Latency-aware RDMSim: Enabling the Investigation of Latency in Self-Adaptation for the Case of Remote Data Mirroring

Sebastian Götz
Technische Universität Dresden
Germany
sebastian.goetz1@tu-dresden.de

Nelly Bencomo
Durham University
United Kingdom
nelly@acm.org

Huma Samin
Durham University
United Kingdom
huma.samin@durham.ac.uk

ABSTRACT

Self-adaptive systems are able to adapt themselves according to changing contextual conditions to ensure a set of predefined objectives (e.g., certain non-functional requirements like reliability) is reached. For this, they perform adaptation actions which have an effect on the objectives. But, this effect is not reached immediately. Instead, there is a latency between performing an adaptation action and the effect on the objective. In this paper, we present an exemplar which allows to investigate this latency of adaptation actions for a remote data mirroring system (RDMs) based on a previous exemplar, which wasn't latency-aware. The purpose of an RDM is to replicate a data package among multiple locations (called mirrors) to protect against data loss and unavailability. We present a simulation framework for RDM systems, which offers various adaptation actions: changing the number of mirrors, changing the number of links per mirror and changing the topology of the network. The framework is able to predict the effect on three selected non-functional requirements (cost, performance and reliability) and how long a reconfiguration will take, i.e., the latency. We show how to use this information to construct a simple, exemplary optimiser.

CCS CONCEPTS

• **Software and its engineering → Abstraction, modeling and modularity**; **Model-driven software engineering**; *Data flow architectures.*

KEYWORDS

latency-awareness, self-adaptation, models@run.time

ACM Reference Format:
Sebastian Götz, Nelly Bencomo, and Huma Samin. 2024. Latency-aware RDMSim: Enabling the Investigation of Latency in Self-Adaptation for the Case of Remote Data Mirroring. In *19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '24), April 15–16, 2024, Lisbon, AA, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3643915.3644106

1 INTRODUCTION

Modern software needs to be self-adaptive [13] as it does not run on a single stationary computer anymore. Instead, applications are distributed amongst many machines in the cloud and programmed according to paradigms like server-less computing [10]. A basic ingredient for all of the above systems is, thus, a feedback loop. That is applications continuously monitor themselves and their environment, analyse whether they still do what they are supposed to do, plan how to counter deviations from this plan, and finally execute this plan. This basic principle is well known as MAPE-K [8], an acronym for monitor, analyse, plan, execute, and knowledge. The knowledge in MAPE-K is a runtime model of the current and, if necessary, past states of the system. The introduction of such a runtime model, which is causally connected to the system it represents, is the principle idea of the research field models@run.time [1]. The causal connection [9] defines how a change in the runtime model is reflected in the running system and vice versa.

An interesting and still open question for the causal connection is how to assess the time required for a change on one side to be visible on the other, i.e., the latency. Knowing how long a change takes in the runtime model to be reflected in the underlying system and vice versa allows to specify latency-aware adaptation logic.

Latency-aware self-adaptation introduces a new dimension for optimisation in self-adaptive systems. Besides optimising the objectives of such a system, latency-awareness allows to optimise these objectives over time. If the planner can choose between multiple adaptation actions to improve the objectives of the system, the knowledge about the latency of these actions enables the planner to choose, e.g., the fastest action. In consequence, the planner can now optimise how long a system deviates from its optimum. This includes the possibility to incorporate real-time deadlines for available adaptation actions.

Approaches for latency-aware self-adaptation already exist. For example, the works by Camara et al. and Moreno et al. [2, 3, 11, 12] that use latency information for proactive adaptation. Also, more recently, Keller and Mann provided a literature study on adaptation latency in the context of service-oriented systems [7].

In this paper, we present a simulator framework which can be used to experiment with and investigate the latency of adaptations in the context of remote data mirroring [5, 6]. The framework is based on an existing non-latency-aware exemplar [14] and extends it by an object net of mirrors, links and data packages to enable the investigation of how long adaptation actions require to be completely realised. The framework, the exemplary optimiser, and a test-suite demonstrating various scenarios are provided in an open source repository[1].

