STORING, RETRIEVING, AND PROCESSING UPDATES: A TIMELINESS PERSPECTIVE

By

VISHAKHA RAMANI

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Prof. Roy D. Yates

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October 2024

ABSTRACT OF THE DISSERTATION

Storing, Retrieving, and Processing Updates: A Timeliness Perspective

by VISHAKHA RAMANI

Dissertation Director: Prof. Roy D. Yates

Time-critical applications, such as virtual reality and cyber-physical systems, require not only low end-to-end latency, but also the timely delivery of information. While high-speed Ethernet adoption has reduced interconnect fabric latency, bottlenecks persist in data storage, retrieval, and processing. This work examines status updating systems where sources generate time-stamped updates that are stored in memory, and readers fulfill client requests by accessing these stored updates. Clients then utilize the retrieved updates for further computations.

The asynchronous interaction between writers and readers presents challenges, including: (i) the potential for readers to encounter stale updates due to temporal disparities between the writing and reading processes, (ii) the necessity to synchronize writers and readers to prevent race conditions, and (iii) the imperative for clients to process and deliver updates within strict temporal constraints.

In the first part, we study optimal reading policies in both discrete and continuous time domains to minimize the Age of Information (AoI) of source updates at the client. One of the main contributions of this part includes showing that lazy reading is timely. In the second part, we analyze the impact of synchronization primitives on update timeliness in a packet forwarding scenario, where location updates are written to a shared routing table, and application updates read from it to ensure correct delivery. Our theoretical and experimental results show that using a lock-based primitive is suitable for timely application update delivery at higher location update rates, while a lock-free mechanism is more effective at lower rates. The final part focuses on optimizing update processing

when updates require multiple sequential computational steps. We compare the age performance across a multitude of pipelined and parallel server models and characterize the age-power trade-off in these models. Additionally, our analysis reveals that synchronous sequential processing is more conducive to timely update processing than asynchronous methods, and that parallel processing outperforms pipeline services in terms of AoI.

To Dadiji...

ACKNOWLEDGMENTS

While a PhD is a solitary endeavour, I have been fortunate to have found a support system that made this arduous journey seem less daunting. I expected to be enervated at the end of this journey, but thanks to the faculty, colleagues, friends and family, I look forward to the next phase of my life with alacrity.

To begin, I would like to express my deepest gratitude to my advisor, Prof. Roy Yates. When I first considered pursuing a PhD, I shared my apprehensions with him about my mathematical proficiency and at being perplexed about a proper approach to research. His advice was transformative: "I think you should try to push yourself toward analysis/theory. You'll know in six months if this is the right step for you and whether you are excited by what you are learning in the papers you're reading. There were math topics I couldn't understand at age 19 that I had no trouble with at age 22. It's a lot like lifting weights or speaking a foreign language; if you put in the time, you can master new skills."

Although, I still consider myself a novice in mathematical analysis and research in general, any progress I have made, it was not just me putting in the time, it was also due to Roy's guidance, as he consistently nudged me in the right direction. For all his academic guidance and beyond (which mostly includes bikes and bike racing), I am profoundly grateful.

I would also like to extend my immense gratitude towards Prof. Dipankar Raychaudhuri. My initial foray into networks research began with a master's thesis under his supervision. It is thanks to his weekly group meetings, which fostered open discussions, that I learned nearly everything I know about how networks are built. Though my core research ended up drifting towards theoretical analysis, a minuscule fraction of his vast knowledge in the field seeped into my own research ethos, inculcating the habit of diligently considering each design choice in a system. Somehow this opens up a range of important research questions, transforming seemingly innocuous problems into scintillating ones. I am also deeply thankful to him and his wife, Prof. Arundhati Raychaudhuri, for the delectable dinners and for never letting me feel alone during the holidays. Without their support,

none of this would have been possible.

I am also very grateful to the other members of my thesis committee. My sincere thanks to Prof. Emina Soljanin, who helped me connect some of the ideas in this thesis to problems in traditional distributed computing, and also for TA lunch. I also want to thank Prof. Yao Liu for her valuable insights into various applications related to the work in this thesis. Special thanks to Prof. Sennur Ulukus, who, as she mentioned, is my academic sister, for her critiques and suggestions on this thesis and on my research in general.

I am grateful to the professors from whom I've had the privilege of learning. Prof. Predrag Spasojevic introduced me to WINLAB and inspired many "light-bulb" moments in his classes. Prof. Waheed Bajwa's course demystified the complex subject of stochastic systems, providing a conceptual foundation for my thesis. Although I never took a class with Prof. Anand Sarwate, his advice during our weekly meetings greatly enhanced my ability to communicate research effectively. I also extend my thanks to Richard Howard for offering a physicist's perspective on everyday problems. Finally, I thank my collaborator, Yu-Pin Hsu; here's to solving many more interesting problems together.

The past two summers have been profoundly rewarding, largely because of my experience interning at IBM Research. I'm especially thankful to Sara Kokkila Schumacher, who is not only a brilliant researcher but also an exceptional mentor. She inspired and supported me in pursuing my scholarly interests during my time there. I am also deeply grateful to my colleagues at IBM Research, including Diana Arroyo, Marquita Ellis, Olivier Tardieu, Asser Tantawi, Priya Nagpurkar, and Alaa Youssef, who generously devoted their time and effort to assist me whenever I encountered challenges.

No PhD thesis associated with WINLAB would be complete without acknowledging Ivan Seskar. His meticulous critiques of our model assumptions often led to improved models that better reflected real-world problems. The warm demeanor and support of Jake Kolodziejski and Jenny Shane helped foster a welcoming and inclusive lab environment. I am also grateful for Ivan's epicurean cookie spread , which often served as a light-hearted incentive to rate his wildlife photos

and provided the backdrop for engaging conversations on a wide range of topics, from nature and science to technology, medicine, politics, and the universe at large. I will always cherish our hobnobbing over HobNobs. I'm thankful for the support of WINLAB's outstanding staff. Thanks to Noreen DeCarlo for assisting with the conference travel, and to Lisa Musso for managing all the administrative requirements.

I'd like to thank my peers I've met along on my graduate study journey. They became my friends and inspired me in multiple ways. To Shreyasee and Shalini, your warmth and friendship have been a constant source of comfort, making me feel at home. Exploring NYC and savoring pani puri together are the memories I treasure dearly. To Carolina, thank you for being my study companion and a fellow Broadway enthusiast. To Sharvani, thank you for being a confidant and a supportive friend. Thanks to Wachirawit, whom I met during the internship at IBM Research, for your friendship and introducing me to a great circle of friends with whom I enjoyed many game nights. To Filippo, thank you for your patience with our yet-to-happen European tours. Siddharth, your kindness and help in both personal and professional matters have been invaluable. Thanks to my friends outside of academia – Leesha, Pallavi, Aditya, Sukanya, Kriti, and Ranjeet – who are like family to me.

This last paragraph is for my family. To my cousins – Harsha, Bhavisha, Divya, Rekha, Rashmi, Jitesh, Gautam, and Shailendra – thank you for all your love and for keeping me sane with your laughter and playfulness. To my nephews, Romesh and Sagan, thank you for letting me love you. Thank you to Gopal Uncle for being a beacon of positivity and for numerous wise teachings. To my sister and brother-in-law, thank you for being my best friends. Whether it was through late-night conversations, shared laughter, or just being there when I needed you most, you've both been my rock. To Ma and Papa, for everything! There are not enough thank yous in the world. I owe every success to the lessons you taught me, and I hope to make you proud in all that I do. And lastly, to Devanshu, whom I am lucky to have found and would have been lost without.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Part I

Introduction

CHAPTER 1
TIMELINESS IN DISTRIBUTED SYSTEMS

1.1 Age-of-Information (AoI) as a Network Performance Metric

Emerging time-critical applications, including Autonomous Vehicles, Augmented Reality (AR), Virtual Reality (VR), remote telesurgery, and multi-player cloud gaming, have a common Quality-of-Service (QoS) requirement: The information received needs to be fresh at the concerned destination. For example, in autonomous vehicle systems, vehicles share their position and velocity status information with each other. In an urban setting, a vehicle traveling at a typical speed of 36 kilometers per hour moves approximately 1 centimeter in 1 millisecond. This implies that if a vehicle receives information about the position of a nearby vehicle with a delay of 1 millisecond, it would still have a fairly accurate idea of the nearby vehicle's position within 1 centimeter. Information received with a larger delay leads to the vehicle having outdated status information, leading to more position uncertainty of the nearby vehicle.

In VR systems, user actions are detected by sensors, processed by the system, and translated into corresponding visual feedback that is rendered and perceived by the user via a Head-Mounted Display (HMD). Timely frame updates ensure that the displayed environment accurately reflects the user's movements and interactions.

In the aforementioned applications, it is crucial to note that *sending information either too often or too infrequently can be detrimental*. Increasing the frequency of messages puts a load on the network which is limited by available bandwidth. While the effect of a single vehicle may be negligible, many vehicles, especially in an urban setting, exchanging information at an excessive message rate can create significant network congestion. In a VR setup, if the display time on a Head-Mounted Display (HMD) is less than 15ms i.e. each frame of the virtual environment is being updated at a rate faster than 60 frames per second (fps), the frames become useless since the human

brain cannot distinguish between consecutive frames at such a high frequency[1]. Conversely, we also observed that reducing the information rate can lead to the recipient having unnecessarily stale information due to a lack of frequent updates leading to faulty safety maneuvers by an autonomous vehicle, or motion sickness in VR scenario.

The nature of these applications underscores the need to adapt the source rate of status messages to an optimal operating point. This concept lies at the core of *timely updating* [2]. The study of timely updating uses a consistent model: a source generates time-stamped status update messages that traverse a communication system before reaching a monitor. The objective of real-time status updating is to maximize the timeliness of the relevant status information at each monitor [3]. This focus on real-time updating led to the introduction of *Age of Information* (AoI) performance metric that describes the timeliness of a monitor's knowledge of an entity or process [4]. An update bearing a timestamp u is characterized by its *age* denoted as $\Delta = t - u$ at a time $t \geq u$. A monitor receiving a stream of updates has age process $\Delta(t) = t - u(t)$ when $u(t)$ is the time-stamp of the most recently received update. The concept of age signifies the elapsed time since the last update was received at the monitor, indicating that freshness diminishes as age increases.

Since the introduction of AoI in the seminal paper by Kaul et al. [3], numerous studies have focused on addressing the timeliness of status updates in various queues and networks, presenting new analytical models and tools for age analysis, and utilizing age-based QoS metrics for applications such as mobile cloud gaming, caching, and learning. For a comprehensive review of AoI advancements, readers are encouraged to consult the surveys by Yates et al. [4] and Kosta et al. [5], as well as the references therein.

1.2 Introduction to Producer-Consumer Paradigm

At the core of many distributed systems and software architectures lies a classic concurrent programming design pattern – the producer-consumer paradigm [6]. In this paradigm, there are typically two main entities: producers and consumers. Producers generate data or events and push them

[1]Research indicates that the human brain's visual system can reliably extract conceptual information from visual input within a minimum viewing time of approximately 15ms [1].

into a shared space. Consumers, on the other hand, retrieve data or events from this shared space and process them asynchronously. This model of interaction fosters loose coupling among system components, allowing components to interact without having to be aware of each other's existence or state at the time of communication. This concept of producer-consumer paradigm and the concurrent execution of concerned entities is applicable across both single-node and multi-node systems as well as to a myriad of applications, spanning domains such as autonomous vehicles, online gaming, industrial automation, packet routing, flight control systems, financial trading platforms etc.

Such decoupling affords several notable advantages. First, it ensures failure isolation, as faults within one component are contained and prevented from propagating to others, thereby enhancing system resilience. In addition, the paradigm promotes flexibility, enabling independent modifications to producers or consumers without disrupting the functionality of the counterpart. The model also supports scalability by allowing multiple producers and consumers to operate in parallel, facilitating horizontal scaling to accommodate increased workload. Furthermore, asynchronous communication inherent in this paradigm can reduce latency by permitting producers to continue generating data without waiting for immediate consumer processing [7].

Here, application logic is decomposed into smaller modules and each module independently focuses on specific tasks. For an application running on a single-node multi-processor system, producers and consumers constitute concurrently running processes. For example, in real-time gaming, various game aspects such as input processing, physics simulation, rendering, and networking are handled by different processors on the gaming machine. The input processing module acts as a producer, broadcasting player actions (e.g., keyboard inputs, mouse clicks) to consumer processes responsible for updating the game world, triggering animations, and handling player interactions [8].

In a distributed system with multiple multicore nodes, producers and consumers are spread across different machines, necessitating network access for communication [9]. For instance, suppose we have a large dataset consisting of millions of documents, and we want to perform word count analysis on these documents. We can use MapReduce [10] to distribute the workload

across multiple nodes in a distributed system. During the Map phase, each node acts as a producer, extracting words from documents and emitting intermediate key-value pairs (word, 1). These pairs are then shuffled, sorted, and consumed by reducer nodes in the Reduce phase, where each node acts as a consumer. The reducers aggregate the word counts and produce the final output [11, 10].

1.2.1 Communication between Producers and Consumers

This producer-consumer paradigm, while beneficial for improving compute performance, also carries a significant communication overhead between concurrently running, decoupled modules. In distributed computing, two primary communication models have been explored: the message passing model and the shared memory model [12, 13]. In the message passing model, we envision a network where processors are distributed across nodes and interconnected by reliable communication channels that maintain message integrity. Processes communicate via send and receive primitives, akin to TCP-like protocols found in modern networks. In the alternative shared memory model, communication is based on abstraction of a hardware shared memory in which processors communicate by writing and reading to shared registers [14].

This communication between processes is known as Inter-Process Communication (IPC). There is no clear consensus on the optimal IPC method for applications. Each model possesses its own advantages, benefiting algorithms in distinct ways [13]. Interestingly, despite their differences, it has been demonstrated that the message-passing and shared-memory models are equivalent, suggesting that one model can simulate the other [12]. Whether through a message passing mechanism where updates are pushed to a queue and consumers pull the update from the queue, or a memory-based mechanism where updates are written to and read from a shared memory location, the overall idea of the producer-distribution-consumer model remains consistent: storing data from producers, retrieving, and processing it on consumers.

There have been many studies on the impact of IPC mechanisms on the throughput and latency of an application [15, 16, 17, 18], however, with respect to status updating systems, the bottlenecks introduced through these mechanisms on timely storing, retrieving and processing of data have been

largely unexplored. This thesis aims to identify these bottlenecks, how they affect the timeliness in status updating systems, and propose optimization strategies for these systems based on a comprehensive understanding of these bottlenecks. While the impact of each IPC mechanism on the timeliness can inspire individual thesis as each mechanism engenders a separate system model to study, in this thesis our focus is on memory as an IPC choice. For our work, the memory based IPC is an abstraction that represents hardware shared memory if an application is running on a single node multi-processor machine. It also represents software distributed shared memory if an application is spread across multiple nodes.

1.3 Timeliness in Shared Memory Systems

In the burgeoning field of AoI, this thesis aims to address a significant gap in AoI research by investigating timeliness in storing, retrieving and processing of updates in a producer-consumer systems using shared memory as a means of information dissemination. In this work, we focus on a class of system (see Fig. 1.1) in which a source generates time-stamped *updates* (i.e., measurements of a random process of interest representing some real-world phenomenon) and a shared data structure stores these updates[2]. Going forward, we refer to the shared data structure as *shared memory*, or simply as *memory*. In this setup, a Writer queries fresh measurements from the source or sensor to update the memory, while a Reader satisfies clients' requests for these measurements by accessing the memory. We define a client as an entity that utilizes the source updates to perform computations. The computed updates are then transmitted to a monitor. Notably, the client can either coincide with the Reader, operating within a single process, or exist as a separate process. In such systems, a source-writer pair is the producer while the reader-client pair is a consumer.

As shown in Fig. 1.1, age process $x(t)$ is the age of a data item (update) in the memory and it describes how fresh an update in the memory is with respect to the real world. The Reader generates an age process $y(t)$ at the input to the client as it receives samples of the updates published in the memory. Thus, $y(t)$ describes how fresh the update with the client is with respect to the update in

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

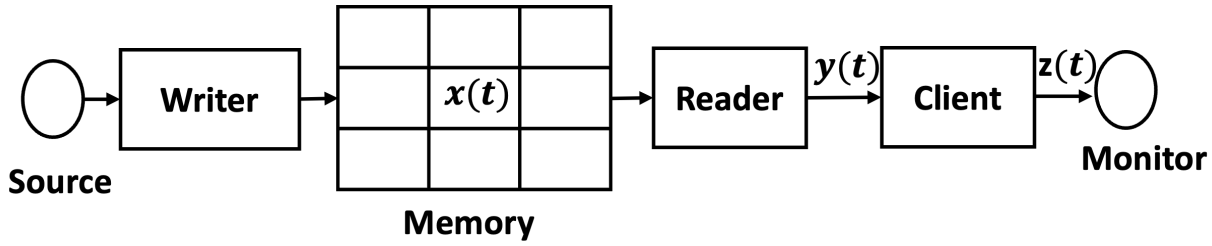


Figure 1.1: A writer updates memory based on the update received from source. A client requests the Reader process to read the source updates from the memory. The source update publication in the memory generates age process $x(t)$, and the update sampling by the Reader generates age process $y(t)$ at the client input. The computation delay at client generates age process $z(t)$ of the processed update.

the memory. Ideally, both the updates in the memory and those received by the client should offer a timely representation of the real world. Consequently, there are measures that can be taken to minimize the age in memory, ensuring that updates are as fresh as possible when stored. Similarly, efforts can be made to minimize the age at the client, ensuring that the client receives updates promptly after they are published in the memory.

However, the asynchronous nature of reader-writer interactions within memory systems introduces significant challenges. First, optimizing readers' memory accesses is crucial for timely processing of source updates. Readers may only become aware of fresher updates in the memory when they choose to query it. Second, the memory access process must be regulated by a synchronization method between readers and writers to prevent race conditions. Without proper synchronization, simultaneous access by multiple readers or writers may lead to data corruption or inconsistent states within the memory.

Furthermore, these issues are coupled. Consider two widely used synchronization primitives: lock-based primitive Readers-Writer Lock (RWL) [19, 20] and lock-less primitive Read-Copy-Update (RCU) [21]. RWL enforces mutual exclusion between readers and writers. This means that while multiple readers can access data concurrently, writers require exclusive access, thereby ensuring that updates are written in place. On the other hand, RCU provides non-blocking access to readers. Writers, instead of directly modifying the current data in the memory, create a copy and apply modifications to the copy. This mechanism allows readers to continue accessing the original

data without waiting for the writer to finish updating.

Ideally, we could minimize $x(t)$, the age of updates in memory, by writing too frequently. However, this approach would lead to readers getting blocked if RWL is used, resulting in increased age $y(t)$ at the client. Even with RCU, frequent writing could lead to readers reading stale copies, potentially resulting in incorrect computations on the client side. Moreover, using RCU may result in the creation of multiple versions of updates, minimizing $x(t)$ but increasing memory usage. On the other hand, one could attempt to minimize $y(t)$, the age of updates at the client, by reading memory frequently, but with RWL, this may block the writer from writing fresh source updates, increasing $x(t)$. Even with RCU, frequent reading does not guarantee that the freshest update will be read as the writer could be in the process of writing a new update, resulting in compute resource waste.

As such, there is a single question that forms the core of the most part of the thesis:

'How can we minimize the age processes $x(t)$ and $y(t)$?'

As discussed, the answer to this question depends on the type of synchronization primitive used and the policy employed by the Reader to sample the memory. Accordingly, the first five chapters, specifically Chapters 3 - 8, can be broadly categorized under two research themes: the development of optimal policies for timely memory sampling (Part II), and the impact of synchronization primitives, specifically RCU and RWL, on AoI (Part III).

Finally, we acknowledge that the system model depicted in Fig. 1.1 represents the simplest form of memory systems, but in practice, there can be various flavors of such systems. The real-world systems often entail multiple sources and readers, where each source update is associated with a particular Reader. Multiple reader scenarios resemble multiple access channels, where readers must take turns and share access to memory, adhering to various constraints. However, the crux of the issue in these systems lies in reader-writer contention. Even with a single reader, reader-writer contention remains a fundamental problem that has not received extensive exploration in the Age of Information (AoI) literature.

Additionally, practical systems may involve a "preparation time" that the source requires to

produce or deliver an update in response to a writer's query. However, to focus on the impact of shared memory, we make the assumption throughout this work that preparation times are negligible. Consequently, the writer receives fresh (zero age) updates from the source. We refer to such a source and writer as "tightly coupled". We now give an overview of the first two research themes of this thesis.

1.4 Research Theme 1: Optimizing Memory Access

In this area of research, our focus is on developing optimal policies for memory access when readers have unrestricted access to the memory. This means allowing the Reader to read whenever it chooses, without any restrictions imposed by synchronization primitives. We start with the simplest case in Chapter 3, where we consider a discrete-time slotted system, and assume that the Reader is notified when an update is published in the memory, enabling the Reader to know the update age in the memory. At each time slot, the Reader determines whether to access the memory and read a source update. We consider a scenario where a non-negative fixed cost c is associated with reading the memory during each time slot. We formulate a discrete-time decision problem to find a sampling policy that minimizes average cost comprising age at the client and the cost incurred due to sampling. We establish that an optimal policy is a stationary and deterministic threshold-type policy, and subsequently derive optimal threshold and the corresponding optimal average cost.

In Chapter 4, we extend the study in Chapter 3 by analysing the model where Reader is unaware of the memory state. We develop and analyse various heuristic algorithms that provide practical solutions in scenarios where the memory state is unknown. We will see that the performance of heuristics can approach the lower bound (known-state average cost) under certain conditions, such as low sampling costs, or high source update probability in the memory.

Chapter 5 focuses on the optimization of client operations within a continuous-time system. Here, the client, termed as a Decision Process (DP), is tasked with deriving a *decision update* from a pair of source updates, which it then delivers to the monitor. The main goal is to minimize the age of this decision update at the client's output, represented by the age process $z(t)$ in Fig. 1.1. We first

analyze the stationary expected age both at the input and output of the DP, denoted by $\mathrm{E}[y(t)]$ and $\mathrm{E}[z(t)]$, respectively. We show that a lazy computation policy in which the DP may sit idle before computing its next decision update can reduce the average AoI at the monitor even though the DP exerts no control over the generation of source updates. In this policy, the DP Reader, responsible for fetching source updates for the DP, decides to read after a random time T, representing the DP's computation time. Subsequently, it may choose to wait for a time $W \geq 0$ before fetching the next sample from memory. In this work, the Reader operates without knowledge of the update age in memory, and as such the Reader's inter-sample times are independent of the age process in the shared memory, forming a renewal process. The rationale behind the optimality of lazy sampling lies in its ability to mitigate the detrimental impact of high-variance computation times T on the sampling policy employed by the DP Reader.

1.5 Research Theme 2: Impact of Synchronization Primitives on AoI

In the second part of this thesis, we consider a conventional packet forwarder node which is a shared memory multiprocessor machine running a high speed packet processing software that provides classical routing and switching functionalities [22]. The packet forwarder maintains a Forwarding Information Base (FIB), a shared data structure accessed by both readers and writers. When the user moves, it sends its new address (location update) to the packet forwarder, and so the Writer is a process that writes location updates in the FIB. 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.

Usually, concurrent access to the FIB is protected by one of the two fundamental synchronization primitives — Read-Copy-Update (RCU) or Readers-Writer Lock (RWL). With RWL, old location update will be read if an app update addressing has locked out the Writer with a newer location update. With RCU, app update addressing will be incorrect if the Writer is in process of writing a new location update. The misaddressed app updates are never received at the mobile terminal,

thereby increasing the age of app updates at the user. The impact of these primitives on timely mobile routing is theoretically analyzed in Chapter 6. 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.

In Chapter 7, we employ a high-speed packet processing testbed to quantitatively analyze the packet forwarding application running on a shared memory multi-processor architecture. While modern packet processing frameworks are optimized for maximum packet throughput, their ability to support timely delivery remains an open question. Here we focus on the age of information performance issues induced by throughput-focused packet processing frameworks. Our results underscore the importance of careful selection of offered load parameters and concurrency constructs in such frameworks.

Finally, Chapter 8 explores the trade-off between memory footprint and the average age of updates in shared memory when employing RCU. RCU is a synchronization primitive that allows for concurrent and non-blocking read access to fresh data. This is achieved through the creation of updated data copies, with each prior version retained until all associated read-locks are released. Given the principle that frequent updating keeps information fresh, the concern is whether the accumulation of update copies leads to excessive memory usage. To address this, we analyze trade-offs between memory usage and update age within real-time status updating systems, focusing specifically on RCU. The analysis demonstrates that with finite read time and read request rate, the average number of updates within the system remains bounded.

