Lock-based or Lock-less: Which Is Fresh?

Vishakha Ramani, Jiachen Chen, Roy D. Yates
WINLAB, Rutgers University
Email: {vishakha, jiachen, ryates}@winlab.rutgers.edu

***Abstract*—We examine status updating systems in which time-stamped status updates are stored/written in shared-memory. Specifically, we compare Read-Copy-Update (RCU) and Readers-Writer lock (RWL) as shared-memory synchronization primitives on the update freshness. To demonstrate the tension between readers and writers accessing shared-memory, we consider a network scenario with a pair of coupled updating processes. Location updates of a mobile terminal are written to a shared-memory Forwarder Information Base (FIB) at a network forwarder. An application server sends "app updates" to the mobile terminal via the forwarder. Arriving app updates at forwarder are addressed (by reading the FIB) and forwarded to the mobile terminal. If a FIB read returns an outdated address, the misaddressed app update is lost in transit. We redesign these reader and writer processes using preemption mechanisms that improve the timeliness of updates. We present a Stochastic Hybrid System (SHS) framework to analyze location and app update age processes and show how these two age processes are coupled through synchronization primitives. Our analysis shows that using a lock-based primitive (RWL) can serve fresher app updates to the mobile terminal at higher location update rates while lock-less (RCU) mechanism favors timely delivery of app updates at lower location update rates.**

***Index Terms*—Network performance analysis, Read Copy Update, Reader Writer Locks, Age-of-Information**

I. Introduction

In a broad range of cyber-physical systems, a source generates *status updates*, time-stamped measurements of a random process of interest, that are sent to one or more destinations through a network. In addition to requiring low latency transmission of each update, these applications require timeliness in the status update process at these destinations. For example, in robotic telesurgery, a surgeon needs timely feedback on the robotic arm since stale feedback can lead to organ injuries and other ramifications [1], [2].

It therefore becomes imperative to scrutinize current network architectures in terms of their feasibility to satisfy stringent real-time and freshness requirements. Delays of tens of nanoseconds (e.g., a sequential SSD write of 8 kB takes 1ms, a sequential SSD read of 8kB takes 1μs [3]) can significantly degrade system performance, particularly when they accumulate across multihop network paths. Hence, understanding the behaviour of a single network node is crucial in the development of ultra-low latency networks that deliver timely information.

On the other hand, advances in high performance computing have led to realizations of solutions to a range of problems in engineering, medical sciences, space research etc. that typically involve performing computationally intensive tasks in a short amount of time. This is achieved by a mix of technologies, including shared memory multiprocessor systems (for parallel computing), and vector processing along with various algorithms built on such architectures. A plethora of applications benefit from parallelization of various operations, including, for example, high throughput transaction processing in distributed databases [4] and faster training employing embarrassingly parallel processes in machine learning [5].

Such applications with high inter-processor communication demands expose synchronization between multiple processors as a key bottleneck in parallel computation in terms of scalability and computation times. In particular, a critical success factor in shared memory multiprocessors is synchronization, namely the coordination of concurrent tasks to ensure data consistency and correctness. This issue concerning readers-writer concurrency is manifested in various places. For instance, in a distributed database system, the challenge is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another [6]. In parallel machine learning, wherein there is an equal partitioning of data points across available processors, each having access to some *global state* (for e.g. model parameters), then an incorrect modification of global state could potentially conflict with operations on other processors [7].

For shared memory systems, primitives that enforce synchronization among reader threads and writer threads when accessing a shared resource are a way to avoid race conditions caused by concurrent manipulation by different writer threads. Specifically, these primitives allow multiple threads (readers and writers) to execute concurrently and ensure that results of reading and writing are predictable.

The existing literature on synchronization techniques focuses mostly on the algorithm, implementation and throughput performance (operations per unit time) in the critical sections[1] [8]–[10]. However, there has been a lack of study on the impact of synchronization primitives on the timeliness of the data accessed from shared memory in real-time IoT systems [11], [12]. By contrast, there have been many studies addressing the timeliness of status updates in queues and networks using *Age of Information* (AoI) freshness metrics; see the surveys [13], [14] and references therein.

Age of information is an end-to-end performance metric for the timeliness of a status updating process. An update with time-stamp u is said to have *age* $t - u$ at a time $t \geq u$

[1]Formally, a critical section is a protected section of the shared resource that is protected against multiple concurrent accesses

and the freshness of an update is decreasing with its age. A monitor receiving a stream of updates has an age process $\Delta(t) = t - u(t)$ when $u(t)$ is the time-stamp of the freshest received update. The recent AoI literature has focused on high-level systems such as sensor networks, wireless multiple access channels, and networks of queues in which delays are typically measured in milliseconds.

In this study, we employ AoI freshness metrics to compare the performance of status updating systems employing Read-Copy-Update (RCU) and Readers-Writer lock (RWL) synchronization primitives to disseminate information using shared memory. While sub-microsecond delays may be typical in shared memory reads and writes, the frequency of such operations contribute to latency in network switches and routers. Furthermore, due to advances in both link and radio access network technologies, the transmission time of packets is often negligible. We contend that in many applications the bottleneck in status updating systems has shifted from data communication over links to data storage, processing and retrieval in network nodes. It is therefore imperative to analyse how network devices affect the freshness of updates. With this motivation, we analyze two fundamental synchronization primitives — Read-Copy-Update (RCU) and Readers-Writer Lock (RWL) — to aid both network engineers and software developers in designing architectures and software libraries for applications requiring timely status updates.

II. System Model

In this work, we focus on a class of system in which a source generates time-stamped *updates* (i.e., measurements of a random process of interest) and a concurrent/shared data structure stores these updates[2]. While the system will have many sources, our focus will be on the shared data structure that tracks the status of a single process of interest. Going forward, we refer to the shared data structure as *shared memory*, or simply as *memory*. Hence, a writer is responsible for recording the fresh source/sensor measurements as updates in the memory. A reader fulfills clients' requests for these measurements by reading from the memory.

A. Timely Update Forwarding

To demonstrate the effect of concurrency constructs on timeliness, we consider an example of packet forwarding in a mobile user environment, as shown in Fig. 1. An application server in the network is sending "app updates" regarding a process of interest to a mobile terminal. The application sends its update packets to a forwarding node in the network. This forwarder maintains a Forwarder Information Base (FIB) that tracks the location (i.e. point of attachment network address) of the mobile terminal. At the forwarder, app updates are addressed using the FIB and forwarded to the mobile terminal.

This system has two update processes that we will track. First, we will track the age $\Delta(t)$ of the app update process at the mobile. Second, the mobile terminal sends "location updates" to the forwarder that get written in the FIB. For this process, we wish to track the age $\hat{\Delta}(t)$ of location updates in the FIB.

[2]The concurrent data structures usually reside in shared memory that is an abstract storage environment.

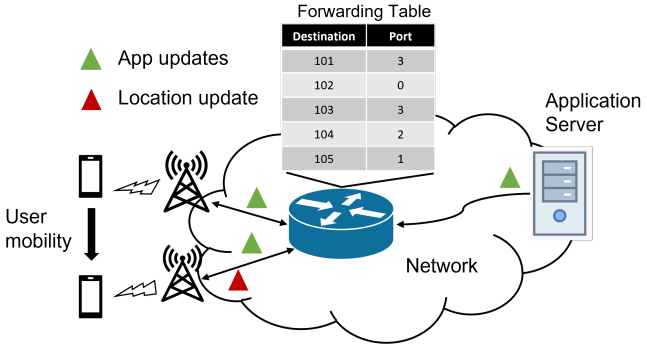


Fig. 1: Packet forwarding application with mobile users

These two age processes are coupled through the FIB. At the forwarder, a writer receives location updates from the mobile and writes them to the FIB while a reader receives app updates from the application server and needs to read the FIB in order to address the app updates for forwarding to the mobile terminal. In short, *location updates are written to the FIB and the app updates are client requests to read the FIB*.