The exemplar allows to investigate the latency of several adaptation actions: changing the number of mirrors in the network, switching between three provided topologies and changing the

SEAMS '24, April 15–16, 2024, Lisbon, AA, Portugal

ACM ISBN 979-8-4007-0585-4/24/04.
https://doi.org/10.1145/3643915.3644106

[1] https://github.com/sebastiangoetz/LRDMSimulator

number of outgoing links per mirror. As in the original exemplar, three general objectives are provided: the number of active links representing the reliability of the network, the current bandwidth used by the network representing the cost and the time to write (the data to all mirrors) as performance objective.

To the best of our knowledge no other exemplar exists, yet, which can be used to evaluate latency-aware self-adaptation approaches.

The remainder of this paper is structured as follows. In the next section we summarise remote data mirroring and the previous exemplar, and will introduce our new exemplar. In section 3, we show how the exemplar provides metadata of adaptation actions to the user, i.e., how each adaptation action effects the three objectives and how to derive their latency. Then we show an exemplary optimiser to showcase how to make use of this metadata and conclude.

2 REMOTE DATA MIRRORING SIMULATOR

In the following, we introduce remote data mirroring, the previous simulation framework and our new simulation framework.

2.1 Remote Data Mirroring

Data plays a very important role in our current world. Its availability is, thus, equally important. A particular approach to increase the availability of data is to replicate it, i.e., the same data is hosted on multiple distributed machines. Or, in other words, the data is mirrored on remote servers. This approach is known under the term remote data mirroring [5, 6]. In this context, servers are called mirrors. Mirrors are connected via links. Once the link between two mirrors is established and the receiving mirror is ready, the source mirror can send the data to the target mirror.

A multitude of strategies to distribute the data amongst a set of servers exist. For example, different topologies of the mirrors lead to different characteristics of the overall system. If the mirrors form a fully connected graph, the data can be distributed within the system very fast, leading to a very high used bandwidth for the overall system. In contrast, if the mirrors form a balanced tree where each mirror has two child mirrors, it will take longer to distribute the data amongst all mirrors, but also require less bandwidth. The time required to distribute the data can be seen as a desired quality to the user, while the used bandwidth can be considered as cost. Current providers charge either directly by the bandwidth used or offer usage packages with defined upper limits of available bandwidth.

Thus, depending on the topology used and the parameters of the respective topologies (e.g., number of children), cost and quality of the system change. As the user's demand for a certain ratio between quality and cost might change, too, it is natural to add self-adaptation capabilities to such a system.

The adaptation manager has multiple configuration options: choosing a topology and its parameters as well as changing the number of mirrors. The manager monitors the used bandwidth, the number of active links, and the time required to distribute the data among all mirrors. The bandwidth metric represents the operational cost, the number of active links the reliability, and the time to write metric the performance of the network. Based on the user's current preferences, the analyser can determine whether the current configuration satisfies them, and, if not, the planner can derive the actions required to reconfigure the system so it does fulfil the user's demands.

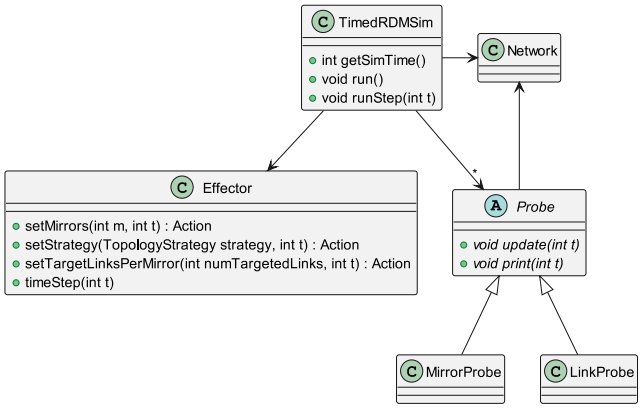

Figure 1: Top-level Architecture of the Timed RDM Simulator

2.2 The Previous Simulation Framework

To enable the investigation of how well self-adaptation approaches can optimise a remote data mirror system, Samin et al. [14] developed a simulator framework which offered probes and effectors to the developers of self-adaptation logic. Probes allow to inspect the current state of the system. Effectors allow to perform reconfiguration actions to the running system.

The system is simulated in time steps where for each step a set of metrics of the systems is computed: the number of active links, the currently used bandwidth and the time required to write data to mirrors. Notably, these metrics are computed based on predefined functions taking an abstract system state as input. That is, in this simulator, the mirrors, links and data packages are not represented as actual runtime objects. Instead, the system state is represented as an abstract description of them.