1.6 Timeliness in Update Processing

In status updating systems built on the producer-consumer paradigm, two critical requirements must be met. First, the client must process updates from producers that accurately reflect the current state

of the system or environment. This means that the input provided to the client must be logically correct, ensuring it represents a timely snapshot of the world. We quantify this input correctness with age of the update at the client input. The second requirement is that not only the client should work on timely updates, but the output of processing should be temporally correct. This means the output update must reach the end user at the correct time to maintain its relevance. We measure this temporal correctness by the age of the update at the client's output, which ensures that the end user receives timely information [3].

For example, cameras at a smart-city intersection [24] can capture video or images of the intersection and send this data to a processing system at the edge. Here, the cameras are the producers and the processing system is the consumer. The system may then process the data—for instance, by running object detection and tracking algorithms—to identify and alert pedestrians crossing the street about an approaching speeding vehicle. In this scenario, the input data (video or images) must be timely to accurately represent the environment, and the output (warnings to pedestrians) must be delivered promptly to be effective.

As discussed in Section 1.4, in Part II of this thesis, our focus will be on optimizing memory access to minimize the age at the client input. Similarly, in Part IV, we concentrate on optimizing the timely delivery of processed updates to the user. However, before providing an overview of our work, which falls under Research Theme 3, we first examine some fundamental aspects of update processing in practical systems that motivate the problem and the corresponding analysis.

1.6.1 Update Processing: The Many Ways

Typically, update processing involves executing a sequence of computational steps. For example, in the aforementioned example of smart-city intersection, the object detection for collision prevention involves a sequential processing of tasks, including pre-processing on input images, feature extraction, followed by object classification.

However, there can be different modes of processing a source update. These modes are usually

[3] A similar, albeit different, dual notion of correctness in real-time producer-consumer systems was first studied in [23].

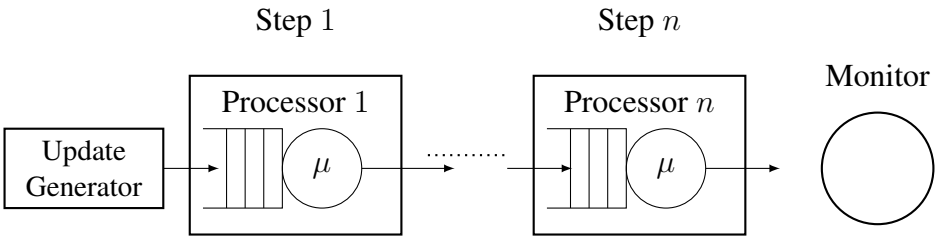


Figure 1.2: n processors in series for source update processing. Processor i is responsible for computation step i.

dictated by system design choices in terms of number of processors deployed to process an update, the underlying communication mechanism between processors as well as energy consumption constraint.

One approach to processing a source update, which requires a sequence of n computation steps, involves deploying n loosely coupled processors. In this configuration, each processor performs one step in the update's processing pipeline. This setup introduces an asynchronous pipeline mechanism, where the output of each processor serves as the input to the subsequent processor. From a queueing theory perspective, this can be modeled as a tandem queue (also known as a series queue) with n servers.

In practical implementations, this asynchronous pipeline is typically realized through low-latency communication between processors. This can be achieved via message-passing mechanisms using queues, or by employing a shared memory paradigm. Fig. 1.2 depicts an asynchronous pipeline setup utilizing queues, where the output of each processor is enqueued as an update for the subsequent processor. Alternatively, with a shared memory approach, each processor writes the result of its computation to a shared memory location, and the next processor retrieves this result by reading from memory.

In contrast, another processing paradigm involves the use of multiple parallel processors, where each processor independently executes all n computation steps. This parallel processing configuration, as illustrated in Fig. 1.3, features m processors running in parallel, each completing the entire sequence of computation steps.

1.6.2 AoI and Multi-step Processing Systems

Regardless of the mode of processing, it is imperative that the system delivers timely processed updates such that the age at the end user is minimized. Despite its importance, the study of timely multi-step update processing remains unexplored in AoI literature. This work aims to bridge that gap by investigating the age performance of two-step ($n = 2$) update processing systems, a fundamental building block for more complex processing systems. Even within this seemingly simple two-step framework, rudimentary questions arise. For instance, which configuration-—series or parallel processing—-proves more effective in maintaining timeliness? Answering this question, however, is far from straightforward and poses a considerable challenge.

In a series server setup, modeled as a tandem queue, each service facility may operate under different service disciplines, such as lossless First-Come-First-Served (FCFS) or lossy Last-Come-First-Served (LCFS) with preemption in either waiting or service stages. These configurations introduce a plethora of complexities. While there is an extensive literature on age performance of fresh arrivals in single-source single-server queues employing various service disciplines (see [4] and the references therein), in our work however, updates arrive at server 2 from server 1 with some existing age, which must be accounted for by the analysis.

Furthermore, the analysis of parallel processor setups with only two servers is equally non-trivial. A server may be "late" in delivering an update, while another parallel server has already delivered a fresher update, rendering the former's delivery inconsequential in terms of age reduction. This scenario underscores the necessity of developing a novel analytical framework to properly evaluate such systems.

1.7 Research Theme 3: Timely and Energy Efficient Multi-step Update Processing

Upon examining the age performance of series and parallel server setups, we observe that there is no straightforward answer to which configuration is superior. Instead, both setups exhibit a phenomenon we term as "wasted power," where computational resources are expended on processing updates

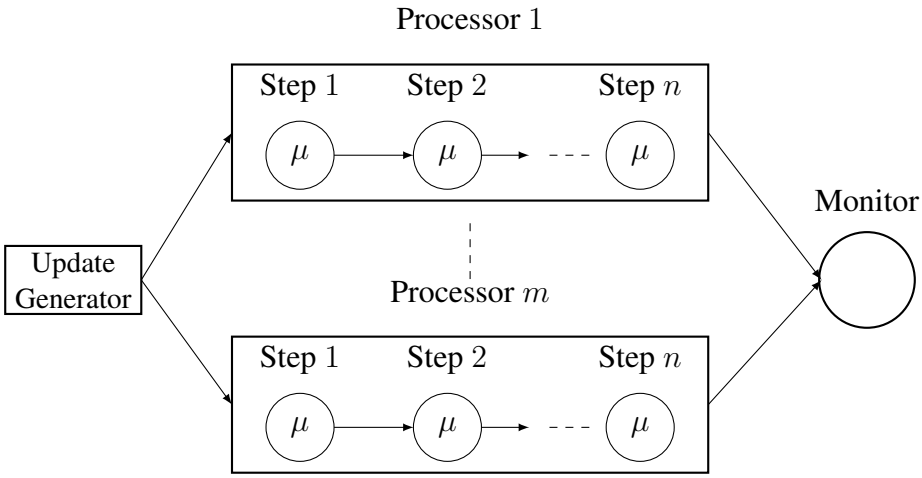


Figure 1.3: Parallel processors setup for source update processing.

that ultimately do not contribute to reducing the age.

In parallel server setups, wasted power occurs when one server completes processing a fresher update before others, thereby rendering the efforts of the remaining servers, still processing older updates, inconsequential. The resources dedicated to these outdated updates are thus squandered. Conversely, in series server setups, wasted power arises when a server preempts its current task upon receiving a fresher update from the preceding server. The computational resources previously invested in processing the preempted update are essentially wasted. Furthermore, there exist operational regimes where a server may discard updates received from the preceding server, nullifying the work already performed.

An additional source of inefficiency in series setups is server idleness. Servers may remain inactive while awaiting the completion of the preceding server's processing task. This idle time represents lost computational potential, as these resources could have been employed to process other updates, potentially improving age performance.

1.7.1 Opportunities in Optimizing Age Performance

The aforementioned observations raise important questions regarding the age-power trade-off in these systems. How can we optimize the allocation of computational resources to minimize wasted power while simultaneously reducing the age? One promising approach to understanding this

trade-off is to identify the optimal service rates that achieve the best system age performance under a fixed power budget. In the context of two-step update processing using two tandem servers, a qualitative characterization of optimal service rates emerges: given a power constraint, should the second server operate faster than the first?

On the other hand, the parallel server model presents an array of challenges in age optimization. The optimal policy depends on various factors, often contingent on the information available to each server about the status of the other.

Consider a scenario where the service time of any update at any server is identical to random variable X, independent of the service time of that update or any other update at any other server. This setup leads to a Markov state representation, $[Y_1(t), Y_2(t), A(t)]$, where $Y_i(t)$ denotes the service already received by an update at server i, and $A(t)$ represents the age at the monitor. If servers are processing fresh updates, then $Y_i(t)$ also corresponds to the age of the update currently in service at server i. Additionally, $Y_i(t) = 0$ indicates that server i is idle.

In this model, let i denote one server and j denote the other. Consider a scenario where the state is given by $Y_i = y$, $Y_j = 0$, and $A = a$, with server j idle. A decision must be made: should server j begin processing a new update, or should it remain idle for a period? If server i has only just started its processing, with Y_i close to zero, it might not be advantageous for server j to start processing a fresh update, as the resulting reduction in age might be minimal. Conversely, having both servers process updates with nearly identical ages could be beneficial, as the effective processing time becomes the minimum of the two servers' processing times. However, this approach carries the drawback that the effort of one server may ultimately be discarded. When the state is (y_1, y_2, a) with $y_1 \gg y_2$, it might be optimal for server 1 to abandon the current update and start over with a fresh update.

However, new policies emerge when the service time X represents a two-stage computation and servers have additional knowledge about the computational stage of the other server. Consider two servers processing the same update, with server 1 completing the first stage of processing before server 2. In this case, it may be optimal for server 2 to abandon its processing and start fresh. Now

if server 2 with a fresher update reaches stage 2, while server 1 with an older update is still in stage 2, it may be optimal for server 1 to abandon its processing. These problem variations highlight the complexity and nuances of the parallel server model, and offer a range of opportunities for further research and optimization.

While we have identified a range of problems associated with optimizing age in two-step processing system, in Part IV of this work, specifically in Chapter 9, we primarily focus on identifying the optimal service rates for each processing step to achieve minimal system age performance, subject to a total power consumption constraint in both series and parallel server setups. Our analysis reveals that synchronous sequential execution generally outperforms asynchronous sequential execution. Additionally, we observe that parallel servers tend to outperform pipelines of servers (servers in series) in terms of AoI.

CHAPTER 2
PRELIMINARIES AND PRIOR WORK

This chapter provides an overview of AoI timeliness metric and common synchronization primitives widely found in modern systems. Section 2.1 introduces the age process and associated age metric, followed by general AoI evaluation methods that are applicable to a wide variety of systems. Sections 2.4 and 2.3 discuss various synchronization primitives broadly classified under two classes: lock-based and lock-free, and focus specifically on RWL and RCU as these will be our focus of study in subsequent chapters. Sections 2.5- 2.9 review prior work and also explain the relationship between the work presented in this thesis and the current state of research.

2.1 AoI Metric and Analysis

In a typical status updating system model illustrated in Fig. 2.1(a), a source generates updates with timestamps, which are then transmitted through a network to a destination monitor. An update carrying a timestamp u is characterized by its age $\Delta = t - u$, where $t \geq u$ denotes the current time. An update is considered fresh when its timestamp corresponds to the current time t and its age is zero. If we denote $u(t)$ as the timestamp of the most recent update received at the monitor by time t, then the age process at the monitor can be defined as the random process $\Delta(t) = t - u(t)$.

Fig. 2.1(b) depicts a sample path of age process $\Delta(t)$ at the monitor. Source submits fresh (age zero) updates to the network at times $t_1, t_2, \ldots$. These updates are delivered to the monitor at times $t'_1, t'_2, \ldots$. At each delivery time t'_i, at the monitor age is reset to age of the update i.e. $\Delta(t) = t'_i - t_i$. In absence of any update delivery, the age at the monitor increases at unit rate. Consequently we obtain a saw-tooth like waveform for the age process as depicted in Fig. 2.1(b). In the following subsections, we introduce two commonly used methods for AoI evaluation.

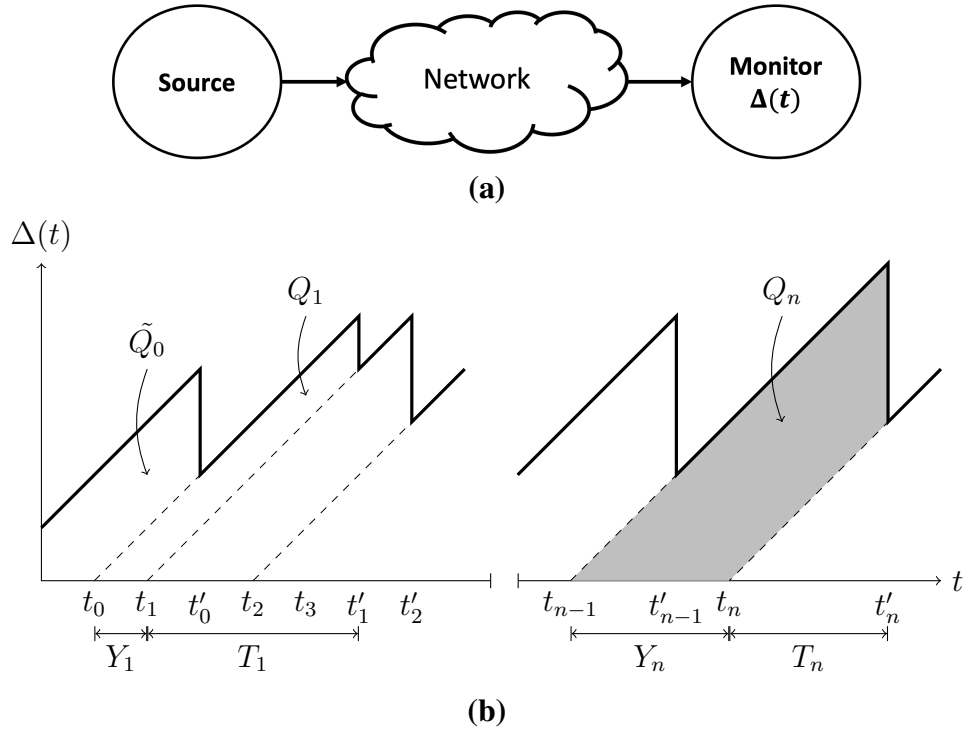


Figure 2.1: Sample path of age process $\Delta(t)$ at monitor. Y_n and T_n are inter-arrival and system times.

2.1.1 Time-Average Age

The time-average age is the area under graph in Fig. 2.1(b) normalized by time interval of observation. A stationary ergodic age process $\Delta(t)$ has average age (often referred to as AoI) [25]:

$$\mathrm{E}[\Delta] = \lim_{T\to\infty} \frac{1}{T} \int_0^T \Delta(t)dt. \tag{2.1}$$

We represent the area under sawtooth waveform as the concatenation of the polygon areas $\tilde{Q}_0, Q_1, \ldots, Q_n$. Let $Y_n = t_n - t_{n-1}$ and $T_n = t'_n - t_n$ denote the inter arrival time and system time of the update, then

$$Q_n = \frac{1}{2}(T_n + Y_n)^2 - \frac{1}{2}T_n^2 = Y_nT_n + \frac{1}{2}Y_n^2. \tag{2.2}$$

The time-average AoI $\mathrm{E}[\Delta]$ satisfies

$$\mathrm{E}[\Delta] = \frac{\mathrm{E}[Q_n]}{\mathrm{E}[Y_n]}. \tag{2.3}$$

2.1.2 Stochastic Hybrid Systems

For some of the system models presented in this thesis, we'll use a Stochastic Hybrid Systems (SHS) [26] approach, a technique introduced for AoI evaluation in [25] and since employed in AoI evaluation of a variety of status updating systems [27, 28, 29, 30, 31, 32, 33, 34]. 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 [25].

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.

$$\mathcal{L}_{\bar{q}} = \{l \in \mathcal{L} : q_l' = \bar{q}\}, \qquad \mathcal{L}'_{\bar{q}} = \{l \in \mathcal{L} : q_l = \bar{q}\}. \tag{2.4}$$

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

$$\mathbf{v}_q(t) = \mathrm{E}[\mathbf{x}(t)\delta_{q,q(t)}] = [\mathrm{E}[x_1(t)\delta_{q,q(t)}] \cdots \mathrm{E}[x_n(t)\delta_{q,q(t)}]], \tag{2.5}$$

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, 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. *[25, 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*

$$\bar{\mathbf{v}}_{\bar{q}} \sum_{l \in \mathcal{L}_{\bar{q}}} \lambda^{(l)} = \mathbf{1}\bar{\pi}_{\bar{q}} + \sum_{l \in \mathcal{L}'_{\bar{q}}} \lambda^{(l)} \bar{\mathbf{v}}_{q_l} \mathbf{A}_l, \quad \bar{q} \in \mathcal{Q}, \tag{2.6}$$

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

2.2 Using SHS: An Illustrative Example

To acquaint readers with the SHS analysis technique and to aid in easily understanding the work in this thesis, in the following, we describe a simple example of using SHS in calculating age in Last-Come-First-Serve (LCFS) queue with preemption in service that permits a new arrival to preempt an update in service. The system model is as follows: Updates arrive fresh (age zero) at a server as a Poisson process with rate λ. The service time at the server is exponential with rate μ. Consistent with the notation in AoI literature, we refer to this model as M/M/1*. Our goal is to evaluate the average age observed at the monitor, which receives updates from the server.

For the M/M/1* system, there are only two possible states: server being idle (state 0) and server being busy (state 1). Thus, the discrete state space is given by $\mathcal{Q} = \{0, 1\}$. There are three possible transitions in this system. First, an update arrival transitions the system from idle state to the busy state. Second, an update departure from the server transitions the system from busy state to idle state. And third, a self-transition occurs within the busy state when a new update arrival preempts the update currently in service. These transitions and the corresponding Markov Chain are illustrated in Fig. 2.2. The stationary distribution $\boldsymbol{\pi}$ of the Markov Chain is given by

$$\boldsymbol{\pi} = [\pi_0 \ \pi_1] = \Big[\frac{1}{1+\rho}, \frac{\rho}{1+\rho}\Big]. \tag{2.7}$$

We need to track ages at the server and the monitor. The continuous component of the SHS for this system is an age vector $\mathbf{x} = [x_0 \ x_1]$, where x_0 is the age at the server, and x_1 is the age at the monitor. When the system is in any discrete state, $\mathbf{x}$ increases at unit rate. At each discrete

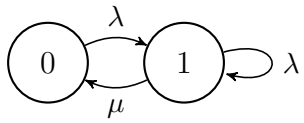


Figure 2.2: The SHS Markov chain for M/M/1*.

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$	$\mathbf{A}_l$
1	$0 \to 1$	λ	$[0, x_1]$	$\left[\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right]$
2	$1 \to 0$	μ	$[x_0, x_0]$	$\left[\begin{smallmatrix} 1 & 1 \\ 0 & 0 \end{smallmatrix}\right]$
3	$1 \to 1$	μ	$[0, x_1]$	$\left[\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right]$

Table 2.1: SHS transitions for M/M/1*.

state transition, the ages x_0 and x_1 may reset depending on the system's dynamics—whether an update enters, completes service, or is preempted. Table 2.1 outlines the SHS transitions and the corresponding reset mappings, which are described in detail below.

- $l = 1$: A fresh update arrives at an idle server, resetting the age at the server $x_0' = 0$. The age at the monitor remains unchanged, as no update has been delivered.
- $l = 2$: The server completes service, delivering the update to the monitor. The monitor receives an update with age x_0, hence $x_1' = x_0$. The server's age remains unchanged, so $x_0' = x_0$.
- $l = 3$: There is a fresh arrival at the server that preempts the update in service. The age at the server resets to $x_0' = 0$, while the age at the monitor remains unchanged, $x_1' = x_1$.

We now use Theorem 1 to solve for $\bar{\mathbf{v}} = [\bar{\mathbf{v}}_0\ \bar{\mathbf{v}}_1] = [v_{00}\ v_{01}\ v_{10}\ v_{11}]$. The next step in SHS analysis is writing down the balance equation for each $\bar{q} \in \mathcal{Q} = \{0, 1\}$ satisfying (2.6). Specifically, we use Table 2.1 to evaluate (2.6) at $\bar{q} = 0$ and $\bar{q} = 1$. We get

$$\lambda[v_{00}\ v_{01}] = [\pi_0\ \pi_0] + \mu[v_{10}\ v_{10}], \tag{2.8a}$$

$$(\lambda + \mu)[v_{10}\ v_{11}] = [\pi_1\ \pi_1] + \lambda[0\ v_{01}] + \lambda[0\ v_{11}]. \tag{2.8b}$$

Let $\rho = \lambda/\mu$. It follows from (2.7) and (2.8) that

$$v_{01} = \frac{1}{\mu\rho(1+\rho)}\Big(1 + \frac{\rho}{1+\rho}\Big), \quad \text{and} \tag{2.9a}$$

$$v_{11} = \frac{1}{\mu}\Big(1 + \frac{\rho}{(1+\rho)^2}\Big). \tag{2.9b}$$

Theorem 1 implies that the average age at the monitor is $\Delta_{\text{M/M/1}^*} = v_{01} + v_{11}$. Applying (2.9) yields

$$\Delta_{\text{M/M/1}^*} = \frac{1}{\mu}\Big(1 + \frac{1}{\rho}\Big). \tag{2.10}$$

We observe that the age expression in (2.10) is same as the independently derived expression in [35, Equation (48)] where the authors instead used sawtooth waveform based age analysis. The sawtooth waveform analysis of M/M/1* in [35] is pretty complex. On the contrary, we see that SHS simplifies the analysis significantly. The system dynamics can be effectively captured using a Markov Chain, demonstrating the tractability of the SHS approach in such scenarios.

2.3 Lock-based Synchronization Primitives

Lock-based synchronization mechanisms are built upon the concept of mutual exclusion, which ensures that only one thread can access a shared resource at any given time. Common examples include:

Mutual Exclusion (Mutex)

Mutex is perhaps the most common technique to synchronize the concurrent accesses to shared data. Mutex is used to ensure that only one thread can access a shared resource at a time. When a thread wants to access the shared resource, it must acquire the mutex lock. If the lock is held by another thread, the requesting thread will be blocked until the lock is released.

Semaphores

Semaphores serve as a generalized form of mutexes, facilitating multiple threads' access to shared resources. When employed to regulate access to a resource pool, they effectively monitor the availability of resources by maintaining a count or flag. Threads decrement this count upon entering the protected section and increment it upon exiting. If the count reaches zero, signaling full resource utilization, subsequent threads seeking entry may become blocked until the count increases again.

Spinlock

Unlike traditional locks, which may put a thread to sleep when the resource is unavailable, a spinlock causes the thread to repeatedly check if the lock is available. This checking is typically done in a tight loop, known as“spinning". Spinlocks are commonly used in scenarios where the expected wait time for a lock to become available is short. They are often implemented at the kernel level and are typically more efficient than traditional mutexes in situations where contention is low and the time spent waiting for the lock to become available is less than the overhead of putting the thread to sleep and waking it up.

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”[1] 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* [20]. 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.

2.4 Lock-free Synchronization Primitives

Lock-free algorithms do not use locks and are non-blocking [36, 37], meaning an algorithm is lock-free if it ensures that some thread always makes progress. Lock-free primitives rely on atomic operations provided by the hardware or supported by the programming language. Common atomic operations include compare-and-swap (CAS) [38], fetch-and-add (FAA) [39], and load-linked/store-conditional (LL/SC). These primitives allow for optimistic synchronization, which is effective when

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

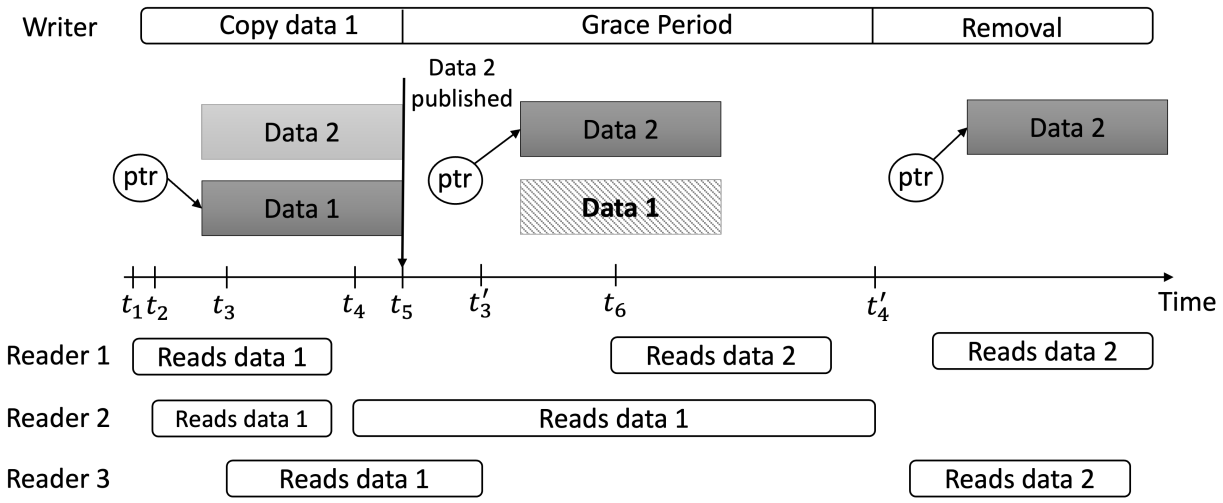


Figure 2.3: RCU working example.

synchronization conflicts are rare. For instance, CAS compares the current value of a variable with an expected value. If the current value matches the expected value, it updates the variable with a new value. CAS performs this operation atomically, meaning it ensures that no other thread has modified the variable between the read and the write phases. A widely used non-blocking algorithm for a shared data structure is Read-Copy-Update (RCU) with a single writer and any number of readers. Note that RCU with multiple writers is not necessarily lock-free as multiple writers generally serialize with a lock.