If the mobile terminal has moved and the FIB holds the wrong address, the misaddressed app updates are assumed to be lost and such packet losses will be reflected in increased age in the app updates at the mobile terminal. Misaddressed packets can arise in RCU if the reader reads the FIB while the write of a fresh location update is in progress. Misaddressed packets occur in RWL when a read lock prevents the writer from writing a fresh location update.

The app updates in a router's queue are read/service requests to readers. The key idea here is that when the read returns with an address, the queue entity should keep the most recent/freshest read request and forwards that update using the value of returned read. This results in the mobile client receiving the freshest app update. To highlight, the freshness of app updates received at mobile client depends upon two factors: first, if the update was addressed correctly, and second, only the freshest app updates are served at the router. Notice that both these factors are inherently affected by the synchronization mechanism being used.

B. Model Assumptions

There is no apparent consensus in the literature on modeling random variables associated with the time needed for the execution of a read or write operation. This is indeed the time needed for a software function call that depends upon the underlying hardware and operating system used. In this work, we will assume exponential distributions for both write and read times. If the time of an operation (read or write) has an exponential (μ) distribution, then the average time of the operation is $1/\mu$ and we refer to μ as the speed of the operation. While exponential models are decidedly too simple, they enable (with a manageable number of parameters) an

analytic characterization of update age performance induced by the complex RCU and RWL synchronization primitives.

We assume the source submits fresh location updates as rate $\hat{\lambda}$ Poisson process to the network and that these updates arrive fresh at the FIB writer, i.e. with age 0. The write operations to unlocked memory locations have independent exponential $(\hat{\mu})$ service times. We further assume a preemption in service model; if the writer is busy writing a location update and a new location update arrives then the update in service is preempted and writer starts serving this fresher update. Specifically, the writer discards the location update in service and starts writing the fresh update.

We similarly model the arrivals of client requests (app updates) to the FIB reader as a rate λ Poisson process, and assume each read request's service/read time is an independent exponential (μ) random variable. At the FIB reader, we allow the client requests to be preempted while waiting for the read to return. Specifically, after the read returns, only the fresher app update is addressed with the FIB read and any previous old update waiting for the read value is preempted.

We note that modeling and simulation settings are necessarily simplified to facilitate getting some insight into understanding the system. It is possible to extend our approach with more detailed models where a writer takes multiple stages to finish a write, where each stage takes an exponential time, then the total write time PDF is the convolution of these exponential PDFs. However, without the simplified exponential models, the age analysis is intractable, and the alternative is to simulate.

C. Paper Outline and Contributions

In section III, we provide an overview of the RCU and RWL synchronization primitives. In Section IV, we introduce the Stochastic Hybrid Systems (SHS) method for AoI evaluation. We use SHS to compare RCU and RWL in terms of the average age of location and app updates in the update forwarding system introduced in Section II-A. We show how the app update age process and location update age process are coupled through the FIB. In section V we perform numerical evaluations to understand and compare RCU and RWL and their effect on location and app update age process. This includes a comparison of preemptive and non-preemptive RCU and RWL models. While one may speculate that the lockless RCU approach that enables writing to the FIB without delay should outperform the delay-inducing locks of RWL, our results show that RCU is superior in some operating regimes but worse in others. While these conclusions are specific to our update forwarding example, the approach we develop here can guide the construction of similar comparisons for other distributed updating applications employing shared memory.

III. Overview of Synchronization Primitives

A. Read-Copy-Update (RCU)

While conventional locking techniques such as Readers-Writer Locks (RWL) enforce strict mutual exclusion between readers and writers in order to prevent destructive modifications, they fall short on concurrent computations by virtue of mutual exclusion. Replacing expensive conventional locking techniques, Read-Copy-Update (RCU) is a synchronization primitive that allows concurrent forward progress for both writers and readers [15]. RCU can be broadly described in two steps [16]: 1) To publish a newer version of a data item[3], the writer creates a copy of the RCU protected data item, modifies this copy with the newer version of this data item, and atomically replaces the old reference with a reference to this newer version. This publishing process runs concurrently with ongoing read processes that continue to read the old copy/version using the old reference. However, new read requests read the most recent version. 2) Since some readers in progress hold reference to "stale" data, the system defers memory reclamation of old data until after each reader in progress has finished executing its read-side critical section. Therefore, at any point of time, RCU can maintain multiple time-stamped versions of a data item that are concurrently read by readers in the system.

Every reader that enters a read-side critical section prior to any modification from the writer is able to finish executing its respective critical section. A "grace period" starts at the moment the writer publishes the modified data item and enables all RCU read-side critical sections in existence at the beginning of a given grace period to complete [17]. Thus, the end of grace period ensures that it is safe to reclaim the memory and delete the stale copy.

B. Readers-Writer Lock (RWL)

RWL is a synchronization primitive that enforces mutual exclusion between readers and writers; multiple readers are allowed to read the shared data structure concurrently, while a writer requires exclusive access or a "lock"[4] to that data structure. The focus of most RWL implementations is that no thread should be allowed to starve. Therefore, with just one writer, RWL implementations are mostly *write preferring* [18]. In this writer priority RWL, once the writer starts waiting in a queue to acquire the lock, the RWL mechanism prevents new readers from acquiring the lock. The writer's acquisition of the lock occurs once all readers already holding the read lock have finished reading. During the write lock, new read lock requests are queued until the writer has released its lock.

C. RCU and RWL Background

RCU has been used in a multitude of places in both user-space and the Linux kernel. For example, in the networking protocol stack, LC Tries employs locking via RCU to enable efficient IP address lookups [19], [20]. User-space RCU [21] is used in high-performance DNS servers [22], in the Linux networking toolkit [23], in distributed object storage systems [24]. Most recently, RCU protected data structures have been employed to ensure wait-free access to machine learning models by inference threads [25]. One drawback of a classical RCU mechanism is the *wait-for-readers* (using

[3]A writer has *published* an update when it exits its write critical section

[4]In shared-memory multiprocessor architectures, a lock is a mechanism that restricts the access to a shared data structure among multiple processors

synchronize_rcu()) primitive where updaters wait for all pre-existing readers to complete their read-side critical sections. Various RCU variants have been proposed (Predicate RCU [26], [27], read-log update [28]) that address the wait-for-readers problem. Apart from this, [29] introduced a real-time variant of RCU that allows preemption of read-side critical sections.

A caveat of RCU is that it doesn't support multiple concurrent updates. A body of research focuses and design of algorithms that support concurrent updates and multi-versioning [28], [30], [31]. Further, RCU implementation and verification is non-trivial and several attempts have been made to systematically check the RCU design and code [32]–[35].

Readers-Writer locks are ubiquitous in today's system and are found to support concurrency in virtual file systems, large key-value stores, database systems, software transactional memory implementations [36]. Conventional implementation of Readers-Writer lock suffers from reader-reader scalability and different designs have been proposed for scalable Readers-Writer locks [37], [38]. Authors in [39] present the design of a family of RW locks to leverage NUMA features and deliver better performance.

IV. AoI Evaluation of app updates using SHS

A. Stochastic Hybrid Systems (SHS) Overview

To evaluate AoI of app updates, we use a Stochastic Hybrid Systems (SHS) [40] approach, a technique introduced for AoI evaluation in [41] and since employed in AoI evaluation of a variety of status updating systems [42]–[49]. A stochastic hybrid system has a state-space with two components – a discrete component $q(t) \in \mathcal{Q} = \{0, 2, \ldots, M\}$ that is a continuous-time finite state Markov Chain and a continuous component $\mathbf{x}(t) = [x_0(t), \ldots, x_n(t)] \in \mathbb{R}^{n+1}$. In AoI analyses using SHS, each $x_j(t) \in \mathbf{x}(t)$ describes an age process of interest. Each transition $l \in \mathcal{L}$ is a directed edge (q_l, q_l') with a transition rate $\lambda^{(l)}$ in the Markov chain. The age process vector evolves at a unit rate in each discrete state $q \in \mathcal{Q}$, i.e., $\frac{d\mathbf{x}}{dt} = \dot{\mathbf{x}}(t) = \mathbf{1}_n$. A transition l causes a system to jump from discrete state q_l to q_l' and resets the continuous state from $\mathbf{x}$ to $\mathbf{x}'$ using a linear transition reset map $\mathbf{A}_l \in \{0,1\}^{(n\times n)}$ such that $\mathbf{x}' = \mathbf{x}\mathbf{A}_l$. For simple queues, examples of transition reset mappings $\{\mathbf{A}_l\}$ can be found in [41].