But, this abstract description of the system state does not allow to perform custom investigations of the simulated system. In particular, it does not allow to inspect how long it takes for an adaptation action to be actually fully realised in the simulator. For example, if new mirrors are added to the system, these mirrors need some time to boot and to establish the links according to the currently used topology. Using the previous exemplar, it is impossible to measure the time required for this change to be realised.

In consequence, we developed a new simulator, which simulates the remote data mirroring network as an actual object network.

2.3 The Latency-aware Simulation Framework

The architecture of our framework is depicted in Figure 1. A user of the framework creates an instance of the `Simulator` and retrieves `Probes` and an `Effector`. The probes provide insights into the current state of the framework. The effector allows to add adaptation actions at defined time steps of the simulation. The user can then either run the complete simulation or perform a step-wise simulation to perform additional actions in between the time steps.

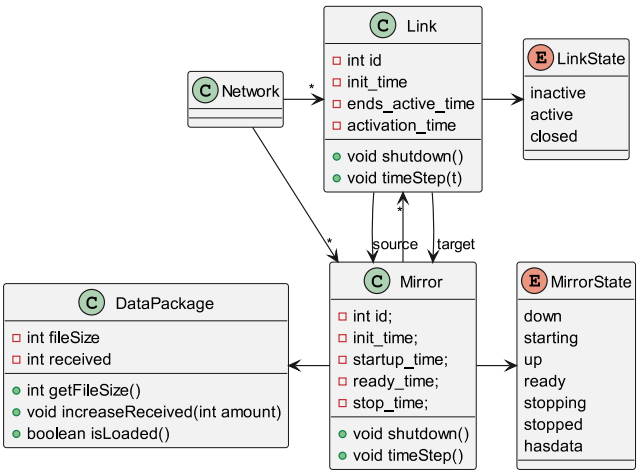

Figure 2: Domain Model of the Simulator

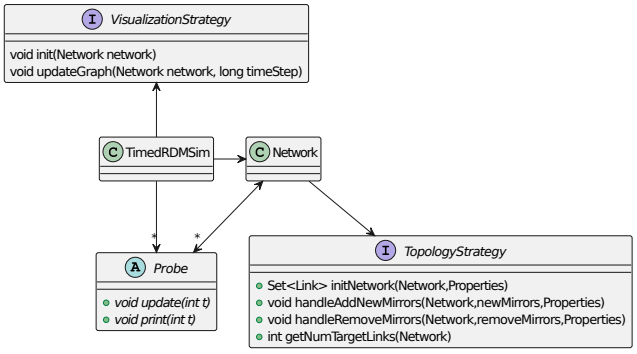

Figure 3: Variation and Extension Points of the Framework

2.3.1 The Domain Model of the Simulator. Internally, the simulator uses an object network of mirrors, links and data packages as depicted in Figure 2. Mirrors hold a list of all links they are part of, while each link refers to exactly one source mirror and one target mirror. Mirrors additionally reference their data package.

A mirror has multiple states. Initially, the mirror is in `down` mode and immediately switches to `starting` (i.e., booting). When it is up it starts to activate all its links. When the first link becomes active, the mirror becomes ready to receive data which it does until it received the complete data package. Finally, it is in state `hasdata`. When the mirror is shut down, the mirror switches to `stopping` mode and, once the shutdown is completed, to `stopped` mode. The switch between the states is timed. It takes `starting_time` time steps for a mirror to switch from `down` to `starting` mode and `ready_time` time steps to switch to `up` mode. The time of the shutdown sequence is determined by `stop_time`. Each of these times is randomly generated within boundaries specified in a configuration file by the user of the simulator. The time required until the mirror becomes `ready` depends on the time required for its links to get `active`. The time required until a mirror reaches the `hasdata` mode is to be observed during simulation as in each time step a certain fraction of the data can be received if the mirror is `ready` and has an active link to another mirror which is in `hasdata` mode.

Links can be either `inactive`, `active` or `closed`. Initially, they are `inactive` and take `activation_time` time steps until they are in `active` mode. But, links need to be activated by mirrors in `up` mode (or later modes). Thus, for a link to become `active` both source and target mirror first need to reach at least `up` mode and then the `activation_time` starts. Links are closed immediately.