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, RCU is a synchronization primitive that allows concurrent forward progress for both writers and readers [40]. RCU can be broadly described in two steps [41] (also illustrated in Fig. 2.3):

1. **Publishing a new version:** When a writer needs to update a data item, it first creates a copy of the RCU-protected data item, modifies this copy with the new data, and then atomically

replaces the old reference with a reference to the new version. This update process occurs concurrently with ongoing read operations, allowing readers to continue accessing the old version using the old reference. However, any new read operations after the update will access the most recent version. For example, in Fig. 2.3, the writer publishes Data 2 (referenced by pointer *ptr*) at time t_5. Read requests before t_5 (i.e., at $t_i \leq t_5, i \in \{1, 2, 3, 4\}$) will access Data 1, while any read request after t_5 will access Data 2.

2. **Memory Reclamation:** Since some readers may still be referencing the old version of the data, the system defers reclaiming the memory used by the old data until all active readers have completed their read operations. For instance, in Fig. 2.3, Readers 2 and 3, which initiated their reads at t_4 and t_3 respectively, continue to access Data 1 even after the writer has published Data 2 at t_5. Although the pointer *ptr* references Data 2 from t_5 onwards, Data 1 remains in memory until all readers referencing it have finished their operations.

Therefore, at any given time, RCU can maintain multiple time-stamped versions of active data items, which are concurrently accessed by readers in the system. For instance, in Fig.2.3, there are two active copies of the data – Data 1 and Data 2 – during the time interval $[t_5, t'_4]$. Any reader that enters a read-side critical section before the writer's modification can complete its critical section without disruption. A "grace period" begins when the writer publishes the modified data item, ensuring that all RCU read-side critical sections existing at the start of the grace period are allowed to finish [42]. In Fig. 2.3, the grace period concludes at t'_4, when the last read request accessing Data 1 (in this case, Reader 2) completes. At the end of the grace period, it is safe to reclaim the memory and delete the outdated data. In Fig. 2.3, Data 1 is deleted after t'_4.

2.5 Related Work: Memory Systems and Freshness

The class of systems in which updates are stored in a memory system, and a client queries from memory bears similarities to distributed storage systems maintaining a database. Freshness in distributed systems has received attention in the literature [43, 44]. For instance, [44] addresses a

scenario where a remote real-world database is updated independently without pushing updates to the client, requiring the client to periodically poll the source to detect changes and refresh its copy.

The authors in [44] explore different synchronization policies to improve the freshness of a local copy of a real-world database, focusing on the web environment with the need to keep a web page index repository up-to-date. It's worth noting that our work differs from [44] in technical aspects, particularly in the definitions of age. While [44] assumes that the local copy has age zero if it is up-to-date with the real world and linearly increases thereafter, in our work, we assume that the age of both the real-world database and the local copy increases in the absence of any updates.

Another distinction lies in the behavior after synchronization. [44] assumes that upon synchronization with the real world, the local copy will convey the same information. However, in our work, we consider contention between readers and writers, where reading may yield an outdated update because the writer is in the process of writing a fresher one, potentially resulting in the copy at the client not being synchronized.

Leader-based data replication systems in distributed storage systems was explored in [45]. The study derives the average age of a read query and determines the optimal number of leaders that minimize the average age of retrieved data. [46] investigates a dynamo-style quorum-based replicated storage system and analyzes the trade-off between staleness and delay, which depends on the write quorum size. Our work differs from previous studies in that we design policies for the Reader to determine when it should read to minimize the average age at the client.

The contention between writes and reads to/from the memory system and the resulting trade-off between read latency and read query freshness has also been studied in context of scalabale database systems [47, 48, 49, 50]. For instance, [47] examines the contention between updating the database and servicing reads represented by transactions with deadlines. Prioritizing update writing may result in the system failing to meet transaction deadlines, while prioritizing transactions may lead to reading stale data if updates from the outside world were not written to the database.

A part of the work done in this thesis (particularly Chapter 5) bears resemblance to the work presented in [51], where the authors evaluate the performance of lock-based concurrency control

algorithms on maintaining the temporal consistency of data within database. The authors introduce the notion of age and age dispersion to define temporal consistency of data. The model examined in [51] involves sensor transactions that write time-stamped measurements (referred to as image objects) of real-world environments into a real-time database. An update transaction reads a set of image objects and writes back derived objects, the values of which are derived from the image objects. The sensor and update transactions are considered periodic, meaning that sensor transactions periodically record real-world measurements into the database, while update transactions perform periodic computations on these measurements.

However, our work diverges from [51] in several significant aspects. One major distinction is that we do not impose periodic writes and reads in our model. Furthermore, the concurrency control algorithms investigated in our work differ from those explored in [51]. Specifically, the lock-based concurrency construct examined in our research does not allow for the creation of multiple versions of objects. Instead, the real-world measurement update always occurs in place within the lock-based construct. While [51] served as an inspiration for our work, we argue that the results obtained from our research have broader applicability. Our findings are relevant not only to hardware shared memory systems but also to distributed database setups.

2.6 Related Work: Update On-demand

The model of Readers querying memory employs an update on-demand paradigm, where Readers can request memory updates at their discretion. The concept of timely on-demand update generation and reception has been studied under various names in AoI literature: generate-at-will, source sampling, pull-based communication, and in this thesis memory sampling. Among these, the overarching idea of generate-at-will and source sampling is that there is an entity, often referred to as a sampler, that determines when a source should sample the underlying physical process, generate a new packet, and send status packets to the destination. Several variations of this concept have been investigated, including those presented in [52, 53, 54, 55, 56, 57, 58, 59].

For instance, [52] examines a system where an energy-harvesting source submits updates to

the network for delivery to a monitor. It was demonstrated that, given knowledge of the service facility state, a zero-wait (just-in-time) policy, which submits an update as soon as the system becomes idle, does not always minimize AoI. Instead, a lazy update policy, where the source waits before submitting another update, often performs better. Building on this foundational work, [53] established sufficient and necessary conditions for the optimality of the zero-wait policy.

2.6.1 Memory Sampling: Is it a Variant of Generate-at-Will?

At first glance, memory sampling appears to be a version of the generate-at-will model, given the similarities between the two. Both involve sampling to obtain updates, which may already be stale. In generate-at-will, updates might become stale while in transit [57] or where the update generation time contributes to the aging process [58]. In many studies, various costs associated with generating updates have been considered. For instance, update generation can incur energy costs [54]. This situation is similar to memory sampling, where there are sampling costs involved.

However, a key distinction lies in the consumption pattern of updates. In traditional generate-at-will systems, updates are typically consumed once and not retrieved again. In contrast, shared memory introduces the uncertainty of potential multiple accesses to the same update, leading to possible redundant processing at the client. This difference sets memory sampling apart from its generate-at-will counterpart.

The impact of such a redundant processing in generate-at-will update systems has not been previously explored. However, we do observe the rich literature associated with classic model variations in on-demand update. For example, [58] considers an information update system where a receiver requests updates from a provider in order to minimize the age of information at the receiver. The authors account for non-negligible update generation times due to the complex processing tasks required to generate an update. By using distortion as a proxy for the quality of the update and assuming negligible communication time between the provider and receiver, they study age-optimal policies for determining the update request times at the receiver and the update processing times at the provider, subject to a maximum allowed distortion. The work by [57] investigates the optimal

online sampling strategy for the remote estimation of a Wiener process over a channel modeled as a queue. The study addresses the challenge of minimizing the estimation error while considering the inherent delay in the queue. Similarly, [54] addresses the scenario where generating an update incurs energy costs, in addition to the updating costs, which include both energy and delay costs for the packets carrying rich information. The authors propose a joint status sampling and updating process aimed at minimizing the AoI at the destination while adhering to an average energy cost constraint.

2.6.2 Memory Sampling: A Distinct Variant of Pull-Based Communication

Memory sampling shares similarities with pull-based communication systems, where updates are received on demand, such as in control applications responding to external triggers or user input. However, a key distinction lies in the relevance of update freshness. In pull-based communication, freshness matters only when the receiver queries the information, leading to the proposal of Age of Information at Query (QAoI) [60] and Effective AoI (EAoI) [61] metrics, which capture the varying importance of information freshness.

In contrast, memory sampling differs from these pull-based concepts. Despite employing a pull mechanism, the age of the update remains relevant at all times, both in the reader and the memory. Therefore, we adopt the classic AoI characterization, as first introduced in [3], to capture the time elapsed since the latest update at the receiver.

2.7 Related Work: Cache Updating Systems

Cache updating systems share similarities with shared memory system where the Reader maintains a local cache and can serve client requests using this local cache. The models studied in the context of cache updating share a common theme: a local cache (sometimes referred to as a local server) is connected to one or multiple remote server and maintains local copies of the data items from these server. The differences among these models arise from the constraints imposed on the system. For instance, some models consider a constraint on the total refresh rate [62], while others account for a

capacity-constrained link between the cache and the remote server. A commonality across these studies is that the cache maintains local copies from multiple sources.

In [63], the authors investigate how updated items should be downloaded from a remote server when the link between the local cache and the server is capacity-constrained, limiting the cache's ability to maintain the latest version of each item. They demonstrate that the update rate of each item depends on the square root of its popularity. Similarly, [62] explores a system where a local cache is connected to multiple remote sources, with data at each source being updated as a Poisson process with rate λ_i. The cache refreshes the local copy periodically, and the authors derive an age-minimal optimal policy that determines the refresh rate for each source, given the constraint on the total refresh rate.

We believe that the models we study are different. First, previous works on cache updating have assumed that the remote servers have unconstrained updates. With shared memory model, if the writes are lock protected, then this constrained memory updating needs to be studied and its impact on the age of local copy at the Reader.

2.8 Related Work: Multi-step Processing

Since we study multi-step update processing with respect to tandem queue and parallel queue system models, here we elucidate the existing studies on such network of queues, both in AoI literature as well as in general queuing theory literature.

In the Age-of-Information literature, various studies have focused on age in network of queues [64, 65, 27]. Yates [27] considered line network model of last-come-first served (LCFS) queue with preemption in service. It was shown that the i^{th} node contributes $1/\mu_i$ to the age at the monitor, where μ_i is the service rate at server i. Authors in [66] derived average age for two first-come-first served non-preemptive queues in tandem. The study by [67] models the communication and computation delay in edge computing framework and derives the PDF of Peak Age-of-Information (PAoI) for M/M/1-M/D/1 and M/M/1-M/M/1 tandem queues. In related research, [68] develops a recursive framework to derive the mean peak age of information for N heterogeneous servers in

tandem. Additionally, the work by [69] obtains the distribution of the age and peak age in a system of two tandem queues connected in series with packet prioritization in the second queue.

Age for M/M/2 and M/M/∞ systems was studied in [70] to demonstrate the advantage of having the message transmission path diversity for status updates. The research by [71] studies the age-delay trade-off in G/G/∞ queue. In a different context, [72] observes that a single M/M/1 queue has better age performance than the independent parallel M/M/1 queues with the same total capacity. Furthermore, [73] analyzed age in network of parallel finite identical and memoryless servers, where each server is an LCFS queue with preemption in service. However, our work deviates from [70, 73] in that we relax the assumption of memoryless processing times for updates. This key difference renders the SHS analysis used in [73] inapplicable to our scenario.

On the other hand, with respect to general queuing theory, the problem of optimal service rate control has been extensively studied across various types of queuing networks, ranging from single-queue single server model [74, 75, 76], multiple queue single server model [77], to multiple server, multiple queue model [78]. In studies focused on single-server queue systems, the general setting involves a nondecreasing cost of service and holding costs that are nondecreasing functions of queue length, with rewards associated with customers entering the queue. The arrival rate, λ, and/or the service rate, μ, are subject to control. The objective in these studies is typically to minimize the expected total discounted cost or the long-run average cost. In various systems, the authors establish optimality of monotone policies i.e. optimal arrival rates are non-increasing in number of arrivals and optimal service rates are non-decreasing in queue length as observed in [79].

Several authors have considered tandem queue systems with Poisson arrivals at rate λ and two memoryless servers, serving at rates μ_1 and μ_2 at first and second queue respectively. The first study on optimal service control in tandem queues was conducted by Rosberg et al. [80]. In this study, the authors examined a setting where the service rate at server 1 is selected as a function of the system's state, defined as the tuple of queue lengths at each server, while the service rate at server 2 is held constant. Considering only holding cost and no operating cost, the authors established the optimality of switchover policies, where the optimal rate at server 1 is determined by a switching

function of the queue length at server 2.

Authors in [79] considered a cyclic queue system where a number of ·/M/1 queues are arranged in a cycle. Considering a system cost comprising of both holding and operating costs, the authors determined the optimal policy has a transition-monotone decision rule, where when a customer moves from queue i to the following queue, the optimal service rate at queue i does not increase, and optimal service rate at queue $j, j \neq i$ does not decrease.

Optimal control of service rates of a tandem queue under power constraints is studied in [81]. The authors assume that the service rate is linear to the power allocated to that server and the sum of service rates must not exceed the given power budget. An iterative algorithm is proposed to find the optimal service rates.

2.9 Related Work: Synchronization primitives

A plethora of applications benefit from parallelization of various operations, including, for example, high throughput for transaction processing in distributed databases [82] and faster training times employing embarrassingly parallel processes in machine learning [83]. Such applications with high inter-processor communication demands expose synchronization between multiple processors as a key bottleneck in parallel computation. 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 [84]. 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[85].

The existing literature on synchronization techniques focuses mostly on the algorithm, imple-

mentation and throughput performance (operations per unit time) in the critical sections[2] [86, 37, 87]. For e.g., [86] presents a micro-benchmark suite to evaluate new data structures and synchronization techniques, aimed at helping programmers understand the cause of performance problems of their structures. However, there has been a lack of study on the impact of synchronization primitives on the timeliness of the data stored in shared memory in real-time IoT systems where obtaining fresh information is critical [88, 89].

RCU and RWL

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 [90], [91]. User-space RCU [92] is used in high-performance DNS servers [93], in the Linux networking toolkit [94], in distributed object storage systems [95]. Most recently, RCU protected data structures have been employed to ensure wait-free access to machine learning models by inference threads [96]. One drawback of a classical RCU mechanism is the *wait-for-readers* (using *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 [97], [98], read-log update [99]) that address the wait-for-readers problem. Apart from this, [100] introduced a real-time variant of RCU that allows preemption of read-side critical sections.

A limitation 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 [101, 99, 102]. Further, RCU implementation and verification is non-trivial and several attempts have been made to systematically check the RCU design and code [103, 104, 105, 106].

Readers-Writer locks (RWL) 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 [107]. Conventional implementation of Readers-Writer lock suffers from reader-reader scalability and different designs have been proposed for scalable Readers-Writer

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

locks [108, 109]. Authors in [110] present the design of a family of RW locks to leverage NUMA features and deliver better performance.

Part II

Optimizing Memory Access

CHAPTER 3
EFFICIENT AND TIMELY MEMORY ACCESS - KNOWN MEMORY STATE

3.1 Introduction

The primary question in this chapter is when should the reader sample the memory. Typically, there is a cost associated with memory sampling, and this cost structure varies between systems. In systems with substantial object sizes, retrieving and locally copying objects incurs a high cost, while querying for timestamps remains relatively inexpensive. In contrast, there are systems where memory contains smaller objects, and the cost of retrieval is comparable to the cost of a timestamp query. These are systems where queries are sent to a distant database, with the cost being the latency associated with the query.

In this work, we focus on former class of systems where the Reader knows the freshness of an object in the memory by virtue of inexpensive timestamp retrievals. However, due to longer read times, denoted by high sampling costs, the Reader must decide if sampling is justified compared to age reduction obtained after sampling.

We note that the concept of timely memory sampling, wherein the reader incurs a cost for sampling for age reduction, shares similarities with research focused on managing access for multiple users within a communication channel. Various studies in the AoI literature have explored Whittle's index-based transmission scheduling algorithms [111, 112, 113, 114, 115, 116, 117, 118, 119, 120], where the scheduling problem is decomposed into multiple independent subproblems. Within each subproblem, an additional cost (C) is associated with updating the user.

However, there is a conceptual difference in the cost associated with the decoupled problem and this study. In a Whittle index policy, the minimum cost that makes both actions — updating a user or idling — equally desirable is used as a mechanism to choose one of the many users. In this work, we enforce an explicit cost of accessing the memory, and we study the trade-offs observed with age

and memory access by varying this system parameter. However, it is not a mechanism designed to distinguish between users.

3.1.1 Contributions and Chapter Outline

This chapter investigates the relation between sampling costs and Age-of-Information. In section 3.2, we formulate our problem as a Markov Decision Process (MDP) with the goal of minimizing average cost comprising age at the client and the cost incurred due to sampling. In section 3.3, we establish that an optimal policy of the MDP is a stationary and deterministic threshold-type policy. We then derive optimal threshold and the optimal average cost by exploiting the structure of optimal policy. Finally, section 3.4 presents numerical evaluation on average cost against system parameters.

3.2 System Model

In this work, we focus on a class of systems (see Fig. 1.1) where a Writer writes the time-varying data received from the source into the memory, and a Reader samples the memory on behalf of a client. We consider a discrete-time slotted system with slots labelled $t = 0, 1, 2, \ldots$. The system involves two key processes: writing the time-varying data from the source into the memory and reading the source data from memory. The modeling details of these processes are discussed below.

3.2.1 Writing Source Updates to Memory

We assume the Writer commits/writes fresh (age zero) source updates to memory at the end of each slot with probability p, independent from slot to slot. These source updates generate the age process $x(t)$ in the memory.

In practice, the write time will be non-negligible. However, our focus in this work is not on systems where writing to the memory is the bottleneck process. Instead, our primary interest lies in examining the delays associated with reading and processing of source updates. Note that in the event that these writes do require time $\tau > 0$, $x(t)$ and the update age process at the client will be shifted by τ.

3.2.2 Sampling Source Updates from Memory

At each time slot, the Reader determines whether to access the memory and read a source update. The update in memory is read over a period of a slot, and the reader gets the data at the end of the slot. Notably, this model aligns with the Read-Copy-Update (RCU) memory access paradigm, where a new update can be written in slot t while the Reader is in the process of reading the current update in the same slot. The Reader generates an age process $y(t)$ at the input to the client that is a sampled version of source update age process $x(t)$ in the memory. Hence we say the Reader *samples* the updates in the memory.

The state-dependent action $a(t)$ selected by the Reader at time slot t determines whether the Reader remains idle ($a(t) = 0$) or performs a read operation ($a(t) = 1$). We consider a scenario where a non-negative fixed cost c is associated with reading the memory during each time slot. Ideally, the Reader aims to minimize $y(t)$, which means it would prefer to read in every slot to stay close to the age process $x(t)$. However, this comes at the cost of paying the sampling cost c. If the Reader samples too frequently, it might end up with the same update, resulting in no age reduction but incurring a penalty for sampling. On the contrary, if it reads too infrequently, the age at the client input increases.

In this work, we assume that the Reader is notified when an update is published in the memory, enabling the Reader to know the update age in the memory. Based on the system state, the Reader implements a scheduling scheme that minimizes the average cost $\mathrm{E}[y(t) + ca(t)]$. To find an optimal scheduling policy, we model our problem as a Markov Decision Process (MDP).

3.2.3 Markov Decision Process Formulation

In the context of our MDP model, denoted with $\mathcal{M}$ from here on, the following four components make up the structure:

- States: We denote the set of possible system states by S which does not vary with time. State $s(t) \in S$ is a tuple $(x(t), y(t))$, where at the start of a time slot, $x(t) \in \{0, 1, 2, \ldots\}$ is the age

of the update in the memory, and $y(t) \in \{1, 2, 3, \ldots\}$ is the age of sampled source updates at the client. Notice that S is a countably infinite set since age is unbounded.

- Action: Let $a(t) \in \{0, 1\}$ denote the action taken in slot t indicating Reader's decision, where $a(t) = 1$ if Reader decides to read and $a(t) = 0$ if idle.

- Transition Probabilities: Letting $\bar{p} = 1 - p$, when $a(t) = 1$, the transition probability from state $s = (x, y)$ to state $s' \in S$ is

$$\mathrm{P}[s' \mid s = (x, y), a = 1] = \begin{cases} p & s' = (0, x + 1), \\ \bar{p} & s' = (x + 1, x + 1). \end{cases} \tag{3.1a}$$

  And when $a(t) = 0$, the transition probability is

$$\mathrm{P}[s' \mid s = (x, y), a = 0] = \begin{cases} p & s' = (0, y + 1), \\ \bar{p} & s' = (x + 1, y + 1). \end{cases} \tag{3.1b}$$

- Cost: The cost $C(s(t); a(t))$ incurred in state $s(t)$ in time slot t under action $a(t)$ is defined as:

$$C(s(t) = (x, y);\ a(t) = a) \coloneqq y + ca. \tag{3.2}$$

Let $\pi : S \to A$ denote a policy that for each state $s(t) \in S$ specifies an action $a(t) = \pi(s(t)) \in A$ at slot t. The expected average cost under policy π starting from a given initial state at $t = 0$, $s(0) = (x, y)$, is defined as:

$$g_\pi(x, y) = \limsup_{T \to \infty} \frac{1}{T} \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} (y(t) + ca(t)) \mid s(0) = (x, y) \right]. \tag{3.3}$$

We say that policy π^* is average-cost optimal if $g_{\pi^*}(s) = \inf g_\pi(s)$ for every $s \in S$. We focus on the case where for some constant g, $g_{\pi^*}(s) = g$ for all $s \in S$. Thus, the problem is to obtain π^* such that $g = g_{\pi^*}(s) = \inf g_\pi(s)$ for every $s \in S$.

Our cost minimization problem falls within the category of average cost minimization problems. Given that the age can grow unbounded, both the number of states and the cost in each stage are countably infinite. For such MDPs, the existence of an optimal policy, whether stationary or non-stationary, is not guaranteed [121, Chap 5]. Notably, even the existence of an optimal stationary policy may not hold, while an optimal non-stationary policy might exist [122].

Analyzing average cost problems with an infinite state space poses inherent difficulties. However, under certain conditions and structures, it is possible to develop useful results. Proving the existence of an optimal average cost stationary policy is not an immediate goal in this chapter and we defer this discussion to later in Section 3.5. There, we draw upon results from [123], which provides conditions ensuring the existence of an expected average cost optimal stationary policy. We verify that these conditions hold for our problem. In the subsequent section, we derive results regarding the structure of the optimal policy under the assumption that the optimal policy exists and the relative cost Bellman's equation is valid.

3.3 Characterization of Cost Optimality

3.3.1 Discounted Cost

We begin by introducing the α-discounted version of the problem. Recall that the state for MDP $\mathcal{M}$ is a tuple $s = (x, y)$, and $a \in \{0, 1\}$. Then using (3.1), the discounted cost Bellman's optimality equation for $\mathcal{M}$ is given by

$$\begin{aligned} V(x,y) = \min\{&y + \alpha\left(pV(0,y+1) + \bar{p}V(x+1,y+1)\right), \\ &y + c + \alpha(pV(0,x+1) + \bar{p}V(x+1,x+1))\}. \end{aligned} \tag{3.4}$$

Here, the first term of $\min$ corresponds to the reader staying idle ($a = 0$), and the second term corresponds to the reader sampling ($a = 1$). The action that is a minimizer of (3.4) is referred to as the α-*optimal action* and the resulting policy π^*_α is referred to as the α-*optimal policy*.

We define the value iteration $V_n(s)$ by $V_0(s) = 0, \forall s \in S$, and, for any $n > 0$,

$$V_{n+1}(x,y) = \min\{y + \alpha\left(pV_n(0,y+1) + \bar{p}V_n(x+1,y+1)\right), \\ y + c + \alpha(pV_n(0,x+1) + \bar{p}V_n(x+1,x+1))\}. \tag{3.5}$$

For non-negative costs, it is evident that $V_n(s) \leq V_{n+1}(s)$. It then follows from [121, Theorem 4.2, Chapter III] that

$$\lim_{n\to\infty} V_n(s) = V(s), \qquad s \in S. \tag{3.6}$$

We now state properties of the value function $V(x,y)$.