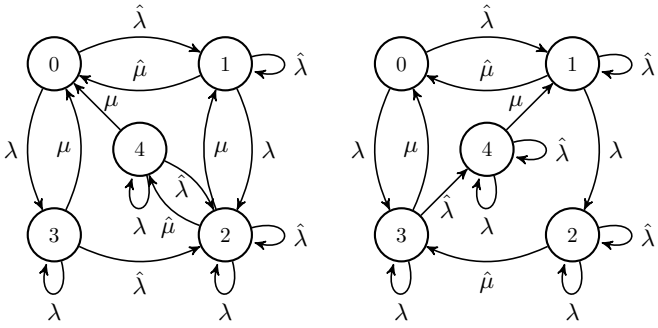
For a discrete state $\bar{q} \in \mathcal{Q}$, let $\mathcal{L}_{\bar{q}}$ and $\mathcal{L}'_{\bar{q}}$ be sets of incoming and outgoing transitions, i.e.

$$\mathcal{L}_{\bar{q}} = \{l \in \mathcal{L} : q_l' = \bar{q}\}, \qquad \mathcal{L}'_{\bar{q}} = \{l \in \mathcal{L} : q_l = \bar{q}\}. \tag{1}$$

Age analysis using SHS is based on the expected value processes $\{\mathbf{v}_q(t) \colon q \in \mathcal{Q}\}$ such that

$$\begin{aligned}\mathbf{v}_q(t) &= \mathrm{E}[\mathbf{x}(t)\delta_{q,q(t)}] \\ &= [\mathrm{E}[x_1(t)\delta_{q,q(t)}] \ \cdots \ \mathrm{E}[x_n(t)\delta_{q,q(t)}]],\end{aligned} \tag{2}$$

with $\delta_{i,j}$ denoting the Kronecker delta function. For the SHS models of age processes considered here, each $\mathbf{v}_q(t)$ will converge to a fixed point $\bar{\mathbf{v}}_q$. The fixed points $\{\bar{\mathbf{v}}_q \colon q \in \mathcal{Q}\}$ are the solution to a set of age balance equations. Specifically,



(a) RCU **(b)** RWL

Fig. 2: SHS Markov chain for **(a)** RCU mechanism and for **(b)** RWL mechanism.

the following theorem provides a simple way to calculate the age balance fixed point and then the average age in an ergodic queueing system.

Theorem 1: [41, Theorem 4] If the discrete-state Markov chain $q(t) \in \mathcal{Q} = \{0, \ldots, M\}$ is ergodic with stationary distribution $\bar{\boldsymbol{\pi}} = [\bar{\pi}_0 \ \cdots \ \bar{\pi}_M] > 0$ and there exists a non-negative vector $\bar{\mathbf{v}} = [\bar{\mathbf{v}}_0 \ \cdots \ \bar{\mathbf{v}}_M]$ such that

$$\bar{\mathbf{v}}_{\bar{q}} \sum_{l\in\mathcal{L}_{\bar{q}}} \lambda^{(l)} = \mathbf{1}\bar{\pi}_{\bar{q}} + \sum_{l\in\mathcal{L}'_{\bar{q}}} \lambda^{(l)} \bar{\mathbf{v}}_{q_l} \mathbf{A}_l, \quad \bar{q} \in \mathcal{Q}, \tag{3}$$

then the average age vector is $\mathrm{E}[\mathbf{x}] = \lim_{t\to\infty} \mathrm{E}[\mathbf{x}(t)] = \sum_{\bar{q}\in\mathcal{Q}} \bar{\mathbf{v}}_{\bar{q}}$.

B. RCU and RWL: SHS framework

We now describe the SHS framework for RCU and RWL considering the packet forwarding example. We consider two age vectors: 1) the age state of the location update process is $\hat{\mathbf{x}}(t) = [\hat{x}_0(t) \ \hat{x}_1(t)]$, where $\hat{x}_0(t)$ is the age of the location update seen by the writer and $\hat{x}_1(t)$ is the age of the current location update in memory; 2) the age state of the application update process that initiates client read requests is $\mathbf{x}(t) = [x_0(t) \ x_1(t)]$, where $x_0(t)$ and $x_1(t)$ are the ages of the most recent application updates at the reader and at the mobile terminal (i.e. the destination monitor) respectively.

Since RWL and RCU are fundamentally two different mechanisms for accessing shared memory, the discrete states $\mathcal{Q} = \{0, 1, 2, 3, 4\}$ are similar albeit different:

State 0 The idle state

State 1 The writer is writing fresh update (with a write lock in RWL)

State 2 The writer is writing a fresh update (with a write lock in RWL) but the *reader* action is different for RCU and RWL. For RCU, the reader is reading a stale address, but in RWL, the reader has requested a read lock and is waiting for the lock to become active.

State 3 The reader reading fresh/correct update from memory (with a read lock in RWL)

State 4 The reader is reading a stale update (with read lock active in RWL) but the *writer* state is different. For RCU, the writer has finished writing the update and this new update is published. For RWL, the writer has requested

a write lock and is waiting for the in-progress read to finish.

The discrete-state Markov chains for RCU and RWL are shown in Fig. 2(a) and 2(b) respectively. The SHS transition reset maps for RCU and RWL are shown in Tables I(a) and I(b). In each table, a transition l from state q_l to q'_l occurs at rate $\lambda^{(l)}$ with age reset maps

$$\mathbf{x}' = \mathbf{x}\mathbf{A}_l, \qquad \hat{\mathbf{x}}' = \hat{\mathbf{x}}\hat{\mathbf{A}}_l. \tag{4}$$

Equation (4) highlights how the the app update process $\mathbf{x}(t)$ and location update process $\hat{\mathbf{x}}(t)$ are coupled only through the changes in the Markov chain at the forwarder. In each transition l, either the app update process $\mathbf{x}(t)$ changes or the location update process $\hat{\mathbf{x}}(t)$ changes, but not both. That is, either $\mathbf{A}_l$ or $\hat{\mathbf{A}}_l$ is an identity matrix for each transition l.

C. RCU and RWL: SHS Transitions

Here we describe the SHS transitions for both RCU and RWL that are enumerated in Tables I(a) and I(b). For each collection of transitions, we focus on the age state process ($\mathbf{x}(t)$ or $\hat{\mathbf{x}}(t)$) that changes. In particular, we first describe what is common to both RCU and RWL. This is followed by details specific to RCU and RWL respectively.

