `EventHandler` in a `Results` object.
4. Transform the `Results` instance it into a tabular dataset.
5. Analyze the dataset.

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. What is the runtime performance of risk propagation?

Barabasi–Albert graphs (Barabàsi and Albert, 1999) are parametrized by the order n, the initial order n_0, and the size increase m_0 upon each incremental increase to the order. The latter two parameters are determined by solving

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

Table 1.1: Default parameter values for evaluation.

(1.1), where $\mathrm{frac}(x)$ is the fractional part of a real number x.

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

Erdös–Rényi $G_{n,m}$ random graphs (Erdös and Rényi, 1959) are parametrized by the order n and the size m. Random regular graphs (Kim and Vu, 2003) are parametrized by the order n and, using the degree sum formula, the degree $d = \lfloor \frac{2m}{n} \rfloor$. Lastly, Watts–Strogatz graphs (Watts and Strogatz, 1998) 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.

The default Akka configuration[10].

[10] `https://doc.akka.io/docs/akka/current/general/configuration-reference.html`

- Fixed clock time
- Monitor actor:
  - PinnedDispatcher
  - Thread pool executor
  - No core timeout
- User actors:
  - Dispatcher
  - Thread pool executor
  - 100 throughput
  - Max pool size: 2 147 483 647 (max Java integer value)
- 5 contact networks with distinct risk scores and contact times
- 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 $F_X(x)$.

Parmeter experiments:

- $n = 10^4$, $m = 5 \cdot 10^4$
- Distributions: uniform, standard normal
- Send coefficients: 0.8–2.0, in increments of 0.1
- Tolerance: 0.001–0.01, in increments of 0.001

- All 9 distribution combinations: uniform, standard normal
- 5 contact networks with distinct risk scores and contact times

Runtime baseline experiment:

- $n = 10^4$, $m = 10^5$
- All 9 distribution combinations: uniform, standard normal
- 1 burn-in + 5 contact networks with distinct risk scores and contact times
- Log lifecycle events and last event for message-passing runtime

Runtime experiment:

- Cross product of $n \in \{\, 10^5 x \mid x \in [1, 10] \,\}$ and $m \in \{\, 10^6 x \mid x \in [1, 10] \,\}$
- Uniform distribution for all 3 data types
- 1 burn-in + 5 contact networks with distinct risk scores and contact times
- Log lifecycle events and last event for message-passing runtime

1.1.2 Results

Bibliography

[1] Albert-Làszlò Barabàsi and Rèka 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. 5).

[2] Paul Erdös and 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. 6).

[3] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995 (cit. on p. 2).

[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*. STOC '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 213–222. DOI: `10.1145/780542.780576` (cit. on p. 6).

[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. 2).

[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. 6).