Proposition 1. *(Monotonicity): The value function $V(x,y)$ is non-decreasing in both x and y.*

The proof, using mathematical induction on (3.5), is straightforward and is omitted.

Proposition 2. *If the α-optimal action is to sample in (x,y), then the α-optimal action is to sample in every (x,y') with $y' \geq y$.*

Proof of this proposition is provided in the Appendix 3.B. Another version of this proposition asserts that if the α-optimal action is to sample in state (x,y) at stage n, then it is also optimal to sample in every (x,y') with $y' \geq y$ at stage n. The proof employing the value iteration (3.5) is omitted as it is similar to that of Proposition 2.

Proposition 3. *(Concavity): For a fixed x, $V(x,y+1) - V(x,y)$ is non-increasing in y.*

The proof appears in the Appendix 3.C. The intuitive structure of the optimal policy is that with knowledge of the age in the memory, the Reader should refrain from sampling if the reduction in age doesn't justify the sampling cost. To further characterize this intuition, we introduce the following proposition. The proof appears in the Appendix 3.D.

Proposition 4. *If the α-optimal action in state (x,y) is to idle, then the α-optimal action in states $(x+i, y+i), \forall i \geq 1$ is to stay idle.*

Specifically, when the memory is freshly updated, the Reader must assess whether sampling is worthwhile. If it opts against sampling initially, it should consistently abstain from sampling in subsequent slots until the memory undergoes another update, as the age reduction remains constant in the absence of changes. In terms of the MDP $\mathcal{M}$, this concept translates to making a decision in the state $(0, y)$. If the optimal decision is not to sample at this point, then the Reader should consistently refrain from sampling in states $(1, y+1)$, $(2, y+2)$, and so on.

3.3.2 Average Cost Optimality

Since the conditions of Theorem 3 (in section 3.5) hold, the cost-optimal policy π^* is the limit point of α-optimal policies π^*_α with $\alpha \to 1$ [123, Lemma]. Therefore, Propositions 2 and 4 are sufficient to provide the structure of average cost optimal policy. Specifically, Propositions 2 and 4 imply that there exists a threshold Y_0 such that it is optimal to sample in $(0, y)$ for every $y \geq Y_0$ and idle otherwise.

At this point, it is important to mention the set of feasible states under π^*. With $Y_0 = 1$, the optimal policy dictates sampling in every state $(0, y)$ where $y \geq 1$. Upon sampling in $(0, 1)$, the system transitions to feasible states, specifically $\{(0, 1), (1, 1)\}$. In state $(1, 1)$, a close examination of Bellman's equation (3.4) reveals that it is optimal to idle. Therefore, the set of possible states when choosing to idle in $(1, 1)$ becomes $\{(0, 2), (2, 2)\}$. Subsequent transitions follow a pattern where sampling in $(0, y)$ leads to states $\{(0, 1), (1, 1)\}$, and choosing to idle in states (i, i) with $i \in \mathbb{N}$ resulting in $\{(0, i+1), (i+1, i+1)\}$.

In scenarios where $Y_0 > 1$, optimality dictates idling in $(0, y)$ with $y < Y_0$, prompting the system to transition to states $\{(0, y+1), (1, y+1)\}$. The subsequent action in $(0, y+1)$ hinges upon whether $y + 1 < Y_0$. If $y + 1 \geq Y_0$, the system resets, transitioning to either $(0, 1)$ or $(1, 1)$; conversely, if $y + 1 < Y_0$, the system perpetuates a structure akin to that observed in state $(0, y)$. Conversely, if the system transitions to $(1, y+1)$, idling in $(1, y+1)$ is optimal. The resulting permissible states from this point include $\{(0, y+2), (2, y+2)\}$, and this pattern repeats. We summarize this set of feasible states for the optimal policy in the following proposition.

Proposition 5. *For MDP $\mathcal{M}$, under the optimal policy π^* with threshold Y_0, the set of feasible states is*

$$S^* = \{(0, y) \mid y \in \mathbb{N}\} \cup \{(x, y) \mid x \geq 1 \text{ and } y - x < Y_0\}. \tag{3.7}$$

To determine the optimal threshold for an optimal policy π^*, we employ the relative cost Bellman's equation

$$\begin{aligned} g + f(x, y) = \min\big\{ & y + pf(0, y+1) + \bar{p}f(x+1, y+1), \\ & y + c + pf(0, x+1) + \bar{p}f(x+1, x+1)\big\}. \end{aligned} \tag{3.8}$$

Here, g denotes the optimal average cost, and $f(x, y)$ represents the relative cost-to-go function. Our objective is to identify relative cost-to-go function $f(x, y)$ for $(x, y) \in S^*$, facilitating the determination of the optimal threshold and, consequently, the optimal average cost.

Proposition 6. *Defining $(0, 1)$ as the reference state with $f(0, 1) = 0$, the relative cost functions satisify:*

(i) $f(0, Y_0 + 1) - f(0, Y_0) = 1$.

(ii) For any $x \geq 0$,

$$f(x, Y_0 - 1) = \frac{1}{p}(J_0 + \frac{\bar{p}}{p}) - 1, \tag{3.9}$$

where $J_0 = Y_0 - g + pf(0, Y_0)$.

(iii) For every $y < Y_0$, $f(0, y) = f(1, y) \ldots = f(y, y)$.

(iv) When $Y_0 > 1$, $f(0, Y_0) = Y_0 - g + c$.

The proof appears in the Appendix 3.E. We now use Proposition 6 to derive the optimal threshold.

Lemma 1. *As a function of the threshold Y_0, the average cost is*

$$g_0(Y_0) = \frac{1}{2}\left(\frac{1}{p} + Y_0 + \frac{2cp + \bar{p}/p}{pY_0 + \bar{p}}\right). \tag{3.10}$$

The proof appears in the Appendix 3.F.

Theorem 2. *The optimal threshold Y_0^* associated with optimal policy π^* for MDP $\mathcal{M}$ is $Y_0^* = \lceil Y' \rceil$ where*

$$Y' = \sqrt{2c + (1/p - 1/2)^2} - (1/p - 1/2). \tag{3.11}$$

Proof. It follows from (3.10) and some algebra that

$$g_0(Y_0) - g_0(Y_0 + 1) = \frac{-p^2}{2}\left[\frac{Y_0^2 + (2/p - 1)Y_0 - 2c}{(pY_0 + \bar{p})(pY_0 + 1)}\right] \tag{3.12}$$

We define $Q(Y_0) \equiv Y_0^2 + Y_0\,(2/p - 1) - 2c$ and we observe that Y' in (3.11) is the only positive root of $Q(y)$. Further $Q(Y_0) > 0$ for $Y_0 > Y'$. It then follows from (3.12) that $g_0(\lfloor Y' \rfloor) \geq g_0(\lceil Y' \rceil)$ and that $g_0(\lceil Y' \rceil), g_0(\lceil Y' \rceil + 1), \ldots$ is a non-decreasing sequence. □

Theorem 2 provides an explicit expression for the optimal threshold Y_0^*. However, evaluating the optimal average cost $g = g_0(Y_0^*)$ using (3.10) doesn't directly show the relationship between system parameters c and p. The following lemma provides a close approximation to the optimal average cost and captures the impact of these key system parameters on the average cost.

Lemma 2. *The optimal average cost satisfies*

$$g \geq 1/2 + \sqrt{2c + 1/p^2 - 1/p}. \tag{3.13}$$

Proof. Note that

$$g = \min_{Y_0 \in \mathbb{N}} g_0(Y_0) \geq \min_{y \in \mathbb{R}^+} g_0(y). \tag{3.14}$$

To minimize $g_0(y)$ over positive reals, we set $dg_0(y)/dy = 0$, yielding

$$y = \tilde{Y}_0^* = -\bar{p}/p + \sqrt{2c + \bar{p}/p^2}. \tag{3.15}$$

This yields $g \geq g_0(\tilde{Y}_0^*)$, which is the lower bound (3.13). □

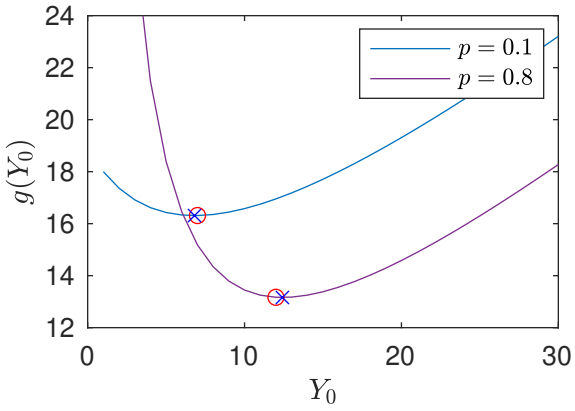


Figure 3.1: Plot of average cost $g_0(Y_0)$ as a function of threshold Y_0 with sampling cost $c = 80$. Here, $\circ$ is the true optimal cost $g_0(Y_0^*)$, and $\times$ is the approximate optimal average cost $g_0(\tilde{Y}_0^*)$.

3.4 Numerical Evaluation

Figure 3.1 shows how the average cost $g_0(Y_0)$, given by (3.10), changes with threshold Y_0. Initially, as Y_0 increases, the average cost decreases. This is because a low threshold leads to excessive sampling, incurring costs without much age reduction, resulting in a higher average cost. As Y_0 increases further, the cost of sampling approaches the gain in age reduction. However, setting Y_0 too high delays memory access, increasing client age and consequently the average cost. Fig. 3.1 highlights the existence of an optimal threshold where the cost of sampling justifies the age reduction.

Fig. 3.2 illustrates the value of optimal threshold Y_0^* as a function of probability p of source update publication in a slot. We observe that the optimal threshold increases with p. When the Reader is required to make a decision in a given slot, it assesses both the age at the client and the age in the memory. These evaluations contribute to determining the potential age reduction vs the cost of sampling. In scenarios where the client's update is deemed sufficiently recent, the Reader may choose to skip sampling. This decision is influenced by a higher probability (p) of obtaining a more recent update soon, that will perhaps be worth sampling.

Figure 3.3 compares the optimal average cost g with the lower bound provided in (3.13). The tightness of the lower bound is evident, as it closely aligns with the curve of the optimal average cost. Additionally, the figure illustrates that the optimal cost tends to increase with an increase

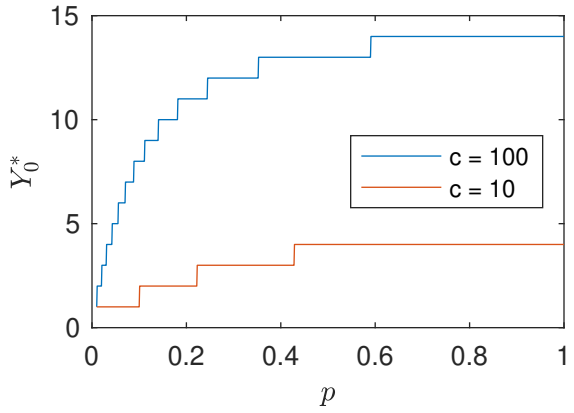


Figure 3.2: Plot of optimal threshold Y_0^* as a function of probability p of source update publication in a slot, with a fixed sampling cost c.

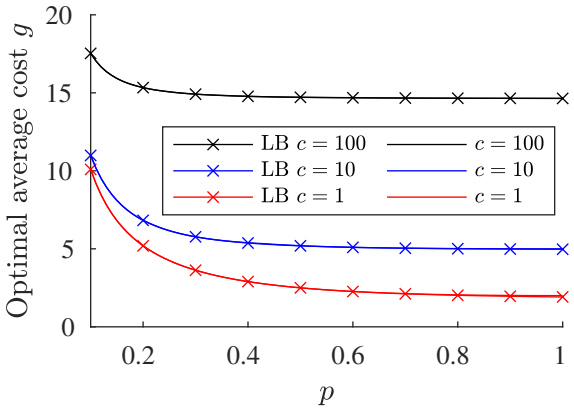


Figure 3.3: Comparison of optimal average cost g and the corresponding lower bound (LB) as a function of probability p of source update publication in a slot, with a fixed sampling cost c.

in the sampling cost c. This suggests that while designing the cost structure, the cost should be sufficiently high but not excessively so. Furthermore, the plot shows that the average cost decreases as the probability p of memory updates in a slot increases. This is intuitive, as frequent memory updates increase the likelihood of the Reader receiving a fresh update when it samples, thereby reducing the age at the client.

3.5 Stationary Average Cost Optimal Policy

In this section we verify that the average cost optimality equation for MDP holds for $\mathcal{M}$. To get started, we need the following result.

Lemma 3. *Under the deterministic stationary policy θ of reading in every slot, the system exhibits an irreducible, ergodic Markov Chain, with expected cost $M(x,y)$ of first passage from state $s=(x,y)$ to $(0,1)$ satisfying*

$$M(x,y) \leq \frac{1+p}{p^2}(c+y) + \frac{3}{2p^3}. \tag{3.16}$$

Proof of Lemma 3 appears in the Appendix 3.A. We now employ the lemma in verifying the conditions of the following theorem.

Theorem 3. *[123, Theorem] If the following conditions hold for MDP $\mathcal{M}$:*

1. *For every state s and discount factor α, the quantity $V(s)$ is finite,*
2. *$f_\alpha(s) \coloneqq V(s) - V(0)$ satisfies $-N \overset{(a)}{\leq} f_\alpha(s) \overset{(b)}{\leq} M(s)$, where $M(s) \geq 0$, and*
3. *For all s and a, $\sum_{s'} \mathbb{P}_{s,s'}(a) M(s') < \infty$,*

then there exists a stationary policy that is average cost optimal for MDP $\mathcal{M}$. Moreover, for $\mathcal{M}$, there exists a constant $g = \lim_{\alpha \to 1}(1-\alpha)V(s)$ for every state s, and a function $f(s)$ with $-N \leq f(s) \leq M(s)$ that solve relative-cost Bellman's equation,

$$g + f(s) = \min_a \{C(s;a) + \sum_{s' \in S} \mathbb{P}_{s,s'}(a) f(s')\}. \tag{3.17}$$

For MDP $\mathcal{M}$, we choose reference state 0 as $(0,1)$. A sufficient condition for 1 and $2(b)$ to hold is the existence of a single stationary policy that induces an irreducible, ergodic Markov Chain, with the associated expected cost of first passage from any state (x,y) to state $(0,1)$ being finite ([123, Propositions 4 and 5]). Lemma 3 verifies that this sufficient condition is met for our problem. A sufficient condition for $2(a)$ is that $V(s)$ is non-decreasing in s [123]. Proposition 1 demonstrates that this sufficient condition is also met.

Now, condition 3 of Theorem 3 asserts that under any a, the quantity $\sum_{s'} \mathbb{P}_{s,s'}(a) M(s')$ should

be finite. For MDP $\mathcal{M}$, from (3.1b), when $a = 0$, we have for any state $s = (x, y)$,

$$\sum_{s'} \mathbb{P}_{s,s'}(0)M(s') = pM(0, y + 1) + \bar{p}M(x + 1, y + 1). \tag{3.18}$$

From (3.1a), when $a = 1$, we similarly have for any state $s = (x, y)$,

$$\sum_{s'} \mathbb{P}_{s,s'}(1)M(s') = pM(0, x + 1) + \bar{p}M(x + 1, x + 1). \tag{3.19}$$

It follows from (3.18), (3.19) and Lemma 3 that condition 3 holds for MDP $\mathcal{M}$. Therefore, there exists a constant $g = \lim_{\alpha\to 1}(1 - \alpha)V(x, y)$ for every state (x, y) that is an optimal average cost and a relative cost to go function $f(x, y)$ with $0 \leq f(x, y) \leq M(x, y)$.

3.6 Conclusion

This chapter focused on a class of systems where source updates are disseminated using shared memory. The Writer process writes these source updates in the memory, and a Reader fulfills clients' requests for these measurements by reading from the memory. We studied the problem of optimizing memory access by the Reader with respect to minimizing average cost. Our main contributions included establishing the existence of an optimal stationary deterministic policy for our Markov Decision Process (MDP). Furthermore, we demonstrated that the optimal policy has a threshold structure.

A key insight from our analysis was that the Reader should choose to sample only when the memory undergoes an update. If the Reader decides not to sample, this decision of staying idle should perpetuate in subsequent slots until the memory is updated with a fresh source update. This is because, in the absence of updates, there is no change in age reduction; it remains the same as when the memory was last updated.

APPENDIX

3.A Proof of Lemma 3

Lemma 3. The expected first passage cost $M(x, y)$ to go from state (x, y) to state $(0, 1)$ under the optimal policy satisfies

$$M(x, y) \leq \frac{1+p}{p^2}(c + y) + \frac{3}{2p^3}. \tag{3.20}$$

Proof. Note that

$$M(x, y) \leq \mathrm{E}[\hat{C}(x, y)], \tag{3.21}$$

where $\hat{C}(x, y)$ is the first passage cost under the policy in which the Reader samples in every slot. Starting from state (x, y) under the "always sample" policy, there is a geometric (p) number N of slots in which the system passes from states (x, y) up through $(x + N - 1, y + N - 1)$ until a memory update takes the system to state $(0, x + N)$ In the next slot, a cost $c + x + N$ is incurred and the system goes to either state $(0, 1)$ with probability p or, with probability $1 - p$, to $(1, 1)$. In the latter case, the additional cost $\hat{C}(1, 1)$ is incurred to reach $(0, 1)$. We define the Bernoulli $(1 - p)$ random variable $\overline{Z}$ such that $\overline{Z} = 1$ if a memory update does *not* occur in state $(0, x + N)$. The cost expended to go from (x, y) to $(0, 1)$ is then

$$\begin{aligned}\hat{C}(x, y) &= \sum_{j=y}^{y+N-1}(c + j) + (c + x + N) + \overline{Z}\hat{C}(1, 1) \\ &= N(c + y) + (c + x) + \frac{N(N + 1)}{2} + \overline{Z}\hat{C}(1, 1).\end{aligned} \tag{3.22}$$

Taking expectation,

$$\mathrm{E}[\hat{C}(x, y)] = \frac{c + y}{p} + (c + x) + \frac{3 - p}{2p^2} + \bar{p}\,\mathrm{E}[\hat{C}(1, 1)]. \tag{3.23}$$

Evaluating (3.23) at $(x, y) = (1, 1)$ yields

$$\mathrm{E}[\hat{C}(1,1)] = \frac{1}{p}\left[\left(\frac{1}{p}+1\right)(c+1) + \frac{3-p}{2p^2}\right]. \tag{3.24}$$

Combining (3.23) and (3.24) yields

$$\mathrm{E}[\hat{C}(x,y)] = \frac{c+y}{p} + (c+x) + \frac{1-p^2}{p^2}(c+1) + \frac{3-p}{2p^3}. \tag{3.25}$$

Since $x \leq y$ and $1 \leq y$ for any feasible state (x, y), we obtain

$$\begin{aligned}\mathrm{E}[\hat{C}(x,y)] &\leq \left(\frac{1}{p}+1\right)(c+y) + \frac{1-p^2}{p^2}(c+y) + \frac{3-p}{2p^3} \\ &\leq \frac{1+p}{p^2}(c+y) + \frac{3}{2p^3}.\end{aligned} \tag{3.26}$$

The claim then follows from (3.21). □

3.B Proof of Proposition 2

Proposition 2. If the α-optimal action is to sample in (x, y), then the α-optimal action is to sample in every (x, y') with $y' \geq y$.

Proof. For brevity, we'll use the following shorthand notation in the proof. For $w \leq v$, we define

$$\tilde{J}(u, v, w) = V(u, v) - V(u, w). \tag{3.27}$$

The monotonicity of the value function (Proposition 1) implies

$$\tilde{J}(u, v_1, w) \leq \tilde{J}(u, v_2, w) \quad \text{for all } u, w, \text{ and } v_1 \leq v_2. \tag{3.28}$$

For the rest of our discussion, we use the following form of discounted-cost Bellman's optimality

equation with $c_\alpha = c/\alpha$:

$$V(x,y) = y + \alpha \min\big\{pV(0,y+1) + \bar{p}V(x+1,y+1), \\ c_\alpha + pV(0,x+1) + \bar{p}V(x+1,x+1)\big\}. \tag{3.29}$$

Let $\hat{x} = x+1$ and $\hat{y} = y+1$. According to (3.29), the condition for the Reader to sample in (x,y) is

$$pV(0,\hat{y}) + \bar{p}V(\hat{x},\hat{y}) \geq c_\alpha + pV(0,\hat{x}) + \bar{p}V(\hat{x},\hat{x}). \tag{3.30}$$

Using the shorthand $\tilde{J}(u,v,w)$, the inequality (3.30) becomes

$$p\tilde{J}(0,\hat{y},\hat{x}) + \bar{p}\tilde{J}(\hat{x},\hat{y},\hat{x}) \geq c_\alpha. \tag{3.31}$$

Given that condition (3.31) holds, we examine the state $(x,\hat{y})$. The value function for this state is

$$V(x,\hat{y}) = \hat{y} + \alpha \min\{pV(0,\hat{y}+1) + \bar{p}V(\hat{x},\hat{y}+1), c_\alpha + pV(0,\hat{x}) + \bar{p}V(\hat{x},\hat{x})\}. \tag{3.32}$$

The condition for the Reader to sample in $(x,\hat{y})$ is

$$pV(0,\hat{y}+1) + \bar{p}V(\hat{x},\hat{y}+1) \geq c_\alpha + pV(0,\hat{x}) + \bar{p}V(\hat{x},\hat{x}), \tag{3.33}$$

or equivalently,

$$p\tilde{J}(0,\hat{y}+1,\hat{x}) + \bar{p}\tilde{J}(\hat{x},\hat{y}+1,\hat{x}) \geq c_\alpha. \tag{3.34}$$

Now we observe from the monotonicity property (3.28) and (3.31) that

$$p\tilde{J}(0,\hat{y}+1,\hat{x}) + \bar{p}\tilde{J}(\hat{x},\hat{y}+1,\hat{x}) \geq p\tilde{J}(0,\hat{y},\hat{x}) + \bar{p}\tilde{J}(\hat{x},\hat{y},\hat{x}) \\ \geq c_\alpha. \tag{3.35}$$

Thus (3.34) holds, confirming that the Reader samples in state $(x,\hat{y})$. □

3.C Proof of Proposition 3

Proposition 3. (Concavity): For a fixed x, $V(x, y+1) - V(x, y)$ is non-increasing in y.

Proof. We want to show that for a fixed x, $V_n(x, i+1) - V_n(x, i) \geq V_n(x, i+2) - V_n(x, i+1)$, for every $i \in \mathbb{N}$. To achieve this, we focus on demonstrating the inequality:

$$V_n(x, i+2) + V_n(x, i) \leq 2V_n(x, i+1) \qquad \forall\, n, i. \tag{3.36}$$

The base case for $n = 1$ is trivially satisfied, as $V_1(x, y) = y$. Now suppose (3.36) holds for $n = 1, 2 \ldots k$ for every i. We will establish the validity of (3.36) under two scenarios, corresponding to the α-optimal action at stage $k+1$ being either to sample or idle in state $(x, i+1)$. First, let's consider the case where it is optimal to sample in $(x, i+1)$ at stage $k+1$. This implies that the value iteration function in this state satisfies:

$$V_{k+1}(x, i+1) = i + 1 + c + \alpha p V_k(0, x+1) + \alpha \bar{p} V_k(x+1, x+1). \tag{3.37}$$

Furthermore, leveraging Proposition 2, we deduce that sampling in $(x, i+1)$ is also the optimal action for state $(x, i+2)$ at stage $k+1$, resulting in:

$$V_{k+1}(x, i+2) = i + 2 + c + \alpha p V_k(0, x+1) + \alpha \bar{p} V_k(x+1, x+1). \tag{3.38}$$

Notice that the value iteration function for (x, i) satisifies

$$V_{k+1}(x, i) \leq i + c + \alpha \left(p V_k(0, x+1) + \bar{p} V_k(x+1, x+1) \right). \tag{3.39}$$

Combining (3.37), (3.38), and (3.39), we establish:

$$V_{k+1}(x, i+2) + V_{k+1}(x, i) \leq 2V_{k+1}(x, i+1). \tag{3.40}$$

Let us now consider the situation where the α-optimal action is to stay idle in state $(x, i+1)$ at stage $k+1$. This implies that

$$V_{k+1}(x, i+1) = i + 1 + \alpha \left(pV_k(0, i+2) + \bar{p}V_k(x+1, i+2) \right). \tag{3.41}$$

Leveraging Proposition 2, we conclude that staying idle in $(x, i+1)$ is also the optimal action for state (x, i) at stage $k+1$, leading to:

$$V_{k+1}(x, i) = i + \alpha \left(pV_k(0, i+1) + \bar{p}V_k(x+1, i+1) \right). \tag{3.42}$$

The value iteration function for $(x, i+2)$ satisfies

$$V_{k+1}(x, i+2) \leq i + 2 + \alpha \left(pV_k(0, i+3) + \bar{p}V_k(x+1, i+3) \right). \tag{3.43}$$

Combining (3.42) and (3.43), we can demonstrate:

$$\begin{aligned} V_{k+1}(x, i+2) + V_{k+1}(x, i) &\leq 2(i+1) + \alpha \Big(p\big(V_k(0, i+3) + V_k(0, i+1)\big) \\ &\qquad + \bar{p}\big(V_k(x+1, i+3) + V_k(x+1, i+1)\big) \Big), \\ &\overset{(a)}{\leq} 2(i+1) + \alpha \Big(2pV_k(0, i+2) + 2\bar{p}V_k(x+1, i+2) \Big), \\ &= 2\Big[i + 1 + \alpha \Big(pV_k(0, i+2) + \bar{p}V_k(x+1, i+2) \Big) \Big], \end{aligned} \tag{3.44}$$

where (a) follows from induction hypothesis that $V_k(x, i+3) + V_k(x, i+1) \leq 2V_k(x, i+2)$. It then follows from (3.41) and (3.44) that

$$V_{k+1}(x, i+2) + V_{k+1}(x, i) = 2V_{k+1}(x, i+1). \tag{3.45}$$

It follows from principle of mathematical induction that (3.36) holds for every n, and hence $V_n(x, y)$ is concave in y. As $\lim_{n\to\infty} V_n(x, y) = V(x, y)$, this implies that $V(x, y)$ is concave in y. $\square$

3.D Proof of Proposition 4

Proposition 4. If the α-optimal action in state (x, y) is to idle, then the α-optimal action in states $(x + i, y + i), \forall i \geq 1$ is to stay idle.