- $l = 1, \ldots, 5$: In each state $0, \ldots, 4$, the writer receives a fresh location update and initiates the write mechanism. Since the location update is fresh, $\hat{x}'_0 = 0$ whereas $\hat{x}'_1 = \hat{x}_1$ is unchanged as the location update has not yet been written to the FIB. In transitions $l = 2, 3$, the writer preempts an in-progress write with an updated location.
  - **RCU** Following transition $l = 5$, the in-progress read will now be returning an outdated address.
  - **RWL** In transition $l = 1$, the writer acquires a write-lock. In transitions $l = 2, 3$, the writer already holds the write-lock. In transitions $l = 4, 5$, the writer requests a write lock but the request is queued as the reader is in a critical section.
- $l = 6, 7$: The writer finishes writing to the FIB and publishes a new location update; $\hat{x}'_0 = \hat{x}_0$ is unchanged since no new location update arrives at the writer but $\hat{x}'_1 = \hat{x}_0$ as the age at the FIB is reset to the age of the just-written update. In transition $l = 6$, the system goes to the idle state.
  - **RCU** In transition $l = 7$, the system goes to state 4 because a grace period starts with a read in progress.
  - **RWL** For transition, $l = 7$, there is a pending read request and so the reader acquires the read lock and enters a read-side critical section.
- $l = 8, \ldots, 12$: In each discrete state $0, \ldots, 4$, an app update arrives, initiating a read request; $x'_0 = 0$ as the app update is fresh at the reader but $x'_1 = x_1$ since the app update has not yet been delivered to the mobile terminal.
  - **RCU** In transitions $l = 9, 10$, the system enters state 2 in which the writer is simultaneously writing a fresh location update. *Consequently, the reader will read a stale address from the FIB in state* 2. For transitions $l = 11, 12$, a read was already in-progress; when that read completes, the address returned by the FIB is used to address this most recent app update. Effectively, the arriving app update preempts the prior update that had been held by the reader.
  - **RWL** In transition $l = 8$, the reader immediately acquires a read-lock on the FIB and initiates the read. In transition $l = 9$, the app update arrives in a write-lock state, the reader requests a read lock on the FIB that is denied and the system transitions to state 2, the write-lock with read pending state. In transition $l = 10$, the fresh app update arrives in state 2 and the system stays in the write-lock with read-pending state. In transitions $l = 11, 12$, the fresh app update arrives in a state with the read-lock already active. Hence, in transitions $l = 10, 11, 12$, the system remains in its same state but the fresh app update preempts the prior app update at the reader that was waiting to be addressed and sent. We note that following transition $l = 10$ or $l = 11$, there is the chance that the app update will eventually be correctly delivered to the mobile. However, in the case of transition $l = 12$, the app update, if not preempted, will eventually read an outdated address and go misaddressed.
- $l = 13$: The reader retrieves a location address from the FIB and exits the critical section; $x'_0 = x_0$ but $x'_1 = x_0$ as the age at mobile terminal is reset to the age of the app update that was just addressed and delivered to the mobile. In both RCU and RWL, the system returns to the idle state.
- $l = 14$: The reader retrieves a stale location address from the FIB, exits the critical section and attempts to forward the app update to the mobile; $x'_0 = x_0$ but $x'_1 = x_1$ is unchanged, i.e. the age at mobile is not reset since the address read is outdated and the misaddressed app update is lost in transit.
  - **RCU** When this transition occurs, the writer is idle, having already finished writing its location update to the FIB. However, the reader is fetching the prior copy holding the outdated location.
  - **RWL** When this transition occurs, the writer has received the location update, but is in a write-pending state waiting for the read-lock to be released.
- $l = 15$: When this transition occurs, an RCU read finishes while an in-progress RCU write is updating the FIB with a location update that occurred while the read was in progress. (This transition is exclusive to RCU since RWL locking prohibits simultaneous reading and writing.) Similar to transition $l = 14$, the reader retrieves the stale (prior) location address and attempts to forward the app update to the mobile. Since this address is outdated, the misaddressed app update is lost in transit; $x'_0 = x_0$ and $x'_1 = x_1$ are unchanged.

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\hat{\mathbf{x}}\hat{\mathbf{A}}_l$	$\hat{\mathbf{A}}_l$	$\mathbf{x}\mathbf{A}_l$	$\mathbf{A}_l$
1	$0 \to 1$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
2	$1 \to 1$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
3	$2 \to 2$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
4	$3 \to 2$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
5	$4 \to 2$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
6	$1 \to 0$	$\hat{\mu}$	$[\hat{x}_0\ \hat{x}_0]$	$\left[\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
7	$2 \to 4$	$\hat{\mu}$	$[\hat{x}_0\ \hat{x}_0]$	$\left[\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
8	$0 \to 3$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
9	$1 \to 2$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
10	$2 \to 2$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
11	$3 \to 3$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
12	$4 \to 4$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
13	$3 \to 0$	μ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_0]$	$\left[\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right]$
14	$4 \to 0$	μ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
15	$2 \to 1$	μ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$

(a) RCU

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\hat{\mathbf{x}}\hat{\mathbf{A}}_l$	$\hat{\mathbf{A}}_l$	$\mathbf{x}\mathbf{A}_l$	$\mathbf{A}_l$
1	$0 \to 1$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
2	$1 \to 1$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
3	$2 \to 2$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
4	$3 \to 4$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
5	$4 \to 4$	$\hat{\lambda}$	$[0\ \hat{x}_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
6	$1 \to 0$	$\hat{\mu}$	$[\hat{x}_0\ \hat{x}_0]$	$\left[\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
7	$2 \to 3$	$\hat{\mu}$	$[\hat{x}_0\ \hat{x}_0]$	$\left[\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$
8	$0 \to 3$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
9	$1 \to 2$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
10	$2 \to 2$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
11	$3 \to 3$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
12	$4 \to 4$	λ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[0\ x_1]$	$\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$
13	$3 \to 0$	μ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_0]$	$\left[\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right]$
14	$4 \to 1$	μ	$[\hat{x}_0\ \hat{x}_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$	$[x_0\ x_1]$	$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$

(b) RWL

TABLE I: SHS transitions for tracking age in Markov chains of Fig. 2 for (a) RCU and (b) RWL.

D. RCU: SHS Age Analysis

For the SHS analysis, we employ the normalized rates

$$\hat{\rho} = \hat{\lambda}/\hat{\mu}, \quad \beta = \lambda/\hat{\mu}, \quad \sigma = \mu/\hat{\mu}. \tag{5}$$

We note that $\hat{\rho}$ is the offered load of location updates being written to the FIB. Similarly, β is the normalized arrival rate of FIB read requests. For RCU, the Fig. 2(a) Markov chain has stationary probabilities $\boldsymbol{\pi} = [\pi_0\ \pi_1\ \pi_2\ \pi_3\ \pi_4]$ with normalization constant C_π given by

$$\boldsymbol{\pi} = C_\pi^{-1}[\sigma\ \ \hat{\rho}\sigma\ \ \beta\hat{\rho}\ \ \tfrac{\beta\sigma}{\hat{\rho}+\sigma}\ \ \tfrac{\beta\hat{\rho}}{\hat{\rho}+\sigma}], \tag{6a}$$

$$C_\pi = (1+\hat{\rho})(\beta+\sigma). \tag{6b}$$

We now apply Theorem 1 to the SHS reset maps in Table I(a). With the shorthand notations

$$\lambda^* = \lambda + \hat{\lambda}, \qquad \mu^* = \mu + \hat{\mu}, \tag{7}$$

From Theorem 1, the RCU location update age process $\hat{\mathbf{x}}(t)$ has age balance fixed points $\hat{\mathbf{v}}_q = [\hat{v}_{q0}\ \hat{v}_{q1}]$ satisfying

$$\lambda^*\hat{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_6 + \mu\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{13} + \mu\hat{\mathbf{v}}_4\hat{\mathbf{A}}_{14}, \tag{8a}$$

$$(\lambda^*+\hat{\mu})\hat{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\hat{\mathbf{v}}_0\hat{\mathbf{A}}_1 + \hat{\lambda}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_2 + \mu\hat{\mathbf{v}}_2\hat{\mathbf{A}}_{15}, \tag{8b}$$

$$(\lambda^*+\mu^*)\hat{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_3 + \hat{\lambda}\hat{\mathbf{v}}_3\hat{\mathbf{A}}_4 + \hat{\lambda}\hat{\mathbf{v}}_4\hat{\mathbf{A}}_5 + \lambda\hat{\mathbf{v}}_1\hat{\mathbf{A}}_9 + \lambda\hat{\mathbf{v}}_2\hat{\mathbf{A}}_{10}, \tag{8c}$$

$$(\lambda^*+\mu)\hat{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \lambda\hat{\mathbf{v}}_0\hat{\mathbf{A}}_8 + \lambda\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{11}, \tag{8d}$$

$$(\lambda^*+\mu)\hat{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\mu}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_7 + \lambda\hat{\mathbf{v}}_4\hat{\mathbf{A}}_{12}. \tag{8e}$$