Both links and mirrors have a unique id and an `init_time`, which is the simulation time at which the link or mirror was added to the network. Links in `closed` mode and mirrors in `stopped` mode are removed from the network in the next time step of the simulation. Data packages don't have an `id`, but a (configurable) `fileSize` and an attribute to store how much data has already been received.

To simulate more realistic scenarios, the simulator offers another parameter: `fault_probability`. A fault probability of 1% leads to crashes of mirrors and their associated links with a 1% chance for each mirror in each timestep.

2.3.2 Variability Points and Visualisation of the Framework. The framework offers interfaces for extensions and variations. In particular, the visualisation of the simulator can be varied, additional topologies can be added and the framework can be extended with further probe types. For visualisations and topologies, the framework uses the Strategy design pattern. For probes the Observer design pattern is used. Figure 3 depicts these three variation points.

To add a new visualisation strategy, the developer has to implement the `init` method, which is called to create the initial visualisation at the beginning of the simulation, and the `updateGraph` method, which is called after each time step.

To add a new topology strategy, the developer has to implement four methods. One used to initialise a new network, one to handle newly added mirrors, one to handle removed mirrors and a method returning the number of target links for the network using this topology. For example, the fully connected strategy returns the number of links of a fully connected graph (i.e., $(m \cdot (m-1))/2$ where m is the number of mirrors). The framework offers three predefined topologies: fully connected (FC), n-connected (NC) and balanced tree (BT). In FC every mirror is linked with every other mirror. In NC every mirror has n outgoing links to other mirrors. In BT the mirrors are arranged as a B-Tree [4].

Finally, to add a new probe type, the developer needs to implement two methods. The first, `update`, is meant to collect data from the network. The second, `print`, is meant to print this information.

As standard visualisation, the framework offers a colour-coded graph representation using Graphstream[2]. As shown in Figure 4, the current network is visualised as a graph at the top and three line charts below it whereof the first shows the bandwidth, the second the relative active links and the last the time to write metric.

In the graph representation mirrors are depicted yellow, if they are starting or up, red, if they are stopping or stopped, green, if they are ready and purple, if they are in `hasdata` state. Links are depicted yellow when inactive, green when active and red when closed.

