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 [6]. 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 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 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 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 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
- 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. 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 time[10], 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`

[10] 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`.

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 `ReceiveEvent`s that involve a distinct sender and receiver to determine the influence set cardinality, source set cardinality, and message reachability of each `User`.

- `Runtimes`: aggregates `LifecycleEvent`s and `LastEvent`s to determine the runtime of creating `User`s, sending `ContactMessage`s, sending `RiskScoreMessage`s, message passing, and the overall execution of `RiskPropagation`. Message passing runtime is the time elapsed from the start of sending `RiskScoreMessage`s until the last `LastEvent`.

- `UserEventCounts`: aggregates `UserEvent`s to determine the frequency of each subtype for each `User`.

- `UserUpdates`: aggregates `UpdateEvent`s to determine the new exposure score of each `User` and the change in value.

1.1.1 Experimental Design

The following research questions (RQs) were the focus of evaluating asynchronous risk propagation:

1. How do the send coefficient and tolerance affect accuracy and efficiency?

2. How do the distributions of risk scores and contact times affect runtime?

3. How does the contact network topology affect runtime?

RQ 2 and RQ 3 focus on benchmarking the reference implementation. More advanced simulation-based analysis of ShareTrace with COVI-AgentSim [4] is the subject of future work.

While RQ 3 explicitly evaluates the impact of contact network topology on runtime, all research questions were assessed using the same random graphs: Barabasi-Albert graphs [1], Erdös-Rényi $G_{n,m}$ graphs [2], Watts-Strogatz graphs [8], and random regular graphs [5]. These graphs were selected because they exhibit, to varying extents, aspects of real-world complex networks [7], such as contact networks; they are available in the JGraphT library; and they all are parametric, either directly or indirectly, in the size and order of the network. The latter property allowed the effects of the topology to be isolated.

The following describes the parametrization of each type of contact network. Barabasi-Albert graphs 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 frac(x) is the fractional part of a real number x.

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

Erdös-Rényi $G_{n,m}$ graphs are parametrized by the order n and the size m. Random regular graphs are parametrized by the order n and, using the degree sum formula, the degree $d = \lfloor \frac{2m}{n} \rfloor$. Lastly, Watts-Strogatz graphs [8] 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.

Table 1.1 specifies the default parameter values and seed that were used during evaluation. Table 1.2 specifies the configurations used to evaluate each RQ. All RQs were evaluated with a fixed user time. The following sampling process was used to generate risk scores and contact times. 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 $c \cdot F_X(x)$ for some scalar $c \in \mathbb{R}$. Risk scores are composite data types, so risk score values and risk score times were generated from independent probability distributions. Because risk scores are probabilities, $c = 1$ was used to scale the values. When sampling the times of risk scores and contacts, $c = T_s$ and $c = T_c$ were used, respectively.

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.

Attribute	RQ 1	RQ 2	RQ 3
Order n and size m	$n = 10^4$ $m = 5 \cdot 10^4$	$n = 10^4$ $m = 5 \cdot 10^4$	$n \in \{ 10^5 x \mid x \in [1 \,..\, 10] \}$ $\times$ $m \in \{ 10^6 x \mid x \in [1 \,..\, 10] \}$
Parameters	$\gamma \in \{ 10^{-1} x \mid x \in [8 \,..\, 20] \}$ $\epsilon \in \{ 10^{-3} x \mid x \in [1 \,..\, 10] \}$	Defaults	Defaults
Distributions	$\{\text{Uniform}, \text{Standard normal}\}^3$	$\{\text{Uniform}, \text{Standard normal}\}^3$	Uniform
Repetitions	5	1 burn-in + 5	1 burn-in + 5
Networks evaluated	160 (40 per type) per parameter	160 (40 per type)	400 (100 per type)

Table 1.2: Experiment configurations. See Table 1.1 for default parameter values. The send coefficient and tolerance were evaluated independently for RQ 1. The notation X^k is used to denote the k-ary Cartesian power of the set X. A "burn-in" repetition was used for RQ 2 and RQ 3 to avoid measuring the impact of Java class loading.

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. 6).

[2] Paul Erdös and Alféd 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] Prateek Gupta et al. *COVI-AgentSim: An agent-based model for evaluating methods of digital contact tracing*. 2020. arXiv: 2010.16004 [cs.CY] (cit. on p. 6).

[5] 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. 6).

[6] 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).

[7] M. E. J. Newman. “The structure and function of complex networks”. In: *SIAM Review* 45.2 (2003), pp. 167–256. DOI: `10.1137/S003614450342480` (cit. on p. 6).

[8] Duncan J. Watts and Steven H. Strogatz. “Collective dynamics of ‘small-world’ networks”. In: *Nature* 393.6684 (1998), pp. 440–442 (cit. on pp. 6, 7).