From Table I(a) we see that $\hat{\mathbf{A}}_8, \hat{\mathbf{A}}_9, \ldots, \hat{\mathbf{A}}_{15}$ are all identity matrices. It follows that (8) simplifies to

$$\lambda^*\hat{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_6 + \mu\hat{\mathbf{v}}_3 + \mu\hat{\mathbf{v}}_4, \tag{9a}$$

$$(\lambda^*+\hat{\mu})\hat{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\hat{\mathbf{v}}_0\hat{\mathbf{A}}_1 + \hat{\lambda}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_2 + \mu\hat{\mathbf{v}}_2, \tag{9b}$$

$$(\hat{\lambda}+\mu^*)\hat{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_3 + \hat{\lambda}\hat{\mathbf{v}}_3\hat{\mathbf{A}}_4 + \hat{\lambda}\hat{\mathbf{v}}_4\hat{\mathbf{A}}_5 + \lambda\hat{\mathbf{v}}_1, \tag{9c}$$

$$(\lambda^*+\mu)\hat{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \lambda\hat{\mathbf{v}}_0 + \lambda\hat{\mathbf{v}}_3, \tag{9d}$$

$$(\lambda^*+\mu)\hat{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\mu}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_7 + \lambda\hat{\mathbf{v}}_4. \tag{9e}$$

Because the RCU app updates employ the same discrete state Markov chain as the RCU location updates, the RCU age balance equations for the app updates are identical to (8) with $\hat{\mathbf{v}}_0, \ldots, \hat{\mathbf{v}}_4$ and $\hat{\mathbf{A}}_1, \ldots, \hat{\mathbf{A}}_{15}$ replaced by $\bar{\mathbf{v}}_0, \ldots, \bar{\mathbf{v}}_4$ and $\mathbf{A}_1, \ldots, \mathbf{A}_{15}$ respectively:

$$\lambda^*\bar{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\bar{\mathbf{v}}_1\mathbf{A}_6 + \mu\bar{\mathbf{v}}_3\mathbf{A}_{13} + \mu\bar{\mathbf{v}}_4\mathbf{A}_{14}, \tag{10a}$$

$$(\lambda^*+\hat{\mu})\bar{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\bar{\mathbf{v}}_0\mathbf{A}_1 + \hat{\lambda}\bar{\mathbf{v}}_1\mathbf{A}_2 + \mu\bar{\mathbf{v}}_2\mathbf{A}_{15}, \tag{10b}$$

$$(\lambda^*+\mu^*)\bar{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\bar{\mathbf{v}}_2\mathbf{A}_3 + \hat{\lambda}\bar{\mathbf{v}}_3\mathbf{A}_4 + \hat{\lambda}\bar{\mathbf{v}}_4\mathbf{A}_5 + \lambda\bar{\mathbf{v}}_1\mathbf{A}_9 + \lambda\bar{\mathbf{v}}_2\mathbf{A}_{10}, \tag{10c}$$

$$(\lambda^*+\mu)\bar{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \lambda\bar{\mathbf{v}}_0\mathbf{A}_8 + \lambda\bar{\mathbf{v}}_3\mathbf{A}_{11}, \tag{10d}$$

$$(\lambda^*+\mu)\bar{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\mu}\bar{\mathbf{v}}_2\mathbf{A}_7 + \lambda\bar{\mathbf{v}}_4\mathbf{A}_{12}. \tag{10e}$$

For these equations, we note from Table I(b) that $\mathbf{A}_1, \ldots, \mathbf{A}_7$ and $\mathbf{A}_{14}, \mathbf{A}_{15}$ are all identity matrices and this will lead to a set of simplified age balance equations, equivalent to (9) for the RCU location updates. Numerical evaluation of $\hat{\mathbf{v}}_0, \ldots, \hat{\mathbf{v}}_4$ and $\bar{\mathbf{v}}_0, \ldots, \bar{\mathbf{v}}_4$ using (9) and the simplified version of (10) respectively is straightforward. It follows from Theorem 1 that average age $\mathrm{E}[\hat{\Delta}]$ of a location update in the FIB and the average age $\mathrm{E}[\Delta]$ of an app update at the mobile terminal are

$$\mathrm{E}[\hat{\Delta}] = \sum_{q=0}^{4} \hat{v}_{q1}, \qquad \mathrm{E}[\Delta] = \sum_{q=0}^{4} \bar{v}_{q1}. \tag{11}$$

E. RWL: SHS Age Analysis

The RWL Markov chain in Fig. 2(b) has stationary probabilities $\boldsymbol{\pi}$ with normalization constant C_π given by

$$\boldsymbol{\pi} = [\pi_0\ \pi_1\ \pi_2\ \pi_3\ \pi_4] = C_\pi^{-1} \begin{bmatrix} \sigma(\hat{\rho}+\sigma+\beta\sigma) \\ \hat{\rho}\sigma(\beta+\hat{\rho}+\sigma) \\ \beta\hat{\rho}\sigma(\beta+\hat{\rho}+\sigma) \\ \beta\sigma(1+\beta+\hat{\rho}) \\ \beta\hat{\rho}(1+\beta+\hat{\rho}) \end{bmatrix}^\top \tag{12a}$$

$$C_\pi = \beta\hat{\rho}(1+\beta+\hat{\rho}) + \sigma(1+\beta)(1+\hat{\rho})(\sigma+\beta+\hat{\rho}). \tag{12b}$$

The age balance equations based on the SHS reset maps shown in Table I(b) for RWL location updates are:

$$\lambda^* \hat{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_6 + \mu\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{13}, \quad (13a)$$

$$(\lambda^* + \hat{\mu})\hat{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\hat{\mathbf{v}}_0\hat{\mathbf{A}}_1 + \hat{\lambda}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_2 + \mu\hat{\mathbf{v}}_4\hat{\mathbf{A}}_{14}, \quad (13b)$$

$$(\lambda^* + \hat{\mu})\hat{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_3 + \lambda\hat{\mathbf{v}}_1\hat{\mathbf{A}}_9 + \lambda\hat{\mathbf{v}}_2\hat{\mathbf{A}}_{10}, \quad (13c)$$

$$(\lambda^* + \mu)\hat{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \hat{\mu}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_7 + \lambda\hat{\mathbf{v}}_0\hat{\mathbf{A}}_8 + \lambda\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{11}, \quad (13d)$$

$$(\lambda^* + \mu)\hat{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\lambda}\bar{\mathbf{v}}_3\hat{\mathbf{A}}_4 + \hat{\lambda}\bar{\mathbf{v}}_4\hat{\mathbf{A}}_5 + \lambda\bar{\mathbf{v}}_4\hat{\mathbf{A}}_{12}. \quad (13e)$$

For the app updates described by the age process $\mathbf{x}(t)$, there is a set of SHS equations identical to (13), but with transition reset maps $\mathbf{A}_l$ in place of $\hat{\mathbf{A}}_l$ and variables $\mathbf{v}_q = [v_{q0}\ v_{q1}]$ in place of $\hat{\mathbf{v}}_q = [\hat{v}_{q0}\ \hat{v}_{q1}]$. Once again, we solve these equations numerically and it follows from Theorem 1 that average age $\mathrm{E}[\Delta]$ of the RWL app update process is given by (11).

The SHS analysis of RCU and RWL corresponding to Equations (6) through (13) incorporate preemption of updates at the FIB reader and writer. We will also consider non-preemptive versions of RCU and RWL. In these systems, app updates arriving during the reader's busy state are discarded. Consequently, when a read returns with an address, it addresses the app update that initiated the read request.