[2]https://graphstream-project.org/

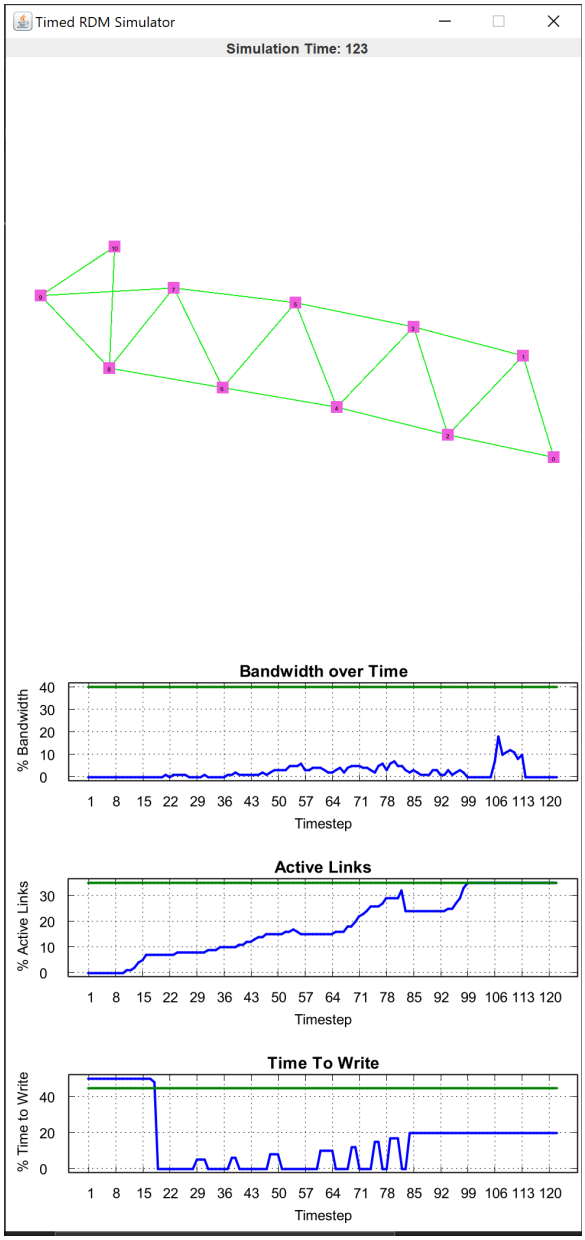

Figure 4: Graphical visualisation of the Simulation

3 LATENCY-AWARE SELF-ADAPTATION

In the following, we first explain how to derive the metadata for adaptation actions, i.e., how to assess their effect on the three metrics as well as on latency. We also show how this information is made available to the user of the simulation framework, i.e., a developer of a latency-aware self-adaptation approach. Next, we illustrate a simple optimiser making use of this knowledge. In this section, we show initial results on how to quantify the effects on the relative active links metric as well as on the associated latency and provide general insights on the bandwidth and time to write metric.

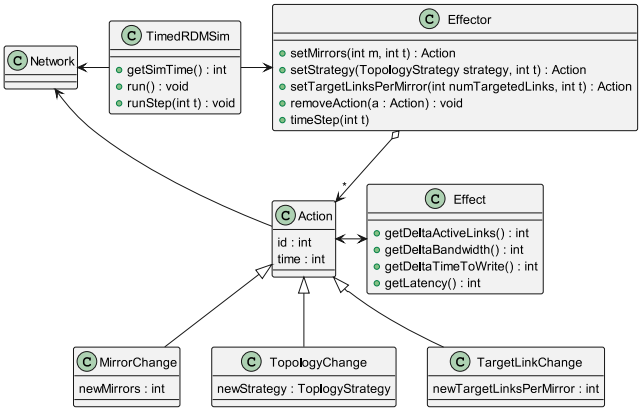

Figure 5: Latency-aware Adaptation Actions as Exposed by the Framework

ΔAL	FC	NC	BT
$m_1 \rightarrow m_2$	0	$2\frac{lpm(m_2-m_1)}{(m_1-1)(m_2-1)}$	$2\frac{(m_2-m_1)}{m_1 m_2}$
$lpm_1 \rightarrow lpm_2$	0	$2\frac{lpm_1-lpm_2}{m-1}$	0
Switch to FC	0	$\frac{2lpm}{m-1}$	$1-\frac{2}{m}$
Switch to NC	$\frac{2lpm}{m-1}-1$	-	$2\frac{m(1-lpm)-1}{m^2-m}$
Switch to BT	$\frac{2}{m}-1$	$2\frac{m(lpm-1)+1}{m^2-m}$	0

Table 1: Effect of Adaptations on Relative Active Links

3.1 Metadata of Adaptations

As mentioned in the last section, the simulator offers an `Effector` to enable users to queue adaptation actions to be performed at specific points in time of the simulation. These methods return an `Action` object referencing an `Effect` object. `Actions` have an `id` and a `time` and can be removed from the `Effector`, if their effect is unwanted. As each adaptation action has different metadata, there's a subclass for each type of action.

The `Effect` class summarises all 4 metrics of interest to the optimiser: the effect on each of the three objects and the latency, i.e., required timesteps.

3.1.1 Relative Active Links. The relative active links metric (AL) can be derived from the current network. For example, when the network currently has the Balanced Tree topology, the current number of mirrors can be used to compute the relative active links metric. The reference or maximum for each of the metrics is the fully connected topology. That is $AL = 1$ if there are $\frac{m*(m-1)}{2}$ links (fully connected graph). The maximum number of links (TL) for the balanced tree topology is $TL^{BT}(m) = m - 1$ (each mirror has one incoming link except for the root). We can, thus, compute $AL^{BT}(m) = (m-1)/\frac{m*(m-1)}{2} = \frac{2}{m}$. Moreover, we can compute the change on the relative active links from m_1 mirrors to m_2 mirrors as $\Delta AL^{BT}(m_1, m_2) = AL^{BT}(m_1) - AL^{BT}(m_2) = \frac{2(m_2-m_1)}{m_1 m_2}$.