Proof. For brevity, we'll use the following shorthand notation in the proof. For $w \leq v$, let

$$\tilde{J}_n(u, v, w) = V_n(u, v) - V_n(u, w). \tag{3.46}$$

We establish key properties of $\tilde{J}_n(u, v, w)$ to be utilized later in the proof.

1. Given $w \leq v$, Proposition 1 implies $V_n(u, v) \geq V_n(u, w) \geq 0$, and hence

$$\tilde{J}_n(u, v, w) \geq 0. \tag{3.47}$$

2. If $v_2 \geq v_1$, and $w_2 \geq w_1$, it follows from concavity property (Proposition 3) that

$$V_n(u, v_2) - V_n(u, w_2) \leq V_n(u, v_1) - V_n(u, w_1) \tag{3.48}$$

and as a consequence,

$$\tilde{J}_n(u, v_2, w_2) \leq \tilde{J}_n(u, v_1, w_1), \quad \forall u, \text{with } w_1 \leq w_2, \text{and } v_1 \leq v_2. \tag{3.49}$$

3. Let $\hat{u} = u + 1$, $\hat{v} = v + 1$ and $\hat{w} = w + 1$. Under the condition of not sampling in (u, v), it can be shown that

$$\tilde{J}_n(u, v, w) = v - w + \alpha\big(p\tilde{J}_{n-1}(0, \hat{v}, \hat{w}) + \bar{p}\tilde{J}_{n-1}(\hat{u}, \hat{v}, \hat{w})\big). \tag{3.50}$$

We now resume the proof of proposition. Letting $\hat{x} = x + 1$ and $\hat{y} = y + 1$ and $c_\alpha = c/\alpha$, we

re-write the value iteration in state (x, y) given by (3.5) as:

$$V_{n+1}(x, y) = y + \alpha \min\{pV_n(0, \hat{y}) + \bar{p}V_n(\hat{x}, \hat{y}), c_\alpha + pV_n(0, \hat{x}) + \bar{p}V_n(\hat{x}, \hat{x})\}, \tag{3.51}$$

Given that Reader doesn't sample in (x, y) implies that for all n, the terms inside the $\min$ function in (3.51) satisfy:

$$pV_n(0, \hat{y}) + \bar{p}V_n(\hat{x}, \hat{y}) \leq c_\alpha + pV_n(0, \hat{x}) + \bar{p}V_n(\hat{x}, \hat{x}). \tag{3.52}$$

Expressing inequality (3.52) in terms of $\tilde{J}_n(u, v, w)$, we get:

$$p\tilde{J}_n(0, \hat{y}, \hat{x}) + \bar{p}\tilde{J}_n(\hat{x}, \hat{y}, \hat{x}) \leq c_\alpha. \tag{3.53}$$

Given that (3.53) holds for every n, we examine state $(x + i, y + i)$. The value iteration expression at stage $n + 1$ is given by:

$$\begin{aligned} V_{n+1}(x + i, y + i) = y + i + \alpha \min\big\{&pV_n(0, \hat{y} + i) + \bar{p}V_n(\hat{x} + i, \hat{y} + i), \\ &c_\alpha + pV_n(0, \hat{x} + i) + \bar{p}V_n(\hat{x} + i, \hat{x} + i)\big\}. \end{aligned} \tag{3.54}$$

To establish that the optimal action in state (x, y) being to stay idle implies the same for states $(x + i, y + i)$, we aim to show that the terms inside the $\min$ function in (3.54) satisfy:

$$pV_n(0, \hat{y} + i) + \bar{p}V_n(\hat{x} + i, \hat{y} + i) \leq c_\alpha + pV_n(0, \hat{x} + i) + \bar{p}V_n(\hat{x} + i, \hat{x} + i). \tag{3.55}$$

or equivalently,

$$p\tilde{J}_n(0, \hat{y} + i, \hat{x} + i) + \bar{p}\tilde{J}_n(\hat{x} + i, \hat{y} + i, \hat{x} + i) \leq c_\alpha. \tag{3.56}$$

Given that (3.53) holds for all n, proving that (3.56) holds for all n is equivalent to showing that the LHS of (3.56) is less than LHS of (3.53). For that it is sufficient to show for all $n \geq 1$

$$I_1(n + 1) = \tilde{J}_n(0, \hat{y} + i, \hat{x} + i) - \tilde{J}_n(0, \hat{y}, \hat{x}) \leq 0, \tag{3.57}$$

and

$$I_2(n+1) = \tilde{J}_n(\hat{x}+i, \hat{y}+i, \hat{x}+i) - \tilde{J}_n(\hat{x}, \hat{y}, \hat{x}) \leq 0. \tag{3.58}$$

With $i \geq 1$, it is clear that $\hat{y}+i \geq \hat{y}$ and $\hat{x}+i \geq \hat{x}$. Leveraging (3.49) , we conclude that $\tilde{J}_n(0, \hat{y}+i, \hat{x}+i) \leq \tilde{J}_n(0, \hat{y}, \hat{x})$, leading to $I_1 \leq 0$ for every n. We use inductive arguments to show that $I_2(n+1) \leq 0$. When $n = 1$, we see that

$$\tilde{J}_1(u, v, w) = V_1(u, v) - V_1(u, w) = v - w. \tag{3.59}$$

This means that

$$I_2(2) = \tilde{J}_1(\hat{x}+i, \hat{y}+i, \hat{x}+i) - \tilde{J}_1(\hat{x}, \hat{y}, \hat{x}) = 0,$$

and hence the base case holds. Now assume that $I_2(n+1) \leq 0$ for $n = 1, \ldots k-1$ for all $i \geq 0$. This implies:

$$I_2(k) = \tilde{J}_{k-1}(\hat{x}+i, \hat{y}+i, \hat{x}+i) - \tilde{J}_{k-1}(\hat{x}, \hat{y}, \hat{x}) \leq 0, \quad \forall i \geq 0. \tag{3.60}$$

We have established that $I_1(k) \leq 0$, implying that (3.57) holds at $n = k-1$. Combining this with (3.60), we conclude that both I_1 and I_2 hold at $n = k-1$. This, in turn, implies that (3.56) holds at $n = k-1$. Consequently, (3.55) holds at $n = k-1$. Therefore, the assumption $I_2(k) \leq 0$ for all $i \geq 0$ implies that the action that minimizes (3.54) at stage k is to stay idle in state $(x+i, y+i)$ for all $i \geq 1$.

Now, we need to demonstrate that:

$$I_2(k+1) = \tilde{J}_k(\hat{x}+i, \hat{y}+i, \hat{x}+i) - \tilde{J}_k(\hat{x}, \hat{y}, \hat{x}) \leq 0, \quad \forall i \geq 0. \tag{3.61}$$

Given that the assumption is to not sample in $(\hat{x}+i, \hat{y}+i)$ for $i \geq 0$ at stage k, this means that it is optimal to not sample in $(\hat{x}, \hat{y})$ (Proposition 2). Hence, employing Property (3) of $\tilde{J}_n(u, v, w)$, from

(3.50), we have with $\hat{i} = i + 1$,

$$\tilde{J}_k(\hat{x}+i, \hat{y}+i, \hat{x}+i) = \hat{y} - \hat{x} + \alpha(p\tilde{J}_{k-1}(0, \hat{y}+\hat{i}, \hat{x}+\hat{i}) + \bar{p}\tilde{J}_{k-1}(\hat{x}+\hat{i}, \hat{y}+\hat{i}, \hat{x}+\hat{i})). \quad (3.62)$$

Similarly, we have

$$\tilde{J}_k(\hat{x}, \hat{y}, \hat{x}) = \hat{y} - \hat{x} + \alpha(p\tilde{J}_{k-1}(0, \hat{y}+1, \hat{x}+1) + \bar{p}\tilde{J}_{k-1}(\hat{x}+1, \hat{y}+1, \hat{x}+1)). \quad (3.63)$$

From (3.49), we can state that:

$$\tilde{J}_{k-1}(0, \hat{y}+\hat{i}, \hat{x}+\hat{i}) \leq \tilde{J}_{k-1}(0, \hat{y}+1, \hat{x}+1). \quad (3.64)$$

Additionally, it follows from (3.60),

$$\tilde{J}_{k-1}(\hat{x}+\hat{i}, \hat{y}+\hat{i}, \hat{x}+\hat{i}) \leq \tilde{J}_{k-1}(\hat{x}+1, \hat{y}+1, \hat{x}+1). \quad (3.65)$$

Based on (3.64) and (3.65), we observe that

$$\tilde{J}_k(\hat{x}+i, \hat{y}+i, \hat{x}+i) - \tilde{J}_k(\hat{x}, \hat{y}, \hat{x}) = I_2(k+1) \leq 0. \quad (3.66)$$

Thus, by induction, we establish that $I_2(n+1) \leq 0$ holds for all $n \geq 1$. □

3.E Proof of Proposition 6

Proposition 6. Defining $(0, 1)$ as the reference state with $f(0, 1) = 0$, the relative cost functions satisify:

(i)

$$f(0, Y_0 + 1) - f(0, Y_0) = 1. \quad (3.67)$$

Proof. Given that it is optimal to sample in $(0, Y_0)$, the relative-cost Bellman's equation in

state $(0, Y_0)$ is given as

$$g + f(0, Y_0) = Y_0 + c + pf(0, 1) + \bar{p}f(1, 1). \tag{3.68}$$

The optimal action in $(0, Y_0 + 1)$ is also to sample (Proposition 2), and therefore, the relative-cost Bellman's equation in state $(0, Y_0 + 1)$ becomes

$$g + f(0, Y_0 + 1) = Y_0 + 1 + c + pf(0, 1) + \bar{p}f(1, 1). \tag{3.69}$$

It follows from (3.68) and (3.69) that $f(0, Y_0 + 1) - f(0, Y_0) = 1$. □

(ii) For any $x \geq 0$,

$$f(x, Y_0 - 1) = \frac{1}{p}(J_0 + \frac{\bar{p}}{p}) - 1, \tag{3.70}$$

where $J_0 = Y_0 - g + pf(0, Y_0)$.

Proof. For any $x \geq 0$, the optimal action in $(x, Y_0 - 1)$ is to idle, and the Bellman's equation (3.8) becomes

$$f(x, Y_0 - 1) = -g + Y_0 - 1 + pf(0, Y_0) + \bar{p}f(x + 1, Y_0). \tag{3.71}$$

Let $J_0 = -g + Y_0 + pf(0, Y_0)$, we obtain

$$f(x, Y_0 - 1) = J_0 - 1 + \bar{p}f(x + 1, Y_0). \tag{3.72}$$

Since the optimal action in $(x, Y_0 - 1)$ is to idle, then from Proposition 4, the optimal action in $x \geq 0$, $(x + 1, Y_0)$, is to idle as well. The Bellman's equation (3.8) in $(x + 1, Y_0)$ becomes

$$\begin{aligned} &f(x + 1, Y_0) \\ &= -g + Y_0 + pf(0, Y_0 + 1) + \bar{p}f(x + 2, Y_0 + 1), \end{aligned}$$

$$\stackrel{(a)}{=} -g + Y_0 + p(1 + f(0, Y_0)) + \bar{p} f(x + 2, Y_0 + 1),$$
$$= J_0 + p + \bar{p} f(x + 2, Y_0 + 1), \tag{3.73}$$

where (a) follows from Proposition 6(i). Substituting (3.73) into (3.72), we obtain

$$f(x, Y_0 - 1) = J_0(1 + \bar{p}) - 1 + p\bar{p} + \bar{p}^2 f(x + 2, Y_0 + 1). \tag{3.74}$$

Repeating this procedure n times yields

$$f(x, Y_0 - 1) = J_0 \sum_{i=0}^{n} \bar{p}^i + p\bar{p} \sum_{i=1}^{n-1} (i + 1)\bar{p}^i + \bar{p}^2 \sum_{i=0}^{n-2} (i + 1)\bar{p}^i$$
$$+ \bar{p}^{n+1} f(x + n + 1, Y_0 + n) - 1, \tag{3.75}$$

and in the limit $n \to \infty$ we have

$$f(0, Y_0 - 1) = \frac{J_0}{1 - \bar{p}} + \frac{p\bar{p}}{(1 - \bar{p})^2} + \frac{\bar{p}^2}{(1 - \bar{p})^2} - 1 = \frac{1}{p}(J_0 + \frac{\bar{p}}{p}) - 1. \tag{3.76}$$

Here, $\bar{p}^{n+1} f(x + n + 1, Y_0 + n) \to 0$ when $n \to \infty$ as $f(x + n + 1, Y_0 + n)$ is bounded. This bounding property is derived from Theorem 3, where it is established that $f(x + n + 1, Y_0 + n) \leq M(x + n + 1, Y_0 + n)$. Then it follows from (3.16),

$$f(x + n + 1, Y_0 + n) \leq \frac{1 + p}{p^2}(c + Y_0 + n) + \frac{3}{2p^3}. \tag{3.77}$$

□

(iii) For every $y < Y_0$,

$$f(0, y) = f(1, y) = \cdots = f(y, y). \tag{3.78}$$

Proof. From the threshold structure of the optimal policy, the optimal action in $(x, Y_0 - 2)$ is

to stay idle, and the relative-cost Bellman's equation (3.8) becomes

$$\begin{aligned} & f(0, Y_0 - 2) \\ & = -g + Y_0 - 2 + pf(0, Y_0 - 1) + \bar{p}f(1, Y_0 - 1), \\ & \overset{(a)}{=} -g + Y_0 - 2 + pf(0, Y_0 - 1) + \bar{p}f(0, Y_0 - 1), \\ & = -g + Y_0 - 2 + f(0, Y_0 - 1), \end{aligned} \tag{3.79}$$

where (a) follows from Proposition 6(ii) as $f(x, Y_0 - 1)$ is independent of x. This fact along with (3.79) implies that $f(x, Y_0 - 2)$ is also independent of x, and so $f(0, Y_0 - 2) = f(1, Y_0 - 2) \dots f(Y_0 - 2, Y_0 - 2)$. In fact this can be generalized such that $(x, Y_0 - k)$ with $x \geq 0$ and $k \in \{1, 2, \dots, Y_0 - 1\}$ is independent of x. □

(iv) When $Y_0 > 1$,

$$f(0, Y_0) = Y_0 - g + c. \tag{3.80}$$

Proof. At $(0, Y_0)$ the Reader samples and the Bellman's equation (3.8) becomes

$$f(0, Y_0) = Y_0 - g + c + pf(0, 1) + \bar{p}f(1, 1). \tag{3.81}$$

When $Y_0 > 1$, we have from Proposition 6(iii), $f(0, 1) = f(1, 1)$, and since $f(0, 1) = 0$, it follows that

$$f(0, Y_0) = Y_0 - g + c. \tag{3.82}$$

□

3.F Proof of Lemma 1

Lemma 1. The average cost as a function of the threshold Y_0 is given by:

$$g_0(Y_0) = \frac{1}{2}\left(\frac{1}{p} + Y_0 + \frac{2cp + \bar{p}/p}{pY_0 + \bar{p}}\right). \tag{3.83}$$

Proof. We break down the proof into three parts. In the first and second parts, we derive analytical expressions for the optimal average cost when $Y_0 = 1$ and $Y_0 = 2$, respectively. In the third part, we focus on obtaining a general expression for the optimal average cost when $Y_0 > 2$. Surprisingly, we discover that the average cost equation as a function of Y_0 obtained in the third part is a general equation for any $Y_0 \geq 1$.

(*Part* 1): If $Y_0 = 1$, it is optimal to sample in $(0, 1)$. Thus Bellman's equation (3.8) yields

$$f(0,1) = -g + 1 + c + pf(0,1) + \bar{p}f(1,1). \tag{3.84}$$

Defining $(0, 1)$ as the reference state with $f(0, 1) = 0$ yields $0 = -g + 1 + c + \bar{p}f(1, 1)$, or equivalently,

$$f(1,1) = \frac{g - 1 - c}{\bar{p}}. \tag{3.85}$$

Now it is optimal to never sample in $f(1, 1)$. Then

$$\begin{aligned} f(1,1) &= -g + 1 + pf(0,2) + \bar{p}f(2,2), \\ &\overset{(a)}{=} g + 1 + p(1 + f(0,1)) + \bar{p}f(2,2), \\ &= -g + 1 + p + \bar{p}f(2,2), \end{aligned} \tag{3.86}$$

where (a) follows from Proposition 6(i). Since staying idle is the optimal action in $(1, 1)$, then Proposition 4 implies that staying idle is also the optimal action in state $(2, 2)$, and hence

$$\begin{aligned} f(2,2) &= -g + 2 + pf(0,3) + \bar{p}f(3,3), \\ &= -g + 2 + p(2 + f(0,1)) + \bar{p}f(3,3), \\ &= -g + 2 + 2p + \bar{p}f(3,3). \end{aligned} \tag{3.87}$$

Substituting $f(2,2)$ obtained in (3.87) in (3.86), we obtain

$$f(1,1) = -g(1+\bar{p}) + 1 + p + 2\bar{p} + 2p\bar{p} + \bar{p}^2 f(3,3). \tag{3.88}$$

Similarly, we can obtain $f(3,3)$ as

$$f(3,3) = -g + 3 + 3p + \bar{p} f(4,4). \tag{3.89}$$

Again substituting $f(3,3)$ obtained in (3.89) in (3.88), we obtain

$$f(1,1) = -g(1+\bar{p}+\bar{p}^2) + 1 + p + 2\bar{p} + 2p\bar{p} + 3\bar{p}^2 + 3p\bar{p}^2 + \bar{p}^3 f(4,4). \tag{3.90}$$

Repeating this n times, we obtain

$$\begin{aligned} f(1,1) = &-g(1+\bar{p}+\bar{p}^2 \ldots + \bar{p}^n) + (1 + 2\bar{p} + 3\bar{p}^2 + \ldots + (n+1)\bar{p}^n) \\ &+ p(1 + 2\bar{p} + 3\bar{p}^2 + \ldots + (n+1)\bar{p}^n) + \bar{p}^{n+1} f(n+2, n+2), \end{aligned} \tag{3.91}$$

and letting $n \to \infty$, we obtain

$$\begin{aligned} f(1,1) &= \frac{-g}{1-\bar{p}} + \frac{1}{(1-\bar{p})^2} + \frac{p}{(1-\bar{p})^2}, \\ &= \frac{-g}{p} + \frac{1}{p^2} + \frac{1}{p}, \\ &= \frac{-g+1}{p} + \frac{1}{p^2}. \end{aligned} \tag{3.92}$$

Here, $\bar{p}^{n+1} f(n+2, n+2) \to 0$ when $n \to \infty$ as $f(n+2, n+2)$ is bounded. This bounding property is derived from Theorem 3, where it is established that $f(n+2, n+2) \leq M(n+2, n+2)$. Subsequently, (3.16) implies that

$$f(n+2, n+2) \leq \frac{1+p}{p^2}(c+n+2) + \frac{3}{2p^3}. \tag{3.93}$$

Now equating $f(1,1)$ in (3.85) and (3.92), we obtain

$$g = \frac{1}{p} + cp. \tag{3.94}$$

(*Part* 2): If $Y_0 = 2$, then it is optimal to sample in $(0,2)$. The realtive-cost Bellman's equation (3.8) is

$$f(0,2) = -g + 2 + c + pf(0,1) + \bar{p}f(1,1) \overset{(a)}{=} -g + 2 + c, \tag{3.95}$$

where (a) follows from Proposition 6(iii) which for $Y_0 = 2$ implies that $f(1,1) = f(0,1) = 0$. Now from (3.9), we have for $x = 1$, and $Y_0 = 2$,

$$f(0,1) = \frac{1}{p}\left(J_0 + \frac{\bar{p}}{p}\right) - 1 = \frac{1}{p}\left(-g + 2 + pf(0,2) + \frac{\bar{p}}{p}\right) - 1. \tag{3.96}$$

With $f(0,2)$ given by (3.95), we have

$$f(0,1) = \frac{1}{p}\left(-g + 2 + p(-g + 2 + c) + \frac{\bar{p}}{p}\right) - 1. \tag{3.97}$$

Equating $f(0,1) = 0$ gives

$$\begin{aligned} g &= \frac{1}{1+p}\left(2 + 2p + cp + \frac{\bar{p}}{p} - p\right), \\ &= \frac{1}{1+p}\left(1 + 1 + p + cp + \frac{\bar{p}}{p}\right), \\ &= \frac{1}{1+p}\left(1 + \frac{1}{p} + p(c+1)\right), \\ &= \frac{1}{p} + \frac{(c+1)p}{1+p}. \end{aligned} \tag{3.98}$$

(*Part* 3): Now consider the case when $Y_0 > 2$. Since it is optimal to not sample in state $(0, Y_0 - 2)$,

the Bellman's equation for state $(0, Y_0 - 2)$ becomes

$$\begin{aligned} f(0, Y_0 - 2) &= Y_0 - 2 - g + pf(0, Y_0 - 1) + \bar{p}f(1, Y_0 - 1), \\ &\overset{(a)}{=} Y_0 - 2 - g + f(0, Y_0 - 1), \end{aligned} \tag{3.99}$$

where (a) follows from Proposition 6(iii). Moreover,

$$\begin{aligned} f(0, Y_0 - 3) &= Y_0 - 3 - g + pf(0, Y_0 - 2) + \bar{p}f(0, Y_0 - 2), \\ &= Y_0 - 3 - g + f(0, Y_0 - 2), \\ &= 2(Y_0 - g) - (2 + 3) + f(0, Y_0 - 1). \end{aligned} \tag{3.100}$$

Repeating this procedure k times yields,

$$\begin{aligned} f(0, Y_0 - k) &= (k - 1)(Y_0 - g) - (2 + 3 + 4 + \cdots + k) + f(0, Y_0 - 1), \\ &= (k - 1)(Y_0 - g) - \frac{k(k + 1)}{2} + 1 + f(0, Y_0 - 1). \end{aligned} \tag{3.101}$$

Recalling $(0, 1)$ as the reference state with $f(0, 1) = 0$, evaluating (3.101) at $k = Y_0 - 1$ yields

$$(Y_0 - 2)(Y_0 - g) = \frac{(Y_0 - 1)Y_0}{2} + 1 + f(0, Y_0 - 1), \tag{3.102}$$

From Proposition 6,

$$f(0, Y_0 - 1) = (1 + 1/p)(Y_0 - g) + c + \bar{p}/p^2 - 1. \tag{3.103}$$

Thus it follows from (3.102) that

$$(Y_0 - g)\left(\frac{1}{p} + Y_0 - 1\right) = \frac{(Y_0 - 1)Y_0}{2} - \frac{\bar{p}}{p^2} - c. \tag{3.104}$$

Rearranging to solve for g yields

$$\begin{aligned} g &= Y_0 - \frac{1}{Y_0 - 1 + 1/p}\left[\frac{Y_0(Y_0-1)}{2} - c - \frac{\bar{p}}{p^2}\right] \\ &= \frac{Y_0}{2} + \frac{\frac{Y_0(Y_0-1+1/p)}{2} - \left[\frac{Y_0(Y_0-1)}{2} - c - \frac{\bar{p}}{p^2}\right]}{Y_0 - 1 + 1/p} \\ &= \frac{Y_0}{2} + \frac{\frac{Y_0}{2p} + c + \frac{\bar{p}}{p^2}}{Y_0 - 1 + 1/p}. \end{aligned} \tag{3.105}$$

Recalling $-1 + 1/p = \bar{p}/p$, we obtain

$$g = \frac{Y_0}{2} + \frac{1}{2p} + \frac{1}{2}\frac{2cp + \bar{p}/p}{pY_0 + \bar{p}}. \tag{3.106}$$

Since (3.106) depends upon Y_0, we express it as

$$g_0(Y_0) = \frac{1}{2}\left(\frac{1}{p} + Y_0 + \frac{2cp + \bar{p}/p}{pY_0 + \bar{p}}\right). \tag{3.107}$$

Finally observe that even though (3.107) was derived for $Y_0 > 2$, we see that $g_0(1) = 1/p + cp$, where the RHS is same as RHS of (3.94), the average cost obtained separately at $Y_0 = 1$. Similarly, $g_0(2) = 1/p + (c+1)p/(1+p)$, where the RHS is same as RHS of (3.98), the average cost obtained separately at $Y_0 = 2$.