In SHS models of these non-preemptive systems, the states of Markov chains in Fig. 2 remain unchanged. However, the self transitions of rate λ are absent. In particular, the SHS transition tables of the non-preemptive RCU and RWL systems are given in Table I except transitions $l = 10, 11, 12$ in both Table I(a) and Table I(b) are deleted. The age balance equations then can be obtained from Theorem 1 in the same way that we derived (8). We don't explicitly enumerate these age balance equations here; however, in section V, we numerically compare the performance of preemptive and non-preemptive systems.

V. Numerical Results

We note that while RCU writes are lock-less, they can be still heavy as the writer tracks the start and end of a grace period, and is also responsible for memory reclamation of stale copies. On the other hand, RWL writes use locks to update the data structure. Locking requires expensive atomic operations such as compare-and-swap and thus the corresponding software functionality tends to run slow [50]. We characterize the RCU and RWL write speeds by the exponential rate parameters $\hat{\mu}_{\text{RCU}}$ and $\hat{\mu}_{\text{RWL}}$; however, it is ambiguous whether $\hat{\mu}_{\text{RCU}} > \hat{\mu}_{\text{RWL}}$ or vice-versa. Thus, in order to focus on the effects of other system parameters, our numerical evaluations assume

$$\hat{\mu}_{\text{RWL}} = \hat{\mu}_{\text{RCU}} = \hat{\mu}. \quad (14)$$

In contrast to the ambiguity associated with relative write speeds, reads in RCU are typically fast, sometimes an order of magnitude faster than uncontended locking [17]. In our SHS models, read rates are characterized by parameter μ and since read side primitives are lighter (i.e. faster) in RCU, this corresponds to $\mu_{\text{RCU}} > \mu_{\text{RWL}}$.

With the definition of the normalized rates

$$\hat{\rho} = \frac{\hat{\lambda}}{\hat{\mu}}, \quad \beta = \frac{\lambda}{\hat{\mu}}, \quad \sigma_{\text{RWL}} = \frac{\mu_{\text{RWL}}}{\hat{\mu}}, \quad \sigma_{\text{RCU}} = \frac{\mu_{\text{RCU}}}{\hat{\mu}}, \quad (15)$$

we now present some results from numerically evaluating (9), (10), (13) with $\hat{\mu} = 1$. Hence age will be measured in the units of $1/\hat{\mu}$, the average shared memory write time. As explained, our numerical evaluations consider cases with $\sigma_{\text{RWL}} < \sigma_{\text{RCU}}$, along with the further assumption $\sigma_{\text{RCU}} = 10$, corresponding to RCU reads being $10\times$ faster than RCU writes.

In addition, our evaluations will vary $\hat{\rho}$ over the interval $[0, 0.1]$. At $\hat{\rho} = 0$, the mobile terminal is stationary and never changes its network address. On the other hand, the upper limit $\hat{\rho} = 0.1$ represents an extreme value in that the average time between location changes $1/\hat{\lambda}$ is only $10\times$ longer than the average time $1/\hat{\mu}$ to write to shared memory. For example, very slow memory writes requiring time $1/\hat{\mu} = 1$ ms would correspond to $\hat{\lambda} = 0.1$ location changes per millisecond, or 100 location changes per second. While this would be an extreme level of user mobility in a traditional wireless network environment, there may be other network scenarios in which the mobile user is perhaps a software agent, for which this is appropriate. With these constraints, we aim to provide an informative comparison between RCU and RWL systems.

In Fig. 3, we plot the average app age $\mathrm{E}[\Delta]$ as a function of $\hat{\rho}$. A larger $\hat{\rho}$ means that the mobile is moving faster and changing its location more frequently, and so more app update packets are misaddressed, resulting in increased app age at the mobile terminal. In the same figure, we notice the effect of slower reads in RWL on app age. The app age at the mobile client increases in proportion to the service time of the app updates at the forwarder. Additionally, a slower read with a read lock activated corresponds to the writer being locked out without being able to write a fresher location update.

On the other hand, timely updating is achieved with RCU's fast read-side primitives and shown in Fig. 3(a). We also note that an RWL system with fast reads, say $\sigma_{\text{RWL}} = 10$, performs better than RCU with $\sigma_{\text{RCU}} = 10$, especially at higher values of $\hat{\rho}$; see Fig. 3(b). In this case, larger $\hat{\rho}$ corresponds to a greater likelihood that the FIB address is outdated, but an exclusive write lock prevents the reader from reading a stale address. The lock-less operation of RCU enables the reader to read the outdated FIB. We note that our analysis and numerical evaluations align with RCU literature that RCU is not suitable for update heavy scenarios.

From the Markov chains in Fig. 2, it is also instructive to evaluate the probability an app update is delivered. For RCU, an app update arriving in state 0 or state 3 is delivered with probability $\mu/(\lambda^* + \mu)$, which is the probability that the address read required by the app update finishes before a location update occurs or gets preempted by a fresher app update. App updates arriving in states 1, 2, and 4 are discarded primarily because the read request initiated by the updates will return a stale address as the writer is writing a fresher update

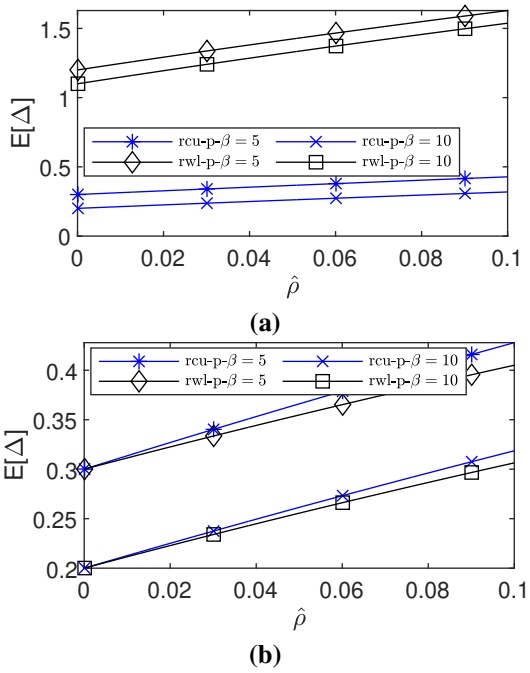


Fig. 3: AoI at mobile client when using RCU preemption (rcu -p) and RWL preemption (rwl-p) as a function of normalized write request rate $\hat{\rho} = \hat{\lambda}/\hat{\mu}$, against different values of normalized read rate $\beta = \lambda/\hat{\mu}$ and $\sigma_{\text{RCU}} = 10$ with **(a)** $\sigma_{\text{RWL}} = 1$, and **(b)** $\sigma_{\text{RWL}} = 10$.