For the n-connected topology with lpm links per mirror the maximum number of links $TL^{NC}(m, lpm)$ is defined as follows:

$$TL^{NC}(m, lpm) = \begin{cases} m \cdot lpm & m > 2 \cdot lpm \\ \frac{m \cdot (m-1)}{2} & else \end{cases} \quad (1)$$

That is if there are enough mirrors ($m > 2 \cdot lpm$) each mirror will have lpm outgoing links. Else we have a fully connected graph.

The relative active links can be computed as follows:

$$AL^{NC}(m, lpm) = \begin{cases} \frac{2 \cdot lpm}{m-1} & m > 2 \cdot lpm \\ 1 & else \end{cases} \quad (2)$$

The effect of changing the number of mirrors in the n-connected topology is computed as:

$$\Delta AL^{NC}(m_1, m_2, lpm) = \frac{2 \cdot lpm \cdot (m_2 - m_1)}{(m_1 - 1)(m_2 - 1)} \quad (3)$$

For the fully connected topology changes on the number of mirrors or links per mirror does not change the relative active links as it is itself the reference and, thus, $AL^{FC}(m, lpm) = 1$ for every number of mirrors and links per mirror.

3.1.2 Relative Bandwidth. The bandwidth metric is more complex to assess as it changes over time. Without any adaptation actions the bandwidth at a certain point in time is the result of the number of ready mirrors with an active link between them. If both mirrors at the end of each active link already have the data, the bandwidth for this link is 0. If one mirror does not have the data, the bandwidth can be derived from the simulator parameters: the (min/max) bandwidth per link (bpl) and the number of sending links (l_{send}). The number of sending links can be retrieved by traversing the current network. The current bandwidth of the network is then simply $TBW = bpl * l_{send}$. As a metric for optimisation the simulator offers the relative bandwidth metric (BW) as the ratio between the maximum possible bandwidth in a fully connected network where all links are sending and the current network's bandwidth:

$$BW(m, l_{send}) = \frac{bpl \cdot l_{send}}{bpl \cdot (\frac{m \cdot (m-1)}{2})} = 2\frac{l_{send}}{m \cdot (m-1)} \quad (4)$$

Thus, to effect this metric either the number of mirrors needs to be changed or the number of sending links. The number of sending links is a subset of the number of active links and changes over time even without any adaptation actions. As depicted in Figure 6, as soon as a mirror received the whole data package the link is not sending anymore.

The most effective way to increase the bandwidth of the net is, thus, to add new mirrors which don't have the data package. Increasing the number of links can increase the bandwidth, too, but only if there are enough mirrors left which still need to receive the data. Moreover, by just increasing the number of links the bandwidth logically will be increased for a shorter period compared to the scenario where more mirrors are added. To decrease the bandwidth, both reducing the number of links as well as reducing the number of mirrors is helpful. Comparing both options, the reduction of mirrors will lead to zero bandwidth faster than reducing the number of links.

	Δt
Increase Mirrors	$t_{startup} + t_{ready} + t_{activation}$
Decrease Mirrors	0
Change Links per Mirror	$t_{activation}$
Switch Topology	$t_{activation}$

Table 2: Latency of Adaptations

3.1.3 Relative Time To Write. The same complexity is present for the time to write metric. To derive it, first the time to write per package (TPP) is computed using the average bandwidth of the links and the filesize of the data package from the simulator parameters ($TPP = filesize/\varnothing bandwidth$).

The time to write the package to the whole network then depends on the topology. For the fully connected case, the time to write is the shortest, as the data package can be sent simultaneously from the first to all other mirrors, i.e., $TTW^{FC} = TPP$. For the balanced tree topology, the time to write can be computed via the depth of the tree. Using the worst-case depth for B-Trees [4], we can compute $TTW^{BT}(m, lpm) = TPP \cdot \lfloor log_{lpm} \frac{m+1}{2} \rfloor$. For the n-connected topology, we can examine the two extremes. Either we have $lpm = 1$ or we have $lpm > 2m$. The second case leads to the fully connected topology and thus has an equal time to write. The first case represents a chain as each mirror has exactly one outgoing link. The last mirror will be connected to the first mirror as the restriction is only on outgoing links (not on incoming links). Notably, for data transmission, the direction of the links is not considered, i.e., links can transfer data in both directions. Thus, the number of hops for the data to perform in parallel is half of the number of links (which equals the number of mirrors). We can compute $TTW^{1C}(m) = TTP \cdot \frac{m}{2}$. For $1 < lpm < 2m$, the TTW is between these two extremes. For the relative time to write, we consequently use TTW^{FC} as best case (=1) and TTW^{1C} (=0) as worst case and scale the current time to write within these limits.