□

CHAPTER 4
EFFICIENT AND TIMELY MEMORY ACCESS - UNKNOWN MEMORY STATE

4.1 Introduction

While the study in Chapter 3 provided novel insights, the model examined doesn't necessarily reflect the system architecture in practical applications. In practice, especially in scalable distributed systems that are built upon producer-consumer paradigm, pull queries from consumers are sent to a remote destination. Given that the cost of timestamp retrievals (which is high due to latency) is almost comparable to the cost of actual data item retrievals, the consumer (Reader) adopts a polling mechanism without being aware of the update freshness. However, such systems, while scalable, introduce inefficiencies such as resource wastage (e.g., network bandwidth) due to regular polling. In practice, producer-consumer systems usually employ mechanisms such as "long polling" [124] to curtail empty or stale responses. Similar to Chapter 3, we abstract such polling regulatory mechanisms by introducing a non-negative cost associated with memory sampling. With such sampling cost and the memory state unawareness at the Reader, the objective in this chapter remains the same as in Chapter 3: to study the trade-off associated with memory sampling and the age at client input.

4.1.1 Contributions and Chapter Outline

Section 4.2 describes the discrete time system model for timely memory sampling where memory state is unknown at the Reader and formulate such a problem as a Markov Decision Process (MDP). In section 4.5, we verify the existence of average cost optimal stationary policy for this system. Section 4.3 presents our main contribution where we develop and study three heuristic scheduling algorithms for the Reader. We start with the most obvious policy of reading in every slot (Always Sample policy), then introduce Probabilistic Reading (PR) policy where the Reader samples with

some non zero probability in any slot. Finally, we analyze Fixed Wait (FW) policy where the Reader waits for fixed number of slots before it samples. In Section 4.4, we present numerical evaluation of the performance of heuristic policies with respect to the corresponding average costs.

4.2 System Model

Similar to Chapter 3, in this chapter as well, we focus on a class of systems (see Fig. 1.1) where a Writer writes the time-varying data received from the source into the memory, and a Reader samples the memory on behalf of a client. We consider a discrete-time slotted system with slots labelled $t = 0, 1, 2, \ldots$. The source update publication generates age process $x(t)$ at the memory. The state dependent action $a(t)$, where the Reader either idles ($a(t) = 0$), or samples ($a(t) = 1$) influences the evolution of age process $y(t)$ at client input as follows:

$$y(t+1) = \begin{cases} x(t) + 1 & \text{if } a(t) = 1, \\ y(t) + 1 & \text{if } a(t) = 0. \end{cases} \tag{4.1}$$

If $c \geq 0$ denotes the memory sampling cost, our objective is to find an optimal read schedule that minimizes average cost $\mathrm{E}[y(t) + ca(t)]$ across slots. As in Chapter 3, we adopt a Markov Decision Process formulation to determine the optimal policy.

4.2.1 Markov Decision Process Formulation

In the context of our MDP model, denoted with $\mathcal{M}'$ from here on, the following four components make up the structure[1]:

- States: State $s(t)$ is a tuple $(y(t), h(t))$, where $y(t)$ is the age of sampled source update at the client input and $h(t)$ is the number of slots elapsed since the last read. We denote the set of possible system states by S which does not vary with time. Notice that S is a countably infinite set since age is unbounded.

[1]Observe that MDP $\mathcal{M}'$ for this system where the Reader samples without update age information is different from MDP $\mathcal{M}$ in Chapter 3.

- Action: Let $a(t) \in \{0, 1\}$ denote the action taken in slot t indicating Reader's decision, where $a(t) = 1$ if Reader decides to read and $a(t) = 0$ if idle.

- Transition Probabilities: Letting $\bar{p} = 1 - p$, when $a(t) = 1$, the transition probability from state $s = (y, h)$ to state $s' \in S$ is

$$\mathrm{P}[s' \mid s = (y, h), a = 1] = \begin{cases} p & s' = (1, 1), \\ p\bar{p} & s' = (2, 1), \\ p\bar{p}^2 & s' = (3, 1), \\ \vdots & \\ p\bar{p}^{h-1} & s' = (h, 1), \\ \bar{p}^h & s' = (y + 1, 1). \end{cases} \tag{4.2a}$$

  And when $a(t) = 0$, the transition probability is:

$$\mathrm{P}[s' = (y + 1, h + 1) \mid s = (y, h), a = 0] = 1. \tag{4.2b}$$

- Cost: The cost $C(s(t); a(t))$ incurred in state $s(t)$ in time slot t under action $a(t)$ is defined as:

$$C(s(t) = (y, h);\ a(t) = a) \coloneqq y + ca. \tag{4.3}$$

The expected average cost under policy π starting from a given initial state at $t = 0$, $s(0) = (y, h)$, is defined as:

$$g_\pi(y, h) = \limsup_{T \to \infty} \frac{1}{T} \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} (y(t) + ca(t)) \mid s(0) = (y, h) \right]. \tag{4.4}$$

The problem is to obtain π^* such that $g = g_{\pi^*}(s) = \inf g_\pi(s)$ for every $s \in S$. As in Chapter 3, the average cost optimization problem associated with $\mathcal{M}'$ involves countable state space with

unbounded cost. Therefore, the first step for us was to verify the existence of optimal stationary policy by verifying that $\mathcal{M}'$ satisfies sufficient conditions presented in [123]. There is good news and bad news. The good news is that an optimal policy does exist (see Section 4.5 at the end of the chapter), with the value function satisfying the Bellman's optimality equation:

$$V_\alpha(y,h) = \min\{\hat{V}_\alpha(y,h;0), \hat{V}_\alpha(y,h;1)\}, \tag{4.5}$$

where

$$\hat{V}_\alpha(y,h;0) = y + \alpha V_\alpha(y+1,h+1), \quad \text{and} \tag{4.6}$$

$$\hat{V}_\alpha(y,h;1) = y + c + \alpha\big(\sum_{i=1}^{h} p\bar{p}^{i-1}V_\alpha(i,1) + \bar{p}^h V_\alpha(y+1,1)\big). \tag{4.7}$$

However, the bad news is that finding optimal policy at the time of writing this thesis seems intractable, and we discuss a few roadblocks below.

Intuition suggests that the Reader should sample when the age at the client input is "large", or if it has been "many" slots since the Reader last read. This means that an optimal policy might have a threshold structure. Particularly, the hypothesis is that if it is optimal to sample in (y,h), then it is also optimal to sample in any (y',h) with $y' \geq y$ or $h' \geq h$. For a two-dimensional state MDP, usually the methodology deployed in proving the threshold results consists of two steps: showing monotonicity of the optimal value function $V_\alpha(y,h)$, and showing that the state-action cost function $\hat{V}_\alpha(y,h;a)$ is submodular in the action and state variable [125].

As proven in Section 4.5, the value function $V_\alpha(y,h)$ for any $(y,h) \in S$ has an uncommon monotonicity property. Particularly, $V_\alpha(y,h)$ is monotonically non-decreasing in y and monotonically non-increasing in h. However, this monotonicity property, along with convoluted transition probabilities associated with MDP $\mathcal{M}'$, makes proving submodularity really challenging.

Nevertheless, despite these challenges in proving the submodularity and fully characterizing the optimal policy, in the next section, we turn our attention to studying heuristic policies. By exploring heuristic policies that leverage the observed monotonicity properties and intuitive threshold-based

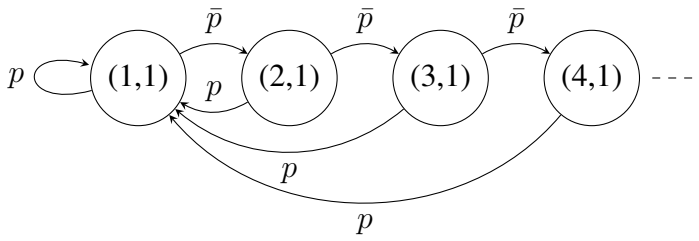


Figure 4.3.1: Discrete-time Markov Chain for the policy in which the Reader reads in every slot.

decision rules, we aim to achieve near-optimal performance.

4.3 Heuristic policies

4.3.1 Always Sample Policy (ASP)

Under the deterministic stationary policy of reading in every slot, the system exhibits an irreducible, ergodic Markov Chain with states $(i, 1)$, where $i \in \mathbb{N}$ represents the age at the client input after sampling. The corresponding Markov Chain is shown in Fig. 4.3.1. The steady-state distribution is

$$\pi_{(i,1)} = p\bar{p}^{i-1}, \qquad i = 1, 2, \ldots \tag{4.8}$$

In state $(i, 1)$, a non-negative cost $C((i, 1); 1) = c + i$ is incurred. Then, the average cost for always sample policy g_{AS} is

$$g_{\text{ASP}} = \sum_i \pi_{(i,1)} C(i, 1) = \sum_i p\bar{p}^{i-1}(c + i) = c + \frac{1}{p}. \tag{4.9}$$

4.3.2 Probabilistic Reading (PR)

In this policy, the Reader decides to read a slot with a fixed probability q. This probabilistic approach balances the trade-off between the frequency of sampling and the associated costs, while also maintaining the freshness at the Reader. To analyze the performance of the PR policy, we begin by examining the average age of update stored in memory. Recall that updates arrive at the Writer following a Bernoulli process with parameter p, leading to inter-publication times, denoted

as X_i, which form an independent and identically distributed (i.i.d.) sequence of geometric random variables with parameter p. The evolution of the age process $x(t)$ in memory is illustrated in Fig. 4.3.2. The area under polygon A_n in Fig. 4.3.2 is $A_n = (X_n - 1)X_n/2$. The average age in the memory can be calculated as:

$$\begin{aligned} \mathrm{E}[x(t)] &= \frac{\mathrm{E}[\text{Area}]}{\mathrm{E}[X]} = \frac{\mathrm{E}[0.5X(X-1)]}{\mathrm{E}[X]}, \\ &= \frac{1}{2\,\mathrm{E}[X]}\left(\mathrm{E}[X^2] - \mathrm{E}[X]\right) = \frac{0.5(2-p)/p^2}{1/p} - \frac{1}{2}, \\ &= \frac{1}{p} - 1. \end{aligned} \tag{4.10}$$

Next, let Z be a random variable indicating whether the Reader reads a slot i.e.,

$$Z = \begin{cases} 1, & \text{if Reader reads in a slot,} \\ 0, & \text{otherwise.} \end{cases}$$

Then, under the PR policy, the average age at the client input will be:

$$\begin{aligned} \mathrm{E}[y(t)] &= \mathrm{E}[y(t) \mid Z = 0]\,\mathrm{P}[Z = 0] + \mathrm{E}[y(t) \mid Z = 1]\,\mathrm{P}[Z = 1], \\ &= (1-q)(\mathrm{E}[y(t)] + 1) + q(\mathrm{E}[x(t)] + 1), \\ &= \frac{1}{q} + \mathrm{E}[x(t)] = \frac{1}{q} + \frac{1}{p} - 1. \end{aligned} \tag{4.11}$$

Since c is the cost associated with sampling, the average updating cost is cq. Therefore the average cost under Probabilistic Reading policy is:

$$g(q) = cq + \frac{1}{q} + \frac{1}{p} - 1. \tag{4.12}$$

Notice that when $q = 1$, the PR policy reduces to the Always Sample policy, where the Reader reads every slot. To find the probability q that minimizes the average cost $g(q)$ in (4.12), we differentiate

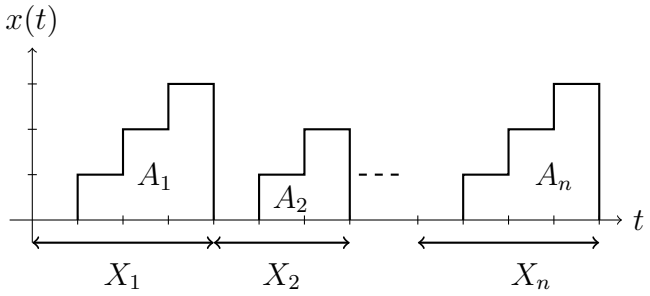


Figure 4.3.2: Age evolution of update stored in the memory.

$g(q)$ with respect to q and set the derivative equal to zero:

$$\frac{dg}{dq} = c - \frac{1}{q^2} = 0 \;\Rightarrow q = \frac{1}{\sqrt{c}}.$$

Substituting $q = 1/\sqrt{c}$ into the cost function $g(q)$ in (4.12) yields the minimum average cost under the PR policy,

$$g_{\text{PR}} = g(1/\sqrt{c}) = 2\sqrt{c} + \frac{1}{p} - 1. \tag{4.13}$$

4.3.3 Fixed-Wait Policy (FW)

In this policy, the reader waits for a deterministic number of slots, represented by $\overline{H}$, before sampling the memory. This means that starting from an arbitrary state $(y, 1)$, the system successively transitions to states $(y + 1, 2)$, then $(y + 2, 3)$, and so on, until it reaches $(y + \overline{H} - 1, \overline{H})$.

The state $(y + \overline{H} - 1, \overline{H})$ physically signifies that $\overline{H}$ slots have elapsed since the Reader last sampled the memory. At this point, according to the FW policy, the Reader performs a sampling action, which effectively resets the age at the client input. The reset age could either be one of the values $1, 2, \ldots, \overline{H}$ depending on the freshness of the update in memory, or it could be $y + \overline{H}$ if no new updates have been published during the waiting period.

For such a policy, rather than the usual interpretation in which state of the system is given by a tuple (y, h), we view the state of the system changing only at the read times. We refer to such a state as a post-action state. Let $Y(t)$ be the age after an action in slot t, then $Y(t)$ is called the post-action

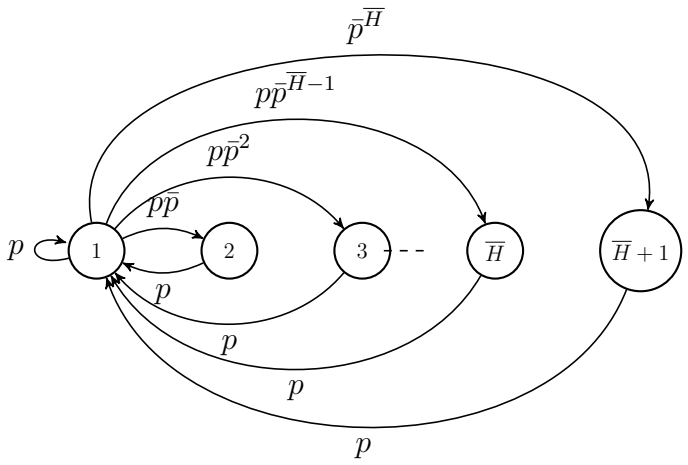


Figure 4.3.3: Embedded Markov Chain for threshold in h policy (for clarity only transitions out of and into state 1 are shown and transitions in and out of other states are omitted).

age, which is different from pre-action age $y(t)$. A similar idea was also presented in [111].

Let $\overline{H} \in 1, 2, \ldots$ represent the fixed wait time. In this context, the Fixed-Wait Sampling Policy induces a semi-Markov process with $\overline{H}$ being the holding time in each state and an embedded Markov Chain $\{Y_n, n \geq 0\}$ with Y_n being the age at client input immediately following transition n. Fig. 4.3.3 illustrates the corresponding embedded Markov Chain; only transitions out of and into state 1 are shown. A notable detail not immediately apparent in Fig. 4.3.3 is that if the embedded Markov Chain state is 1, in the original Markov process, this corresponds to state $(1, 1)$. Starting in state 1, the system waits for $\overline{H}$ slots, taking the original system state to $(\overline{H} - 1, \overline{H})$. Upon the reader's next read, the original system transitions to either state $(1, 1)$, that corresponds to state 1 in embedded Markov Chain with probability p, or to $(2, 1)$ corepsonding to state 2 in embedded markov chain with probability $p\bar{p}$ that corresponds to state 2, and so on, up to $(\overline{H}, 1)$ with probability $p\bar{p}^{\overline{H}-1}$, or to $(\overline{H} + 1, 1)$ with probability $\bar{p}^{\overline{H}}$. In general, transitioning from state $(y, \overline{H})$, with $y \geq 1$ to state $(i, 1)$ with $i \in \{1, 2, \ldots, \overline{H} + 1\}$ in the original Markov process corresponds to a transition in the embedded Markov Chain to state i with probability $p\bar{p}^{i-1}$.

The Markov Chain is irreducible, aperiodic and ergodic and that any transition into state i is with probability $p\bar{p}^{i-1}$. Let $\{\pi_i : i \geq 0\}$ be the steady state probabilities for the embedded Markov Chain. If $U(i)$ is the expected holding interval in state i per transition into i, the the time-average

probability of being in state i is expressed as [126]

$$P_i = \frac{\pi_i U(i)}{\sum_j \pi_j U(j)}. \tag{4.14}$$

For a Fixed-Wait Sampling policy, where the holding time is a constant $\overline{H}$ for each state, it follows that $P_i = \pi_i$. The steady-state probability for state i in the embedded Markov Chain, as depicted in Fig. 4.3.3, is given by:

$$\pi_i = p\bar{p}^{i-1}, \qquad i \in 1, 2, \ldots. \tag{4.15}$$

Under such a policy, the cost in each slot comprises the age in that slot plus the charge c of sampling (if any) in that slot. For instance, the cost incurred in state i, $\hat{c}(i)$, with $i \in \{1, 2, 3, \ldots\}$ is:

$$\hat{c}(i) = i + (i+1) + (i+2) \ldots + (i+\overline{H}) + \frac{c}{\overline{H}}, \tag{4.16}$$

where the last term in the sum represents the charge paid at the beginning where the Reader samples, and then stays idle for $\overline{H} - 1$ slots. In general for any state $i \in \{1, \ldots \overline{H}\}$, the cost incurred is

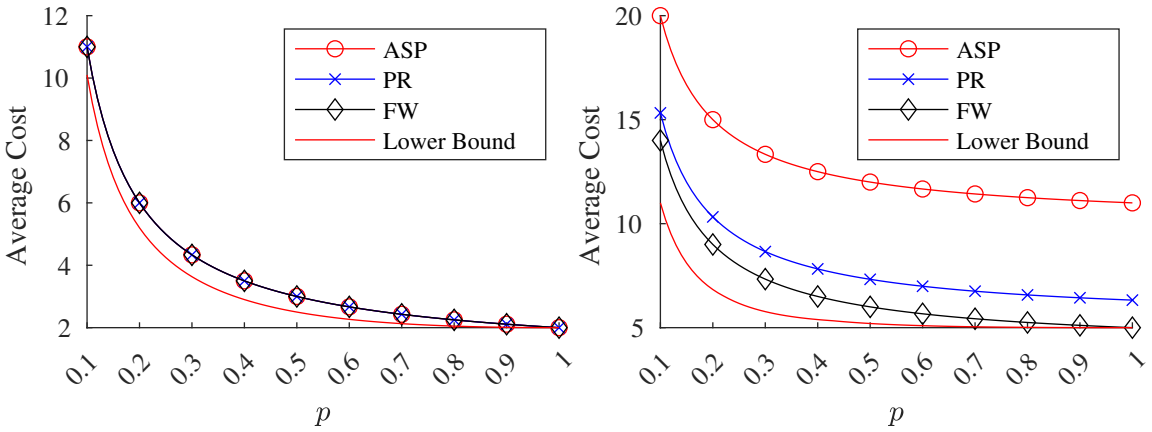
$$\hat{c}(i) = \frac{1}{2}(\overline{H} + 2i - 1) + \frac{c}{\overline{H}}, \tag{4.17}$$

and for any state $\overline{H} + i$ with $i \in \{1, 2, 3, \ldots\}$, the average cost is

$$\frac{1}{2}(3\overline{H} + 2i - 1) + \frac{c}{\overline{H}}. \tag{4.18}$$

Consequently, the average cost under such a threshold policy is given by:

$$\begin{aligned} g(\overline{H}) &= \sum_{i=1}^{\infty} P_i \hat{c}(i) = \sum_{i=1}^{\infty} \pi_i \hat{c}(i), \\ &= p \sum_{i=1}^{\overline{H}} \bar{p}^{i-1} \left(\frac{1}{2}(\overline{H} + 2i - 1) + \frac{c}{\overline{H}} \right) + p\bar{p}^{\overline{H}} \sum_{i=1}^{\infty} \bar{p}^{i-1} \left(\frac{1}{2}(3\overline{H} + 2i - 1) + \frac{c}{\overline{H}} \right), \\ &= \frac{1}{2}(\overline{H} - 1) + \frac{c}{\overline{H}} + \frac{1}{p}. \end{aligned} \tag{4.19}$$



(a) $c = 1$ (b) $c = 10$

Figure 4.4.1: Average cost of heuristic policies (ASP, PR, and FW) as a function of the source update arrival probability p, compared to the optimal policy when the memory state is known (Lower Bound). The plots illustrate the impact of sampling costs: (a) $c = 1$ and (b) $c = 10$.

It is worth noting that $\overline{H} = 1$ implies that the Reader reads in every slot, and the cost is identical to Always Sample Policy. Setting $dg/d\overline{H} = 0$ gives the minimum value of threshold as $\sqrt{2c}$. But since $\overline{H}$ is a discrete variable, the optimal threshold is $\overline{H}^* = \lceil\sqrt{2c}\rceil$. This means that when $c = 0$, the optimal threshold is $\overline{H} = 1$, and this means that the Reader samples in every slot and the average cost will be $g = c + 1/p$. When $c > 0$, using (4.19), the minimum average cost under the FW policy is

$$g_{\text{FW}} = g\left(\left\lceil\sqrt{2c}\right\rceil\right) = \frac{1}{2}\left(\left\lceil\sqrt{2c}\right\rceil - 1\right) + \frac{c}{\lceil\sqrt{2c}\rceil} + \frac{1}{p}. \tag{4.20}$$

4.4 Numerical Evaluation

In this section, we plot the average cost for each heuristic policy—Always Sample Policy (ASP), Probabilistic Reading (PR), and Fixed-Wait (FW)—as a function of the update arrival probability p for two different sampling costs: $c = 1$ (Fig. 4.4.1(a)) and $c = 10$ (Fig. 4.4.1(b)). Additionally, we include a plot of the average cost when the memory state is known. This known-state average cost acts as a theoretical lower bound for the heuristic policies because these policies are designed for scenarios where the memory state is not directly observable.

When $c = 1$, it follows from (4.9), (4.13) and (4.20) that $g_{\text{ASP}} = g_{\text{PR}} = g_{\text{FW}} = 1 + 1/p$.

Consequently, in Fig. 4.4.1(a), we observe that the curves corresponding to ASP, PR, and FW policies overlap for this cost value. This overlapping behavior illustrates that, under a low sampling cost, the choice of policy does not affect the average cost significantly. Additionally, for the PR policy, the optimal sampling probability $q^* = 1$ when $c = 1$, which implies that the Reader samples in every slot in the PR policy.

Moreover, when $c = 1$ and $p = 1$, we observe from (3.11) in Chapter 3 that the optimal threshold is $Y_0^* = 1$, which means the Reader samples in every slot. Thus, we see in Fig. 4.4.1(a) that all the policies converge to the ASP when $c = 1$ and $p = 1$. When p is close to 1, the advantage of knowing the memory state information is not significant, since there will most likely be new update in the memory in every slot, and thus we see that every policy is close to the lower bound.

When c is very large, and the updates are infrequent i.e. p is small, then knowing the state is pretty valuable, and thus we observe a significant gap in the lower bound and other heuristic policies, as illustrated by Fig. 4.4.1(b). Again, we observe from (3.11) in Chapter 3 that when c is large and $p = 1$, the optimal threshold is $Y_0^* \approx \lceil\sqrt{2c}\rceil$, which is approximately equal to the optimal threshold for the FW policy, where $\overline{H}^* = \lceil\sqrt{2c}\rceil$. Thus, we see that the average cost of FW policy is almost identical to the lower bound at $p = 1$.