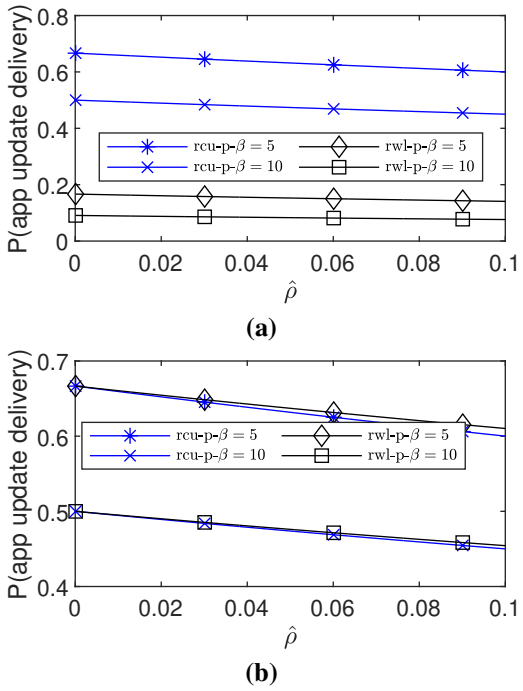


Fig. 4: Probability that an app update arriving at router is delivered correctly when $\sigma_{\text{RCU}} = 10$ and when **(a)** $\sigma_{\text{RWL}} = 1$ and **(b)** $\sigma_{\text{RWL}} = 10$.

in each of these states. Thus, the probability that an app update is delivered under RCU is

$$P_{\text{RCU}} = (\pi_0 + \pi_3)\frac{\mu}{\lambda^* + \mu}. \tag{16}$$

For RWL, app updates arriving in state 0 or state 3 are delivered with same probability as RCU, i.e. $\mu/(\lambda^* + \mu)$. App updates arriving in state 1 or state 2 are delivered with probability $[\hat{\mu}/(\hat{\mu} + \lambda)][\mu/(\lambda^* + \mu)]$. Thus, the probability that an app update is delivered is

$$P_{\text{RWL}} = (\pi_0 + \pi_1\frac{\hat{\mu}}{\hat{\mu} + \lambda} + \pi_2\frac{\hat{\mu}}{\hat{\mu} + \lambda} + \pi_3)\frac{\mu}{\lambda^* + \mu}. \tag{17}$$

Fig. 4 shows that P_{RCU} and P_{RWL} in (16) and (17) decrease as a function of normalized write request rate. In comparing Figs 3 and 4, we see that for both RCU and RWL that the average age $\text{E}[\Delta]$ becomes worse as the delivery probability decreases.

Fig. 5 demonstrates the timeliness gain achieved by employing preemption of app updates held by the reader. For $\sigma_{\text{RCU}} = 10$, $\sigma_{\text{RWL}} = 1$, and $\beta = 10$, this gain is almost 15% for RCU and 45% for RWL. From the AoI perspective, preemption helps more in RWL as it allows a slower read to service the most recent app update. Nevertheless, we note from Fig. 5 that preemption mechanisms generally reduce AoI.

In Fig. 6, we observe that the age of location updates in the memory is $\text{E}[\hat{\Delta}] \approx 1/\hat{\rho}$ for both RCU and RWL. This demonstrates that essentially all location updates are promptly stored in memory and that $\text{E}[\hat{\Delta}]$ is dominated by the relatively low frequency of location changes. This is the exception to the customary assumption that system performance improves with decreasing age. In this case, increasing the rate of location changes reduces the age of location updates in memory, but it also increases the probability that app updates go misaddressed. In this system, the timeliness of location updates would be better described using metrics such as Age of Incorrect Information [51] or Age of Synchronization [52], [53] that account for whether the current update is correct.

VI. Conclusion

This work has explored the impact of synchronization primitives on timely updating. We modeled and developed a packet forwarding scenario in which location updates from a mobile terminal are written to a forwarding table and application updates need to read the forwarding table in order to ensure their correct addressing for delivery to a mobile terminal. In this system, we saw the tension between writer and reader, both in the analytic models and in the corresponding numerical evaluations. While timeliness of the location updates in the table is desirable, excessive updating can be at the expense of timely reading of the table. We now discuss some of the many open questions and issues related to this work.

In this work, we considered only a tightly-coupled source and writer in which fresh (zero age) updates are delivered to the writer. However, there are many physical situations in which a loosely coupled source and writer would be appropriate. For example, when the source is a camera sensor and the update is an image, both image processing at the

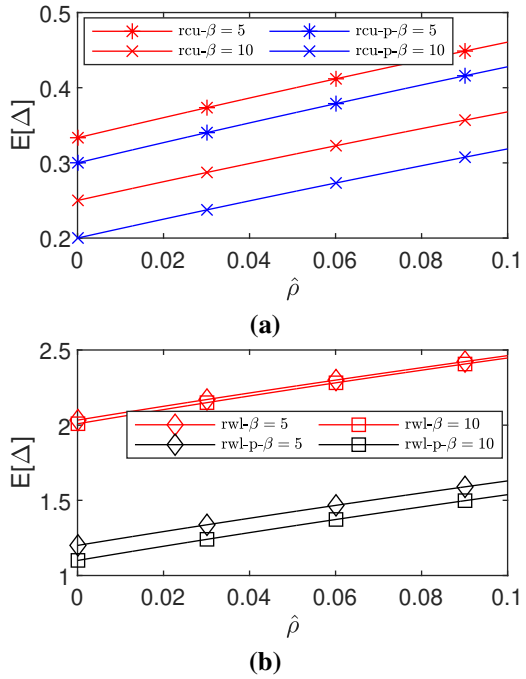


Fig. 5: AoI performance with and without preemption for **(a)** RCU with $\sigma_{\text{RCU}} = 10$, and **(b)** RWL with $\sigma_{\text{RWL}} = 1$.

sensor and transmission of the image to the writer would contribute to the update preparation time. It is obvious that this additional latency would contribute directly to the age of updates written to memory. What is perhaps less obvious is that this should prompt the writer to be parsimonious in writing. In the AoI literature, there is evidence [54], [55] that delaying new updates when the current update is relatively fresh can be age-optimal. The insight is that one should not commit system resources to producing a new update when it offers only a small age reduction relative to the current update. While the setting here is different, it seems likely that similar ideas would also be age-reducing in writing to shared memory.

We have also assumed in this work that a reader does not maintain a local cache copy of its most recent read. With a local cache, the reader can fulfill a read request by either returning the cached update or by requesting a new read lock to return a potentially fresher read from shared memory. While this optimization may improve timeliness of the delivered read, it also highlights a fundamental difference between age and latency. In responding to a client with a local copy, latency, as measured by the *turnaround time*, is reduced since the reader has unrestricted access to its local cache. On the other hand, a response that reads the shared memory is likely to be fresher. However, by virtue of mutual-exclusion, the reader may wait to complete its read of the shared memory, thus increasing the turnaround time to the client. In fact, the reader can optimize its decision making, possibly with age-dependent policies, and this will induce an *age-latency tradeoff* that needs to be explored.

We observe that RWL and synchronization primitives in general admit a combinatorial explosion of system models

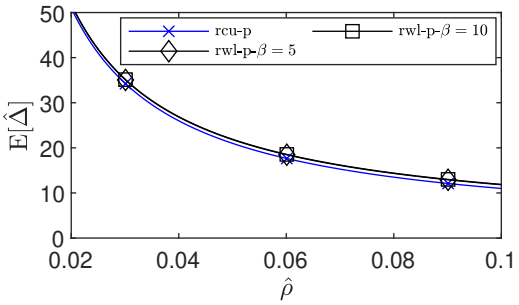


Fig. 6: AoI at memory when $\sigma_{\text{RCU}} = 10$ and $\sigma_{\text{RWL}} = 1$.

in specifying the behaviour of readers and writers. We have already described how the source-writer may be loosely or tightly coupled, how the reader may or may not maintain a local cache, and how both reader and writer may or may not employ update preemption mechanisms. Furthermore, conclusions of this work are tightly coupled to our update forwarding scenario. There is considerable work to be done in the exploring timeliness in other applications and systems employing shared memory.

REFERENCES

[1] H. Laaki, Y. Miche, and K. Tammi, "Prototyping a digital twin for real time remote control over mobile networks: Application of remote surgery," *IEEE Access*, vol. 7, pp. 20 325–20 336, 2019.

[2] R. K. Orosco, B. Lurie, T. Matsuzaki, E. K. Funk, V. Divi, F. C. Holsinger, S. Hong, F. Richter, N. Das, and M. Yip, "Compensatory motion scaling for time-delayed robotic surgery," *Surgical endoscopy*, vol. 35, pp. 2613–2618, 2021.

[3] S. Eskildsen, [Online]. Available from: https://github.com/sirupsen/napkin-math##numbers, 2020.