Again, depending on the current topology, changing the number of mirrors and changing the number of links per mirror has a certain effect on the metric. For brevity, we refer the interested reader to the artefact for the concrete specifications.

3.1.4 Latency. The latency to change the number of mirrors is bounded by the simulator parameters. Removing links and metrics happens immediately. Thus, the latency of removing mirrors on all three metrics is 0.

The latency of adding mirrors on the metrics is more complex. The actual questions are: if n mirrors are added, how long does it take to establish all links and how long does it take to distribute the data package to them? As outlined in section 2.3.1, mirrors move through different states until the links are established and the data is distributed. To determine the latency on active links, bandwidth and time to write, we need to inspect the protocol more closely. Figure 6 depicts an activity diagram showing the state changes of two mirrors and the link between them.

To assess the latency of adaptation actions, we need to know when the links become active. This time can be derived from the simulation parameters as follows. Each mirror requires `startup_time` (min/max) timesteps to boot and `ready_time` (min/max) timesteps

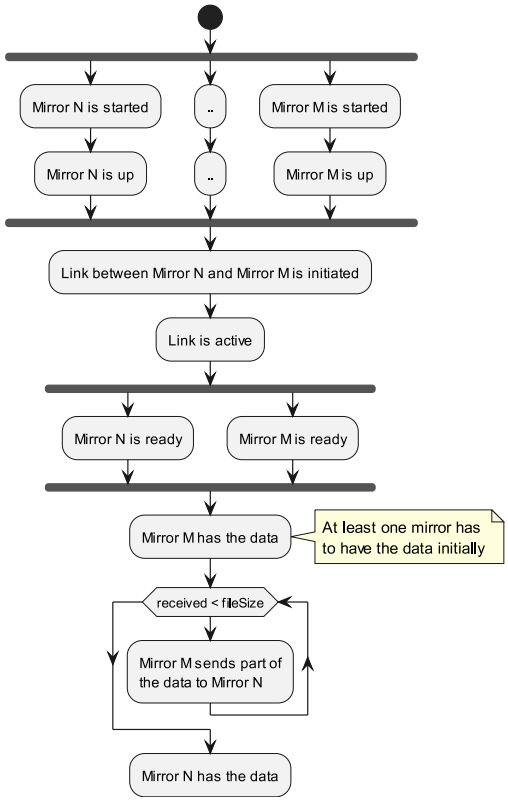

Figure 6: States of Mirrors and Links

to be ready to initiate a connection to other mirrors. Each link requires `link_activation_time` (min/max) timesteps to become active. Thus, the latency on relative active links is defined as:

$$\Delta t = t_{startup} + t_{ready} + t_{activation} \tag{5}$$

This way we can derive a minimum, maximum and average latency. Changing the number of links per mirror as well as changing the topology leads to a re-initialisation of all links according to the new topology or links per mirror. The mirrors are assumed to be ready. In consequence, the latency of such actions is $\Delta t = t_{activation}$.

More complex specifications considering the time required to transfer data are possible, but left for future work. The purpose of this exemplar is to enable researchers to investigate it.

3.2 An Exemplary Latency-aware Optimiser

In the following we outline how to build a latency-aware optimiser on top of our framework. Our intent is not to introduce a novel optimisation technique, but to show how our framework enables their development and investigation.

Thus, we present a simple optimiser with hard-coded rules. For clarity, the optimiser only aims to enforce the objective to keep the active links metric above 35%. For the N-connected topology, we know that we can increase this metric either by removing mirrors from the network or by increasing the number of links per mirror. Thus, at each timestep of the simulation we check if the active links metric fell below 35%. Additionally, we wait until at least 75% of all links got active, so the active links metric gets a chance to get close to its final value. If then the metric is still below 35%, two adaptation actions are created. One to remove a mirror and another to increase the links per mirror by 1. We then use our framework to derive the latency for these actions and choose the fastest one. The resulting simulation is depicted in Figure 4 and can be found in the artefact as `ExampleOptimizer.java`.