4.5 Existence of Average Cost Stationary Optimal Policy

We introduce the α-discounted version of the problem, where $0 < \alpha < 1$ denotes the discount factor. For a given initial state s, the total expected discounted cost under policy π is defined as:

$$V_{\pi,\alpha}(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \alpha^t C(s(t), a(t)) \mid s(0) = s \right]. \tag{4.21}$$

It's worth noting that (4.21) may yield an infinite value, particularly when the cost function $C(s(t), a(t))$ is unbounded. Let $V_\alpha(s) = \inf_\pi V_{\pi,\alpha}(s)$. A policy π^* is considered α-*optimal* if it satisfies $V_{\pi^*,\alpha}(s) = V_\alpha(s), \forall s \in S$. We now state the conditions presented in [123] for existence of stationary optimal policy for average cost MDP with countable state space and unbounded costs:

Theorem 4. *[123, Theorem] If the following conditions hold for MDP $\mathcal{M}'$:*

1. *For every state s and discount factor α, the quantity $V_\alpha(s)$ is finite,*
2. *$f_\alpha(s) \coloneqq V_\alpha(s) - V_\alpha(0)$ satisfies $-N \overset{(a)}{\leq} f_\alpha(s) \overset{(b)}{\leq} M(s)$, where $M(s) \geq 0$, and*
3. *For all s and a, $\sum_{s'} \mathbb{P}_{s,s'}(a) M(s') < \infty$,*

then there exists a stationary policy that is average cost optimal for $\mathcal{M}'$. Moreover, for $\mathcal{M}'$, there exists a constant $g = \lim_{\alpha \to 1}(1-\alpha)V_\alpha(s)$ for every state s, and a function $f(s)$ with $-N \leq f(s) \leq M(s)$ that solve relative-cost Bellman's equation,

$$g + f(s) = \min_a \{C(s;a) + \sum_{s' \in S} \mathbb{P}_{s,s'}(a) f(s')\}. \tag{4.22}$$

For $\mathcal{M}'$, we choose reference state 0 as $(1,1)$. A sufficient condition for 1 and $2(b)$ to hold is the existence of a single stationary policy that induces an irreducible, ergodic Markov Chain, with the associated expected cost of first passage from any state (y,h) to state $(1,1)$ being finite ([123, Propositions 4 and 5]). The following lemma verifies that this sufficient condition is met for our problem.

Lemma 4. *Under the deterministic stationary policy θ of reading in every slot, the system exhibits an irreducible, ergodic Markov Chain, with expected cost $M(y,h)$ of first passage from state $s = (y,h)$ to $(1,1)$ satisfying*

$$M(y,h) \leq \frac{1}{p}\left(y + c(1+p)\right) + \frac{2}{p^2}. \tag{4.23}$$

The proof appears in the Appendix 4.A. Now, condition 3 of Theorem 4 asserts that under any action $a \in \{0,1\}$, the quantity $\sum_{s'} \mathbb{P}_{s,s'}(a) M(s')$ should be finite. For MDP $\mathcal{M}'$, from (4.2b), when $a = 0$, we have for any state $s = (y,h)$,

$$\sum_{s'} \mathbb{P}_{s,s'}(0) M(s') = M(y+1, h+1). \tag{4.24}$$

From (4.2a), when $a = 1$, we similarly have for any state $s = (y, h)$,

$$\sum_{s'} \mathbb{P}_{s,s'}(1) M(s') = \sum_{i=1}^{h} p\bar{p}^{i-1} M(i, 1) + \bar{p}^h M(y + 1, 1). \tag{4.25}$$

It follows from (4.24), (4.25) and Lemma 4 that condition 3 holds for MDP $\mathcal{M}'$.

We now focus on lower bounding the relative cost function $f_\alpha(s) = V_\alpha(s) - V_\alpha(0)$ to satisfy condition 2 in Theorem 4. Since $V_\alpha(y, h)$ is finite for every $(y, h) \in S$, it satisfies the discounted cost optimality equation [121, Theorem 1.1, Chapter III] which is given as:

$$V_\alpha(y, h) = \min\big\{y + \alpha V_\alpha(y + 1, h + 1), y + c + \alpha\big(\sum_{i=1}^{h} p\bar{p}^{i-1} V_\alpha(i, 1) + \bar{p}^h V_\alpha(y + 1, 1)\big)\big\}. \tag{4.26}$$

Here, the first term of $\min$ corresponds to the reader staying idle ($a = 0$), and the second term corresponds to the reader sampling ($a = 1$). We define a value iteration $V_{\alpha,n}(s)$ by $V_{\alpha,0}(s) = 0, \forall s \in S$, and, for any $n > 0$,

$$\begin{aligned} V_{\alpha,n+1}(y, h) = \min\big\{&y + \alpha V_{\alpha,n}(y + 1, h + 1), \\ &y + c + \alpha\big(\sum_{i=1}^{h} p\bar{p}^{i-1} V_{\alpha,n}(i, 1) + \bar{p}^h V_{\alpha,n}(y + 1, 1)\big)\big\}. \end{aligned} \tag{4.27}$$

In the following, Lemma 5 and Lemma 6 define monotonicity properties of value iteration function $V_{\alpha,n}(y, h)$ and value function $V_\alpha(y, h)$ satisfying (4.27) and (4.26) respectively. These properties aid in verifying condition 2 of Theorem 4.

Lemma 5. *The discounted cost value iteration has the following monotonicity properties:*

(a) Function $V_{\alpha,n}(y, h)$ is non-decreasing in y for every n and α.

(b) Function $V_{\alpha,n}(y, h)$ is non-increasing in h for every n and α.

The proof of Lemma 5 appears in Appendix 4.B. Next, using Lemma 5, we state the monotonicity property of the value function $V_\alpha(y, h)$.

Lemma 6. *(a) For any fixed $h \leq y$, function $V_\alpha(y,h)$ is non-decreasing in y i.e.*

$$V_\alpha(y,h) \leq V_\alpha(y+1,h) \leq V_\alpha(y+2,h) \dots \tag{4.28}$$

(b) For any fixed y, function $V_\alpha(y,h)$ is non-increasing in h i.e.

$$V_\alpha(y,1) \geq V_\alpha(y,2) \geq V_\alpha(y,3) \dots \geq V_\alpha(y,y). \tag{4.29}$$

Proof. The proof follows from Lemma 5(a) and 5(b), and the observation that $V_{\alpha,n}(s) \to V_\alpha(s)$ as $n \to \infty$ [121, Theorem 4.2, Chapter III]. □

Lower bound on $f_\alpha(s)$

Let state $(1,1)$ be the reference state. We aim to show that $f_\alpha(s) = V_\alpha(y,h) - V_\alpha(1,1) \geq 0$ for all states (y,h). First, we prove $V_\alpha(y+1,y+1) - V_\alpha(y,y) \geq 1$ for every $y \geq 1$. Then, we use the monotonicity properties provided by Lemma 6 to conclude the validity of condition 2 of Theorem 4 for our case.

Lemma 7. *For every discount factor α, and $y \geq 1$, the following inequality holds:*

$$V_\alpha(y+1,y+1) - V_\alpha(y,y) \geq 1. \tag{4.30}$$

The proof of Lemma 7 is in Appendix 4.C.

Lemma 8. *The relative cost value function for MDP $\mathcal{M}'$ satisfies:*

$$f_\alpha(y,h) \geq 0. \tag{4.31}$$

Proof. We want to show that $f_\alpha(y,h) = V_\alpha(y,h) - V_\alpha(1,1) \geq 0$. By applying Lemma 7 recursively

and adding the inequalities, we obtain

$$V_\alpha(y, y) - V_\alpha(1, 1) \geq 0. \tag{4.32}$$

Finally, by using Lemma 6(b), we have that for any $h \leq y$, $V_\alpha(y, h) \geq V_\alpha(y, y)$, and it follows that

$$V_\alpha(y, h) - V_\alpha(1, 1) \geq 0. \tag{4.33}$$

□

Lemma 8 shows that the relative cost $f_\alpha(s)$ is indeed lower-bounded in $\mathcal{M}'$, thereby satisfying condition 2 of Theorem 4. Consequently, there exists a constant $g = \lim_{\alpha \to 1}(1 - \alpha)V_\alpha(y, h)$ for every state (y, h) that is an optimal average cost along with a relative cost-to-go function $f(y, h)$ satisfying $0 \leq f(y, h) \leq M(y, h)$.

APPENDIX

4.A Proof of Lemma 4

Lemma 4. Under the deterministic stationary policy θ of reading in every slot, the system exhibits an irreducible, ergodic Markov Chain, with expected cost $M(y,h)$ of first passage from state $s=(y,h)$ to $(1,1)$ satisfying

$$M(y,h) \leq \frac{1}{p}\left(y + c(1+p)\right) + \frac{2}{p^2}. \tag{4.34}$$

Proof. Note that

$$M(y,h) \leq \mathrm{E}[\hat{C}(y,h)], \tag{4.35}$$

where $\hat{C}(y,h)$ is the first passage cost under the policy θ in which the Reader samples in every slot. Such a policy will induces an irreducible and ergodic discrete-time Markov Chain with states $(i,1), i \in \mathbb{N}$ as illustrated in Fig. 4.3.1. The expected cost of first passage from state $(i,1)$ to $(1,1)$ is

$$\begin{aligned}
\mathrm{E}[\hat{C}(i,1)] &= \sum_{n=1}^{\infty} p\bar{p}^{n-1}\left(n(i+c) + \frac{n(n-1)}{2}\right), \\
&= \frac{c+i-1}{p} + \frac{1}{p^2}, \\
&\leq \frac{1}{p}(i+c) + \frac{1}{p^2}, \\
&\leq \frac{1}{p}(i + c(1+p)) + \frac{2}{p^2}.
\end{aligned} \tag{4.36}$$

Let $\hat{S}(y,h) = \{(i,1) : i \in \{1,2,\ldots,h\} \cup \{y+1\}\}$ be the set of states reachable from a transient state (y,h) under the "always sample" policy θ. Then, starting from state (y,h) under the "always sample" policy, a cost $y+c$ is incurred in the present slot and the system transitions to state $(i,1)$

with $i \in \hat{S}(y,h)$. In the next slot, an additional cost $\hat{C}(i,1)$ is incurred to reach $(1,1)$. For any $1 < h \leq y$, the expected first passage cost from (y,h) to $(1,1)$ is then:

$$\begin{aligned}
\mathrm{E}[\hat{C}(y,h)] &= y + c + \sum_{i=1}^{h} p\bar{p}^{i-1}\,\mathrm{E}[\hat{C}(i,1)] + \bar{p}^{h}\,\mathrm{E}[\hat{C}(y+1,1)], && (4.37)\\
&= y + c + \frac{1}{p^2} + \sum_{i=1}^{h} p\bar{p}^{i-1}\frac{c+i-1}{p} + \bar{p}^{h}\frac{c+y}{p},\\
&= y + c(1+\frac{1}{p}) + \frac{1}{p^2} + \sum_{i=1}^{h} \bar{p}^{i-1}(i-1) + \frac{\bar{p}^{h}y}{p},\\
&= y + \frac{\bar{p}^{h}}{p}(y-h) + \frac{2-\bar{p}^{h+1}+p(c-1)+cp^2}{p^2},\\
&\leq y + \frac{\bar{p}^{h}}{p}y + \frac{2+cp+cp^2}{p^2},\\
&\leq y + \frac{\bar{p}}{p}y + \frac{2+cp(1+p)}{p^2},\\
&= \frac{1}{p}(y + c(1+p)) + \frac{2}{p^2}. && (4.38)
\end{aligned}$$

□

4.B Proof of Lemma 5

Lemma 5. The discounted cost value iteration has the following monotonicity properties.

(a) Function $V_{\alpha,n}(y,h)$ is non-decreasing in y for every n and α.

(b) Function $V_{\alpha,n}(y,h)$ is non-increasing in h for every n and α.

Proof. We make use of few shorthand notations in expressions that would otherwise be too complicated to write.

Definition 1.

$$\delta_{\alpha}(s;a) = \sum_{s' \in S} \mathbb{P}_{s,s'}(a) V_{\alpha}(s'), \qquad \textit{and} \tag{4.39}$$

$$\delta_{\alpha,n}(s;a) = \sum_{s' \in S} \mathbb{P}_{s,s'}(a) V_{\alpha,n}(s'). \tag{4.40}$$

Definition 2.

$$\hat{V}_\alpha(s;a) = C(s,a) + \alpha\delta_\alpha(s;a), \qquad \textit{and,} \tag{4.41}$$

$$\hat{V}_{\alpha,n+1}(s;a) = C(s,a) + \alpha\delta_{\alpha,n}(s;a). \tag{4.42}$$

Using (4.39), (4.40), (4.41) and (4.42) , we have the following shorthand for discounted cost optimality equation and value iteration

$$V_\alpha(s) = \min_a \{\hat{V}_\alpha(s;a)\}, \tag{4.43}$$

$$V_{\alpha,n+1}(s) = \min_a \{\hat{V}_{\alpha,n+1}(s;a)\}. \tag{4.44}$$

4.B.1 Properties of MDP $\mathcal{M}'$

Recall that the state for MDP $\mathcal{M}'$ is a tuple $s = (y,h)$, and $a \in \{0,1\}$. Then using (4.2a) and (4.2b), we have

$$\delta_\alpha(y,h;0) = V_\alpha(y+1,h+1), \tag{4.45a}$$

$$\delta_\alpha(y,h;1) = p\sum_{i=1}^{h} \bar{p}^{i-1} V_\alpha(i,1) + \bar{p}^h V_\alpha(y+1,1). \tag{4.45b}$$

Similarly,

$$\delta_{\alpha,n}(y,h;0) = V_{\alpha,n}(y+1,h+1), \tag{4.46a}$$

$$\delta_{\alpha,n}(y,h;1) = p\sum_{i=1}^{h} \bar{p}^{i-1} V_{\alpha,n}(i,1) + \bar{p}^h V_{\alpha,n}(y+1,1). \tag{4.46b}$$

Likewise,

$$\hat{V}_\alpha(y,h;0) = C(y,h;0) + \alpha\delta_\alpha(y,h;0), \tag{4.47a}$$

$$\hat{V}_\alpha(y,h;1) = C(y,h;1) + \alpha\delta_\alpha(y,h;1). \tag{4.47b}$$

And,

$$\hat{V}_{\alpha,n+1}(y,h;0) = C(y,h;0) + \alpha\delta_{\alpha,n}(y,h;0), \tag{4.48a}$$

$$\hat{V}_{\alpha,n+1}(y,h;1) = C(y,h;1) + \alpha\delta_{\alpha,n}(y,h;1). \tag{4.48b}$$

It follows from (4.43) and (4.44) that,

$$V_\alpha(y,h) = \min\{\hat{V}_\alpha(y,h;0), \hat{V}_\alpha(y,h;1)\}, \tag{4.49}$$

$$V_{\alpha,n+1}(y,h) = \min\{\hat{V}_{\alpha,n+1}(y,h;0), \hat{V}_{\alpha,n+1}(y,h;1)\}. \tag{4.50}$$

As in common literature on MDP, we refer to $V_{\alpha,n}(\cdot,\cdot)$ as the value function and $\hat{V}_{\alpha,n}(\cdot,\cdot\,;\,\cdot)$ as the state-action cost function. The following proposition will be helpful in proving later results.

Proposition 7. *If $f' \geq f$, and $g' \geq g$, then*

$$\min(f',g') \geq \min(f,g). \tag{4.51}$$

Proof. The proof follows from the following property of the $\min$ function:

$$\min(f',g') \geq \min(f',g',g) \geq \min(f',g) \geq \min(f',f,g) \geq \min(f,g). \tag{4.52}$$

□

4.B.2 Proof of Lemma 5(a)

The proof is by induction in n. It is immediately obvious that $V_{\alpha,0}((y,h)) = 0$ is non-decreasing in y. Now assume that statement is true for all values of $n \leq k$. This means that for any fixed h_0, $V_{\alpha,k}(y,h_0) \leq V_{\alpha,k}(y+1,h_0)$. For $n = k+1$, we have from (4.50)

$$V_{\alpha,k+1}(y,h) = \min\{\hat{V}_{\alpha,k+1}(y,h;0), \hat{V}_{\alpha,k+1}(y,h;1)\} \tag{4.53}$$

Our approach is to establish that $\hat{V}_{\alpha,k+1}(y,h;0)$ and $\hat{V}_{\alpha,k+1}(y,h;1)$ are non-decreasing functions of y. For any $a \in \{0,1\}$, we have from (4.48a) and (4.48b), $\hat{V}_{\alpha,k+1}(y,h;a) = C(y,h;a) + \alpha\delta_{\alpha,k}(y,h;a)$. First, we note that, cost per stage is non-decreasing in y for every $a \in \{0,1\}$, i.e., for a fixed h_0, we have

$$C(y,h_0;a) \leq C(y+1,h_0;a). \tag{4.54}$$

It remains to demonstrate that $\delta_{\alpha,k}(y,h;a)$ is also non-decreasing in y for every $a \in \{0,1\}$. For any $s = (y,h_0)$ and $a = 1$,

$$\delta_{\alpha,k}(y,h_0;1) = p\sum_{i=1}^{h_0} \bar{p}^{i-1} V_{\alpha,k}(i,1) + \bar{p}^{h_0} V_{\alpha,k}(y+1,1). \tag{4.55}$$

Similarly, when $s = (y+1,h_0)$ and $a = 1$,

$$\delta_{\alpha,k}(y+1,h_0;1) = p\sum_{i=1}^{h_0} \bar{p}^{i-1} V_{\alpha,k}(i,1) + \bar{p}^{h_0} V_{\alpha,k}(y+2,1). \tag{4.56}$$

From the induction hypothesis, with $h_0 = 1$, $V_{\alpha,k}(y+1,1) \leq V_{\alpha,k}(y+2,1)$. This means that

$$\delta_{\alpha,k}(y,h_0;1) \leq \delta_{\alpha,k}(y+1,h_0;1). \tag{4.57}$$

Again from induction hypothesis, we have for $a = 0$,

$$\begin{aligned}\delta_{\alpha,k}(y, h_0; 0) &= V_{\alpha,k}(y+1, h_0+1), \\ &\leq V_{\alpha,k}(y+2, h_0+1) = \delta_{\alpha,k}(y+1, h_0; 0). \end{aligned} \tag{4.58}$$

Hence, $\delta_{\alpha,k}(y, h; a)$ is non-decreasing in y and so $\hat{V}_{\alpha,k+1}(y, h; a)$ is non-decreasing in y for every $a \in \{0, 1\}$. Based on (4.53) and Proposition 7, we conclude that $V_{\alpha,k+1}(y, h)$ is non-decreasing in y, and by the principle of mathematical induction, the lemma is true for all positive integers n.

4.B.3 Proof of Lemma 5(b)

The proof is similar to part (a) of the lemma, with the focus on demonstrating that $\hat{V}_{\alpha,k+1}(y, h; 0)$ and $\hat{V}_{\alpha,k+1}(y, h; 1)$ are non-increasing in h. For any $s = (y_0, h)$ with fixed y_0, and $a = 1$,

$$\delta_{\alpha,k}(y_0, h; 1) = p \sum_{i=1}^{h} \bar{p}^{i-1} V_{\alpha,k}(i, 1) + \bar{p}^h V_{\alpha,k}(y_0 + 1, 1). \tag{4.59}$$

Similarly, when $s = (y_0, h+1)$ and $a = 1$,

$$\delta_{\alpha,k}(y_0, h+1; 1) = p \sum_{i=1}^{h+1} \bar{p}^{i-1} V_{\alpha,k}(i, 1) + \bar{p}^{h+1} V_{\alpha,k}(y_0 + 1, 1). \tag{4.60}$$

Observe that

$$\delta_{\alpha,k}(y_0, h+1; 1) - \delta_{\alpha,k}(y_0, h; 1) = p\bar{p}^h \left(V_{\alpha,k}(h+1, 1) - V_{\alpha,k}(y_0 + 1, 1)\right) \leq 0, \tag{4.61}$$

where the last inequality follows from Lemma 5(a) and the fact that $h \leq y_0$. When $a = 0$, we have from induction hypothesis,

$$\delta_{\alpha,k}(y_0, h; 0) = V_{\alpha,k}(y_0 + 1, h+1) \geq V_{\alpha,k}(y_0 + 1, h+2) = \delta_{\alpha,k}(y_0, h+1; 0). \tag{4.62}$$

Hence, $\delta_{\alpha,k}(y,h;a)$ is non-increasing in h and so $\hat{V}_{\alpha,k+1}(y,h;a)$ is non-increasing in h for every $a \in \{0,1\}$. Based on (4.53) and Proposition 7, we conclude that $V_{\alpha,k+1}(y,h)$ is non-increasing in h, and by the principle of mathematical induction, the lemma is true for all positive integers n. $\square$

4.C Proof of Lemma 7

Lemma 7. For any $y \geq 1$, the following inequality holds for MDP $\mathcal{M}$:

$$V_\alpha(y+1,y+1) - V_\alpha(y,y) \geq 1. \tag{4.63}$$

Proof. We prove this by using mathematical induction i.e. we show that for every $n \in \mathbb{N}$,

$$V_{\alpha,n}(y+1,y+1) - V_{\alpha,n}(y,y) \geq 1. \tag{4.64}$$

For the base case of $n = 1$, since $V_{\alpha,0}(y,h) = 0$ for every (y,h), this implies that $V_{\alpha,1}(y,h) = \min\{C((y,h),0), C((y,h),1))\} = y$, and therefore,

$$V_{\alpha,1}(y+1,y+1) = y+1 = V_{\alpha,1}(y,y) + 1 \tag{4.65}$$

Now assume that for any $n = k$, we have $V_{\alpha,k}(y+1,y+1) \geq V_{\alpha,k}(y,y) + 1$ for every $y \geq 1$. Now,

$$V_{\alpha,k+1}(y+1,y+1) = 1 + y + \min\{\alpha\delta_{\alpha,k}(y+1,y+1;0), c + \delta_{\alpha,k}(y+1,y+1;1)\}, \tag{4.66}$$

Observe that

$$\begin{aligned}
\alpha\delta_{\alpha,k}(y+1,y+1;0) &= \alpha V_{\alpha,k}(y+2,y+2), \\
&\overset{(a)}{\geq} \alpha(V_{\alpha,k}(y+1,y+1)+1), \\
&\geq \alpha(V_{\alpha,k}(y+1,y+1),
\end{aligned}$$

$$= \alpha\delta_{\alpha,k}(y, y; 0), \tag{4.67}$$

where (a) is based on induction hypothesis. Now,

$$\begin{aligned}
\delta_{\alpha,k}(y+1, y+1; 1) &= p\sum_{i=1}^{y+1} \bar{p}^{i-1} V_{\alpha,k}(i, 1) + \bar{p}^{y+1} V_{\alpha,k}(y+2, 1), \\
&\overset{(a)}{\geq} p\sum_{i=1}^{y+1} \bar{p}^{i-1} V_{\alpha,k}(i, 1) + \bar{p}^{y+1} V_{\alpha,k}(y+1, 1), \\
&= p\sum_{i=1}^{y} \bar{p}^{i-1} V_{\alpha,k}(i, 1) + \bar{p}^{y} V_{\alpha,k}(y+1, 1), \\
&= \delta_{\alpha,k}(y, y; 1),
\end{aligned} \tag{4.68}$$

where, (a) follows from Lemma 5(a). Hence,

$$\begin{aligned}
V_{\alpha,k+1}(y+1, y+1) &\geq 1 + y + \min\{\alpha\delta_{\alpha,k}(y, y; 0), c + \alpha\delta_{\alpha,k}(y, y; 1)\}, \\
&= 1 + \min\{y + \alpha\delta_{\alpha,k}(y, y; 0), y + c + \alpha\delta_{\alpha,k}(y, y; 1)\}, \\
&= V_{\alpha,k+1}(y, y) + 1.
\end{aligned} \tag{4.69}$$

We have verified the inductive step, and by the principle of mathematical induction (4.64) holds. As $\lim_{n\to\infty} V_{\alpha,n}(y, h) = V_{\alpha}(y, h)$, the lemma holds. □

CHAPTER 5
TIMELY PROCESSING OF UPDATES FROM MULTIPLE SOURCES

5.1 Introduction

In this chapter, we model a class of systems (see Fig. 5.1.1) in which two independent sources submit time-stamped updates to a writer that is responsible for publishing the source measurements as updates in the memory. A decision process (DP), as a subscriber, reads the pair of source 1 and source 2 updates from memory and derives a computational result, a *decision update*, from this pair that is delivered to a monitor. The freshness of status information received by subscriber plays an important role in decision making. In this chapter, we first focus on a fundamental problem: What is the average Age of Information (AoI) of decision updates that are computed from time-varying set of sensor data published in the memory. Then, we evaluate a *lazy computation* policy that is subsequently proven to be an optimal policy in minimizing the age of decision updates.

5.2 System Overview

There are three aspects to the system depicted in Fig. 5.1.1:

1. writing the time-varying data received from two sources into the memory,
2. the arbitration between reader and writer to access memory,
3. reading the source data from memory and generating a decision update.

We assume that the arbitration between reader and writer processes is mediated by an RCU-like paradigm. We now give a brief overview of the writing, reading and decision computation processes.