[4] D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, Jun. 1992. [Online]. Available: https://doi.org/10.1145/129888.129894

[5] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3320060

[6] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981. [Online]. Available: https://doi.org/10.1145/356842.356846

[7] X. Pan, J. Gonzalez, S. Jegelka, T. Broderick, and M. I. Jordan, "Optimistic concurrency control for distributed unsupervised learning," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, pp. 1403–1411.

[8] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2688500.2688501

[9] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 631–644, 2015.

[10] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using rcu balanced trees," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 199–210, 2012.

[11] M. A. Abd-Elmagid, N. Pappas, and H. S. Dhillon, "On the role of age of information in the internet of things," *IEEE Communications Magazine*, vol. 57, no. 12, pp. 72–77, 2019.

[12] K.-D. Kim and P. R. Kumar, "Cyber–physical systems: A perspective at the centennial," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1287–1308, 2012.

[13] A. Kosta, N. Pappas, and V. Angelakis, "Age of information: A new concept, metric, and tool," *Foundations and Trends in Networking*, vol. 12, no. 3, pp. 162–259, 2017.

[14] R. D. Yates, Y. Sun, D. R. Brown, S. K. Kaul, E. Modiano, and S. Ulukus, "Age of information: An introduction and survey," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 5, pp. 1183–1210, 2021.

[15] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," in *In Ottawa Linux Symposium*, 2001, pp. 338–367.

[16] P. E. McKenney, [Online]. Available from: https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html.

[17] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Trans. Parallel Distributed Syst.*, vol. 23, no. 2, pp. 375–382, 2012. [Online]. Available: https://doi.org/10.1109/TPDS.2011.159

[18] P.-J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"," *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, 1971.

[19] S. Nilsson and G. Karlsson, "Ip-address lookup using lc-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083–1092, 1999.

[20] L. trie implementation notes the linux kernel documentation, "Lc-trie implementation notes." [Online]. Available: https://www.kernel.org/doc/html/latest/networking/fib_trie.html

[21] "Userspace rcu," [Online]. Available from: https://liburcu.org/, 2021.

[22] "Knotdns," [Online]. Available from: https://www.knot-dns.cz/, 2021.

[23] "netsniff-ng," [Online]. Available from: http://netsniff-ng.org/, 2021.

[24] "Sheepdog project," [Online]. Available from: https://sheepdog.github.io/sheepdog, 2015.

[25] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ML serving," *CoRR*, vol. abs/1712.06139, 2017. [Online]. Available: http://arxiv.org/abs/1712.06139

[26] M. Arbel and A. Morrison, "Predicate rcu: An rcu for scalable concurrent updates," *SIGPLAN Not.*, vol. 50, no. 8, pp. 21–30, jan 2015. [Online]. Available: https://doi.org/10.1145/2858788.2688518

[27] I. Gelado and M. Garland, "Throughput-oriented gpu memory allocation," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 27–37. [Online]. Available: https://doi.org/10.1145/3293883.3295727

[28] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: a lightweight synchronization mechanism for concurrent programming," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 168–183.

[29] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole, "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux," *IBM Systems Journal*, vol. 47, no. 2, pp. 221–236, 2008.

[30] M. Arbel and H. Attiya, "Concurrent updates with rcu: Search tree as an example," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 196–205. [Online]. Available: https://doi.org/10.1145/2611462.2611471

[31] J. Kim, A. Mathew, S. Kashyap, M. K. Ramanathan, and C. Min, "Mv-rlu: Scaling read-log-update with multi-versioning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 779–792. [Online]. Available: https://doi.org/10.1145/3297858.3304040

[32] M. Kokologiannakis and K. Sagonas, "Stateless model checking of the linux kernel's read–copy update (rcu)," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 3, pp. 287–306, 2019.

[33] ——, "Stateless model checking of the linux kernel's hierarchical read-copy-update (tree rcu)," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 172–181. [Online]. Available: https://doi.org/10.1145/3092282.3092287

[34] L. Liang, P. E. McKenney, D. Kroening, and T. Melham, "Verification of tree-based hierarchical read-copy update in the linux kernel," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 61–66.

[35] J. Tassarotti, D. Dreyer, and V. Vafeiadis, "Verifying read-copy-update in a logic for weak memory," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 110–120, 2015.

[36] D. Dice and N. Shavit, "Tlrw: Return of the read-write lock," in *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 284–293. [Online]. Available: https://doi.org/10.1145/1810479.1810531

[37] D. Dice and A. Kogan, "Bravo—biased locking for reader-writer locks," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 315–328.

[38] R. Liu, H. Zhang, and H. Chen, "Scalable read-mostly synchronization using passive Reader-Writer locks," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 219–230. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/liu

[39] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit, "Numa-aware reader-writer locks," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 157–166.

[40] J. P. Hespanha, "Modelling and analysis of stochastic hybrid systems," *IEE Proceedings-Control Theory and Applications*, vol. 153, no. 5, pp. 520–535, 2006.

[41] R. D. Yates and S. K. Kaul, "The age of information: Real-time status updating by multiple sources," *IEEE Transactions on Information Theory*, vol. 65, no. 3, pp. 1807–1827, 2018.

[42] R. D. Yates, "Age of information in a network of preemptive servers," in *IEEE Conference on Computer Communications (INFOCOM) Workshops*, Apr. 2018, pp. 118–123, arXiv preprint arXiv:1803.07993.

[43] S. Farazi, A. G. Klein, and D. R. Brown, "Average age of information for status update systems with an energy harvesting server," in *IEEE Conference on Computer Communications (INFOCOM) Workshops*, April 2018, pp. 112–117.

[44] A. Maatouk, M. Assaad, and A. Ephremides, "Minimizing the age of information: NOMA or OMA?" in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2019, pp. 102–108.

[45] S. Kaul and R. Yates, "Age of information: Updates with priority," in *Proc. IEEE Int'l. Symp. Info. Theory (ISIT)*, Jun. 2018, pp. 2644–2648.

[46] A. Maatouk, M. Assaad, and A. Ephremides, "On the age of information in a CSMA environment," *IEEE/ACM Transactions on Networking*, pp. 1–14, 2020.

[47] R. D. Yates, "The age of information in networks: Moments, distributions, and sampling," *IEEE Transactions on Information Theory*, vol. 66, no. 9, pp. 5712–5728, 2020.

[48] M. Moltafet, M. Leinonen, and M. Codreanu, "Moment generating function of the AoI in a two-source system with packet management," *IEEE Wireless Communications Letters*, vol. 10, no. 4, pp. 882–886, 2021.

[49] ——, "Source-aware packet management for computation-intensive status updating: MGF of the AoI," in *2021 17th International Symposium on Wireless Communication Systems (ISWCS)*, 2021, pp. 1–6.

[50] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, dec 2007. [Online]. Available: https://doi.org/10.1016/j.jpdc.2007.04.010

[51] A. Maatouk, S. Kriouile, M. Assaad, and A. Ephremides, "The age of incorrect information: A new performance metric for status updates," *IEEE/ACM Transactions on Networking*, vol. 28, no. 5, pp. 2215–2228, 2020.

[52] J. Cho and H. Garcia-Molina, "Effective page refresh policies for web crawlers," *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 4, pp. 390–426, 2003.

[53] J. Zhong, R. Yates, and E. Soljanin, "Two freshness metrics for local cache refresh," in *Proc. IEEE Int'l. Symp. Info. Theory (ISIT)*, Jun. 2018, pp. 1924–1928.

[54] Y. Sun, E. Uysal-Biyikoglu, R. D. Yates, C. E. Koksal, and N. B. Shroff, "Update or wait: How to keep your data fresh," *IEEE Trans. Info. Theory*, vol. 63, no. 11, pp. 7492–7508, Nov. 2017.

[55] R. Yates, "Lazy is timely: Status updates by an energy harvesting source," in *Proc. IEEE Int'l. Symp. Info. Theory (ISIT)*, June 2015, pp. 3008–3012.