Listing 1: Example Optimiser Using Latency Information

```
LinkProbe lp = ...;
for(int t = 1; t < sim.getSimTime(); t++) {
  sim.runStep(t);
  if(lp.getLinkRatio() > 0.75) {
    if (lp.getActiveLinkMetric(t) < 35) {
      mirrors--;
      lpm++;
      Action removeMirror =
        sim.getEffector().setMirrors(mirrors, t + 1);
      Action increaseLPM =
        sim.getEffector().setLPM(lpm, t + 1);
      if(removeMirror.getEffect().getLatency()
       > increaseLPM.getEffect().getLatency()) {
        sim.getEffector().removeAction(removeMirror);
        mirror++;
       } else {
        sim.getEffector().removeAction(increaseLPM);
        lpm--;
} } } }
```

To evaluate novel latency-aware self-adaptation approaches a baseline is required. We suggest this baseline to be the case where all adaptations have the same latency. To show the benefit of taking adaptation latency into account the mean squared differences for a selected objective (e.g., reliability) can be used. For the example above, this would be the mean of differences for each timestep of the simulation between the actual relative active links and the desired 35%. More sophisticated measures are left for future work.

4 CONCLUSION

In this paper we have introduced a novel exemplar to investigate latency-aware self-adaptation in the context of remote data mirroring. The example is based on an existing exemplar and extends it by providing metadata on the offered adaptation actions. The system can be adapted by changing the number of mirrors, changing the number of links per mirror and changing the topology. The simulator reveals the effect of these adaptations on three metrics: the used bandwidth representing the cost, the time to write representing performance and the number of active links representing reliability.

Whilst the original exemplar generated random numbers for the three metrics between configurable boundaries, this exemplar performs an actual simulation of the mirrors, links and data packages constituting the network. This enables users of the simulator to investigate the effects and latency of adaptation actions in this context much more realistically.

REFERENCES

[1] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@ run. time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling* 18 (2019), 3049–3082.

[2] Javier Cámara, Gabriel A. Moreno, and David Garlan. 2014. Stochastic Game Analysis and Latency Awareness for Proactive Self-Adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Hyderabad, India) *(SEAMS 2014)*. Association for Computing Machinery, New York, NY, USA, 155–164. https://doi.org/10.1145/2593929.2593933

[3] Javier Cámara, Gabriel A. Moreno, David Garlan, and Bradley Schmerl. 2016. Analyzing Latency-Aware Self-Adaptation Using Stochastic Games and Simulations. *ACM Trans. Auton. Adapt. Syst.* 10, 4, Article 23 (jan 2016), 28 pages. https://doi.org/10.1145/2774222

[4] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (jun 1979), 121–137. https://doi.org/10.1145/356770.356776

[5] Minwen Ji, Alistair C Veitch, John Wilkes, et al. 2003. Seneca: remote mirroring done write.. In *USENIX Annual Technical Conference, General Track*. 253–268.

[6] Kimberly Keeton, Cipriano A Santos, Dirk Beyer, Jeffrey S Chase, John Wilkes, et al. 2004. Designing for Disasters.. In *FAST*, Vol. 4. 59–62.

[7] Claas Keller and Zoltán Ádám Mann. 2020. Towards Understanding Adaptation Latency in Self-adaptive Systems. In *Service-Oriented Computing – ICSOC 2019 Workshops*, Sami Yangui, Athman Bouguettaya, Xiao Xue, Noura Faci, Walid Gaaloul, Qi Yu, Zhangbing Zhou, Nathalie Hernandez, and Elisa Y. Nakagawa (Eds.). Springer International Publishing, Cham, 42–53.

[8] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[9] Pattie Maes. 1987. Concepts and experiments in computational reflection. *ACM Sigplan Notices* 22, 12 (1987), 147–155.

[10] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[11] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2786805.2786853

[12] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2018. Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation. *ACM Trans. Auton. Adapt. Syst.* 13, 1, Article 3 (apr 2018), 36 pages. https://doi.org/10.1145/3149180

[13] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)* 4, 2 (2009), 1–42.

[14] Huma Samin, Luis H Garcia Paucar, Nelly Bencomo, Cesar M Carranza Hurtado, and Erik M Fredericks. 2021. RDMSim: an exemplar for evaluation and comparison of decision-making techniques for self-adaptation. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 238–244.