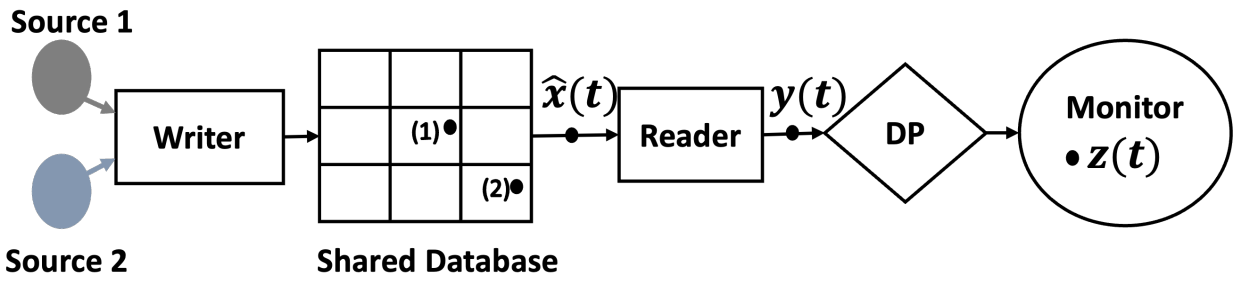


Figure 5.1.1: A writer updates shared database with information fetched from two external sources. A decision process (DP) requests a reader process to read the pair of source updates from the memory. Monitors that track the age of source 1 and 2 updates in the memory are denoted $\bullet(1)$ and $\bullet(2)$ respectively; $\bullet(\hat{x}(t))$ tracks the age of max-age process in the memory, $\bullet(y(t))$ tracks the age of sampled max-age process, and $\bullet(z(t))$ tracks the age of computed decision updates at the external monitor.

5.2.1 Writing Source Updates to the Memory

We assume each source $i \in \{1, 2\}$ independently submits updates as a rate λ_i Poisson process to the network and that these updates arrive fresh at the writer, i.e. with age 0. The write operations to memory have independent exponential (μ) service times. We model the writer as a buffer-less service facility with blocking discipline. Under this model, a source update arriving at the writer will be served only if the writer is idle; otherwise, the update is discarded.

Remark 1: In the present study, we investigate a computational regime characterized by relatively longer decision update times compared to the write times of any update in the memory. Our focus is not on regimes where writing to the memory is the overloaded process. Instead, we are primarily interested in examining the delays associated with computational processing. Whether we adopt a buffer-less or a queuing model, the impact of queuing at the writer is expected to be minimal.

5.2.2 Computing Decision Updates

We view the decision process (DP) reader as one of many subscribers to the updates in the memory system. The DP reader becomes aware of fresher updates in the memory only when it chooses to query the memory for a fresh sample of the source update pair. We assume a reader can fetch the

updates of both source 1 and source 2 from memory in negligible time[1]. With this assumption, the DP reader is an observer that is *sampling* the pair of source updates from the memory as a point process. Based on this sample, the decision process derives a decision update which is sent to the monitor, as shown in Fig. 5.1.1. The reader process fetches the next sample of update pair from the memory only after the computation in progress is completed.

When the DP reader's inter-sample times form a renewal process, this is an example of the model of renewal process sampling of updates introduced in [32]. In this model, the DP reader generates an age process $y(t)$ at the input to the DP that is a sampled version of the max-age process $\hat{x}(t)$ in the memory. Specifically, in the absence of a read, $y(t)$ continues to grow at unit rate. However, if the DP reader makes a read at time τ, then $y(t)$ is reset to $y(\tau) = \hat{x}(\tau)$. This update pair is then processed by the DP for a time T so that at time $\tau + T$ a decision update with age $y(\tau) + T$ is delivered to the monitor. The age at the monitor, $z(t)$, is then reduced to $z(\tau + T) = \hat{x}(\tau) + T$. At this time, the DP reader may choose to fetch a new sample pair from the memory, or it may choose to wait for a time W before fetching the next sample pair. When the DP reader employs non-zero waiting times, we say the DP is using a *lazy sampling* policy [52], and consequently on the DP a *lazy computation* policy is applied. Fig. 5.2.1 illustrates the evolution of age processes $\hat{x}(t), y(t)$, and $z(t)$.

5.2.3 Chapter Overview and Contributions

We divide our AoI analysis into two stages:

1. We analyze the average age of updates in shared memory.

2. Then we analyze the additional delay induced by the decision process computations.

First, section 5.3 presents a stochastic hybrid system (SHS) evaluation of the update age processes in the memory. For the system with sources $i = 1, 2$, we derive the stationary expected ages $\mathrm{E}[x_i(t)]$

[1]This assumption is consistent with RCU reads being lightweight and fast, so that the heavier load is indeed induced by actual decision computation. Further, our model assumes that the DP reader fetches updates from the memory at some finite average rate such that the combined read request process of all subscribers does not overload the shared memory system.

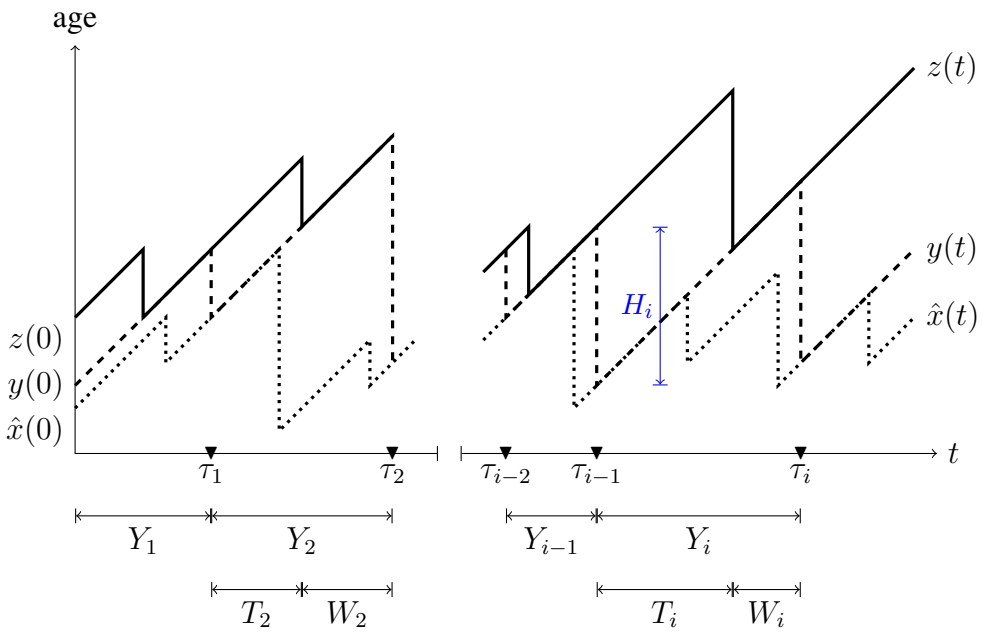


Figure 5.2.1: Example AoI evolution of the max-age process $\hat{x}(t)$ at the memory, the sampled max-age process $y(t)$ with *lazy sampling* at the input to the DP, and the age process $z(t)$ at the monitor. The DP reader samples updates from the memory at times $\tau_1, \tau_2, \ldots$, marked by $\blacktriangledown$. Y_i is the sampling period for sample i, T_i is the computation time for decision update based on sample $i-1$, and W_i is the waiting time to get the i^{th} sample.

as well as the expected age of the *max-age* process $\hat{x}(t) = \max(x_1(t), x_2(t))$.

In section 5.4, stage two of our analysis, we evaluate the age $z(t)$ of the decision update process at the monitor. The decision process is said to be *sampling* the source updates from the memory as it holds a sample of updates that were written to the memory. Even though the sampling and computation of the DP makes no attempt to use the age of its sampled updates to optimize its operation, we show that a lazy sampling policy will be able to reduce $z(t)$. Here we will see that analysis of $z(t)$ is separable from the prior SHS analysis of the max-age process $\hat{x}(t)$ in the shared memory. In particular, the AoI reduction afforded by lazy sampling can be applied to any stationary update age process that is sampled by the DP.

5.3 Age of Source Updates in the Memory

Let $U_{i,1}, U_{i,2} \ldots$ be the sequence of source i update publication times. At any time t, $N_i(t)$ source i updates have been published in the memory, and the most recent update is published at time $U_{i,N_i(t)}$.

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$
1	$0_1 \to 1$	λ_1	$[0, x_1, x_2, \hat{x}]$
2	$1 \to 0_2$	μ	$[x_0, x_0, x_2, x_2]$
3	$0_2 \to 1$	λ_1	$[0, x_1, x_2, \hat{x}]$
4	$0_2 \to 2$	λ_2	$[0, x_1, x_2, \hat{x}]$
5	$2 \to 0_1$	μ	$[x_0, x_1, x_0, x_1]$
6	$0_1 \to 2$	λ_2	$[0, x_1, x_2, \hat{x}]$

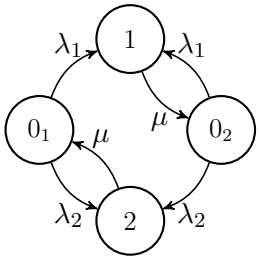


Figure 5.3.1: The SHS transition/reset maps and Markov chain for the update age in the shared memory.

It follows that the source i update process has age $x_i(t) = t - U_{i,N_i(t)}$ in the memory. Under this model, the update age $x_i(t)$ is reset to the write time $W \sim \exp(\mu)$ when it is published at time $U_{i,N_i(t)}$. When the writer writes a fresh source i update at time t', the max-age process $\hat{x}(t)$ is reset to $\hat{x}(t') = x_j(t')$, with $j \neq i$. In the following, we use a Stochastic Hybrid System (SHS) to capture the evolution of update age processes in the memory. An overview of SHS can be found in Chapter 2, Section 2.1.

5.3.1 SHS Analysis of Age in Shared Memory

The age of updates in a shared memory system with bufferless service at the writer can be described by the SHS Markov chain and table of state transitions shown in Fig. 5.3.1. The continuous age state vector is $\mathbf{x} = [x_0, x_1, x_2, \hat{x}]$, where x_0 is the age of the update being written; x_i, $i = 1, 2$, is the age of the source i update in memory; and $\hat{x} = \max(x_1, x_2)$. The discrete state is $\mathcal{Q} = \{0_1, 0_2, 1, 2\}$. At time t, the system is in state 0_i if the writer is idle and the oldest update belongs to source i. State $i \in \{1, 2\}$ corresponds to the writer writing source i update.

We now describe SHS transitions enumerated in the table in Fig. 5.3.1. For each collection of transitions, we focus on the age state components that change.

- $l = 1, 3, 4, 6$: In system idle states 0_1 and 0_2, the writer receives a new source update and initiates a new write mechanism. $x_0' = 0$ as the writer receives a fresh update, and $x_1', x_2', \hat{x}'$ are unchanged as the update is not yet written to the memory.

- $l = 2, 5$: The writer finishes writing and publishes a new source update.

  $l = 2$: the writer publishes source 1 update: $x_1' = x_0$ as the age of source 1 update in the memory is reset to just written update. The source 2 update becomes the oldest update in the memory; hence, $\hat{x}' = x_2$.

  $l = 5$: The writer publishes source 2 update: $x_2' = x_0$, the source 1 update becomes the oldest update, and $\hat{x}' = x_1$.

For the SHS analysis, we employ the normalized rates

$$\rho_1 = \lambda_1/\mu, \quad \rho_2 = \lambda_2/\mu. \tag{5.1}$$

We note that $\rho = \rho_1 + \rho_2$ is the total offered load of source updates being written to the memory. The Markov chain in Fig. 5.3.1 has stationary probabilities $\boldsymbol{\pi}$ with normalization constant C_π given by

$$\boldsymbol{\pi} = [\pi_{0_1} \ \pi_1 \ \pi_2 \ \pi_{0_2}] = C_\pi^{-1}[\rho_2/\rho \ \rho_1 \ \rho_2 \ \rho_1/\rho], \tag{5.2a}$$

$$C_\pi = 1 + \rho. \tag{5.2b}$$

With the shorthand notation

$$\lambda = \lambda_1 + \lambda_2, \tag{5.3}$$

we now use Theorem 1 to solve for

$$\bar{\mathbf{v}} = [\bar{\mathbf{v}}_{0_1} \ \bar{\mathbf{v}}_1 \ \bar{\mathbf{v}}_2 \ \bar{\mathbf{v}}_{0_2}], \tag{5.4}$$

where $\mathbf{v}_q = [v_{q0} \ v_{q1} \ v_{q2} \ v_{q3}], \forall q \in \mathcal{Q}$. This yields

$$\lambda\bar{\mathbf{v}}_{0_1} = \mathbf{1}\bar{\pi}_{0_1} + \mu\bar{\mathbf{v}}_2\mathbf{A}_5, \tag{5.5a}$$

$$\mu\bar{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \lambda_1\bar{\mathbf{v}}_{0_1}\mathbf{A}_1 + \lambda_1\bar{\mathbf{v}}_{0_2}\mathbf{A}_3, \tag{5.5b}$$

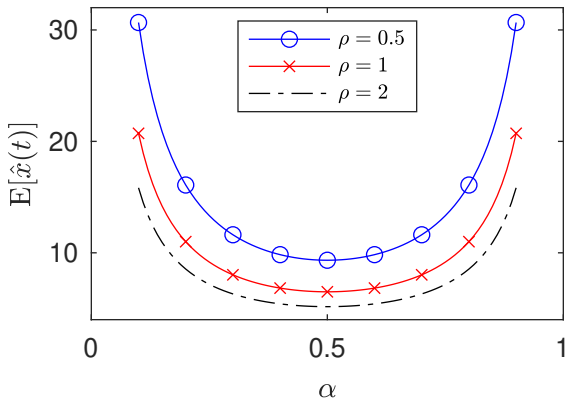


Figure 5.3.2: Average age of max-age process $\hat{x}(t)$ in the memory. For a fixed updating load, we vary α with $\rho_1 = \alpha\rho$ and $\rho_2 = (1-\alpha)\rho$.

$$\mu\bar{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \lambda_2\bar{\mathbf{v}}_{0_1}\mathbf{A}_6 + \lambda_2\bar{\mathbf{v}}_{0_2}\mathbf{A}_4, \tag{5.5c}$$

$$\lambda\bar{\mathbf{v}}_{0_2} = \mathbf{1}\bar{\pi}_{0_2} + \mu\bar{\mathbf{v}}_1\mathbf{A}_2. \tag{5.5d}$$

We can now use Theorem 1 to calculate the AoI of source i update in the memory as $\mathrm{E}[x_i] = v_{0_1,i} + v_{0_2,i} + v_{1,i} + v_{2,i}$ for $i \in \{1,2\}$ as well as $\mathrm{E}[\hat{x}] = v_{0_1,3} + v_{0_2,3} + v_{1,3} + v_{2,3}$. Some algebra yields the following theorem.

Theorem 5.

(a) Source i updates in the memory have average age

$$\mathrm{E}[x_i] = \frac{1}{\mu}\left(\frac{1+\rho}{\rho_i} + \frac{\rho}{1+\rho}\right). \tag{5.6}$$

(b) The max-age process $\hat{x}(t) = \max(x_1(t), x_2(t))$ in the memory has average age

$$\mathrm{E}[\hat{x}] = \frac{(1+\rho)^2(\rho_1^2 + \rho_1\rho_2 + \rho_2^2) + \rho^2\rho_1\rho_2}{\mu\rho(1+\rho)\rho_1\rho_2}. \tag{5.7}$$

Not surprisingly, the expected max-age $\mathrm{E}[\hat{x}]$ is symmetric in the load parameters ρ_1 and ρ_2. However, since the formula (5.7) is somewhat opaque, a plot of $\mathrm{E}[\hat{x}]$ appears in Fig. 5.3.2. A possibly non-obvious observation from the figure is that increasing the overall updating load ρ generally improves

the average max-age because the writer queues no updates. The figure also reveals that the average max-age is penalized by asymmetry in the update rates of the individual sources. This is in part because a source that updates slowly will have high age and thus cause the max-age to be large. However, it is also true that with asymmetric loads, the high rate source will cause updates of the low rate source to be discarded at the writer. Because the writer is non-selective in offering service, it may be performing updates for the high rate source even when the age of that source is already low.

5.4 Age of Decision Updates

In section 5.2.2, we observed that the DP reader is *sampling* the source updates from the memory as a point process. In particular, we assume the inter-sample times $Y_1, Y_2, \ldots$ that are i.i.d continuous random variables identical to Y. In this case, the update sample times form a renewal process, and in the parlance of [32], the update age process $y(t)$ is sampling the max-age update process $\hat{x}(t)$ in the shared memory.

5.4.1 Average Age at the Decision Process

At time t, the most recent read from memory occurred at time $t - Z(t)$. That is, $Z(t)$ is the age of the sampling renewal process. When the renewal process is in equilibrium, $Z(t)$ is stationary with first moment [127, Theorem 5.7.4]

$$\mathrm{E}[Z] = \frac{\mathrm{E}[Y^2]}{2\,\mathrm{E}[Y]}. \tag{5.8}$$

Next, following the approach in [32], we observe that the DP reader does not fetch any update in the interval $(t - Z(t), t]$. Hence, at time t, the update age $y(t)$ satisfies

$$y(t) = \hat{x}(t - Z(t)) + Z(t). \tag{5.9}$$

Further, $Z(t)$ is independent of $\hat{x}(t)$ because the inter-sample times Y_i are independent of the age processes in the shared memory. Thus stationarity of $\mathrm{E}[\hat{x}(t)]$ implies

$$\mathrm{E}[\hat{x}(t - Z(t)] = \mathrm{E}[\hat{x}(t)] = \mathrm{E}[\hat{x}]. \tag{5.10}$$

It then follows from (5.8), (5.9) and (5.10) that $y(t)$ has expected value[2]

$$\mathrm{E}[y] = \mathrm{E}[\hat{x}] + \mathrm{E}[Z] = \mathrm{E}[\hat{x}] + \frac{\mathrm{E}[Y^2]}{2\,\mathrm{E}[Y]}. \tag{5.11}$$

5.4.2 Average Age at the Monitor: Lazy Sampling

When the DP reader samples the shared memory, the DP then computes a decision update based on this sample. On delivery of a decision update to the monitor, the update age $z(t)$ is reset to the age of the oldest source update that was read and used to compute the decision update. This means that an arrival of decision update at the monitor at time t resets $z(t)$ to $y(t)$.

In this work, we assume that the decision computation times are i.i.d continuous random variables $T_1, T_2, \ldots$, each identically distributed to T. We will consider a DP that performs *lazy sampling*: after delivering the computation to the output monitor, the DP reader waits for a random time W before reading again. The alternative to being lazy is the *zero-wait* policy, a special case of lazy when $W = 0$.

Fig. 5.2.1 depicts the evolution of the max-age process $\hat{x}(t) = \max(x_1(t), x_2(t))$, the status-sampling process $y(t)$, and the age at the monitor $z(t)$. with i.i.d inter-sample intervals $Y_1, Y_2, \ldots$ such that samples are taken at times $\tau_i = \sum_{j=1}^{i} Y_j$.

Under lazy sampling, we admit the possibility that the ith computation time T_i and the ith waiting time W_i are correlated. However, in order for the $y(t)$ process to be sampling the shared memory with independent inter-sample times $Y_i = T_i + W_i$, we require that the pairs $(T_1, W_1), (T_2, W_2), \ldots$ to be i.i.d., identical to (T, W). Under this assumption, it follows directly from (5.11) that the

[2]A stronger distributional result is derived in [32, Theorem 6] that is not needed for the average AoI analysis here.

average update age at the input to the DP is

$$\mathrm{E}[y] = \mathrm{E}[\hat{x}] + \frac{\mathrm{E}[(T+W)^2]}{2\,\mathrm{E}[T+W]}. \tag{5.12}$$

Curiously, (5.12) reveals that the problem of minimizing the average age at the input to DP appears to be isomorphic to the timely updating problem that was originally formulated in [128, 52], where the suboptimality of zero-wait policies was first identified. However, in this system, our objective is not to minimize $\mathrm{E}[y]$ but rather to minimize the average age $\mathrm{E}[z]$ at the monitor. Since $z(t)$ is penalized by the waiting time W, choosing W to minimize $\mathrm{E}[y]$ may not be good for $\mathrm{E}[z]$. Fortunately, the following claim verifies this is not the case.

Theorem 6. *If $\hat{x}(t)$ is a stationary process, then for any waiting policy such that W_i depends only on T_i, the average age at the monitor satisfies*

$$\mathrm{E}[z] = \mathrm{E}[y] + \mathrm{E}[T]. \tag{5.13}$$

Proof. Suppose $t \in (\tau_{i-1}, \tau_i]$, the ith inter-sample interval. We observe from Fig. 5.2.1 that $y(\tau_{i-1}) = \hat{x}(\tau_{i-1})$ because the reader fetches update $i-1$ at that time. However, at that time, the monitor has only received the decision update based on update $i-2$, which had age $y(\tau_{i-2})$ at time τ_{i-2} and now, at time $\tau_{i-1} = \tau_{i-2} + Y_{i-1}$, has age $y(\tau_{i-2}) + Y_{i-1}$. Hence, at time τ_{i-1}, the monitor has age

$$z(\tau_{i-1}) = y(\tau_{i-2}) + Y_{i-1}. \tag{5.14}$$

Defining $H_i = y(\tau_{i-2}) + Y_{i-1} - y(\tau_{i-1})$, we can write

$$z(\tau_{i-1}) = y(\tau_{i-1}) + H_i. \tag{5.15}$$

Since $y(\tau_i) = \hat{x}(\tau_i)$ for all i,

$$H_i = \hat{x}(\tau_{i-2}) + Y_{i-1} - \hat{x}(\tau_{i-1}). \tag{5.16}$$

It follows from stationarity of $\hat{x}(t)$ and independence of the sampling times τ_i and $\hat{x}(t)$ that

$$\begin{aligned} \mathrm{E}[H_i] &= \mathrm{E}[\hat{x}(\tau_{i-2})] + \mathrm{E}[Y_{i-1}] - \mathrm{E}[\hat{x}(\tau_{i-1})] \\ &= \mathrm{E}[Y_{i-1}] = \mathrm{E}[T] + \mathrm{E}[W]. \end{aligned} \tag{5.17}$$

At time τ_{i-1}, the ith busy period starts and both $y(t)$ and $z(t)$ grow linearly at rate 1 because neither process sees an update. Hence, $z(t) = y(t) + H_i$ during the busy period. Only when the busy period completes at time $\tau_{i-1} + T_i$ does $z(t)$ drop and become equal to $y(t)$. Let events B_t and I_t correspond to the decision process being busy and idle respectively, at time t. In this interval, the event B_t occurs while $\tau_{i-1} \leq t \leq \tau_{i-1} + T_i$; otherwise I_t occurs if $\tau_{i-1} + T_i \leq t \leq \tau_i$. With these events, we can write

$$z(t) = \begin{cases} y(t) + H_i, & \text{if } B_t, \\ y(t), & \text{if } I_t. \end{cases} \tag{5.18}$$

For $t \geq \tau_{i-1}$, event B_t is independent of H_i, and it follows from the law of total expectation that

$$\begin{aligned} \mathrm{E}[z(t)] &= \mathrm{E}[y(t) + H_i|B]\,\mathrm{P}[B_t] + \mathrm{E}[y(t)|I_t]\,\mathrm{P}[I_t] \\ &= \mathrm{E}[y(t)] + \mathrm{E}[H_i]\,\mathrm{P}[B_t]. \end{aligned} \tag{5.19}$$

In each renewal period, the decision process is busy for time T and then idle for time W. By considering a renewal reward process in which a reward T is earned for the busy period, it follows that the limiting fraction of time spent in a busy state is given by

$$\mathrm{P}[B_t] = \frac{\mathrm{E}[T]}{\mathrm{E}[T] + \mathrm{E}[W]}. \tag{5.20}$$

Applying (5.17) and (5.20) to (5.19) yields the claim. □

We observe that Theorem 6 can give one the mistaken impression that $\mathrm{E}[z]$ is insensitive to the

waiting time W. In fact, the theorem says that the waiting time W affects $\mathrm{E}[y]$ and $\mathrm{E}[z]$ identically. A hand-waving intuition is that $z(t)$ lags $y(t)$ only during the computation time T but, once the computation is complete, $z(t) = y(t)$ during any waiting period.

Combining (5.12) and (5.13), we obtain an end-to-end characterization of the average age in the system:

$$\mathrm{E}[z] = \mathrm{E}[\hat{x}] + \frac{\mathrm{E}[(T+W)^2]}{2\,\mathrm{E}[T+W]} + \mathrm{E}[T]. \tag{5.21}$$

Since the computation time T is given, (5.21) shows that the choice of a waiting function W as a function of T is the same problem formulated in [128, 52]. Hence the solution is the same, namely the β-minimum waiting policy

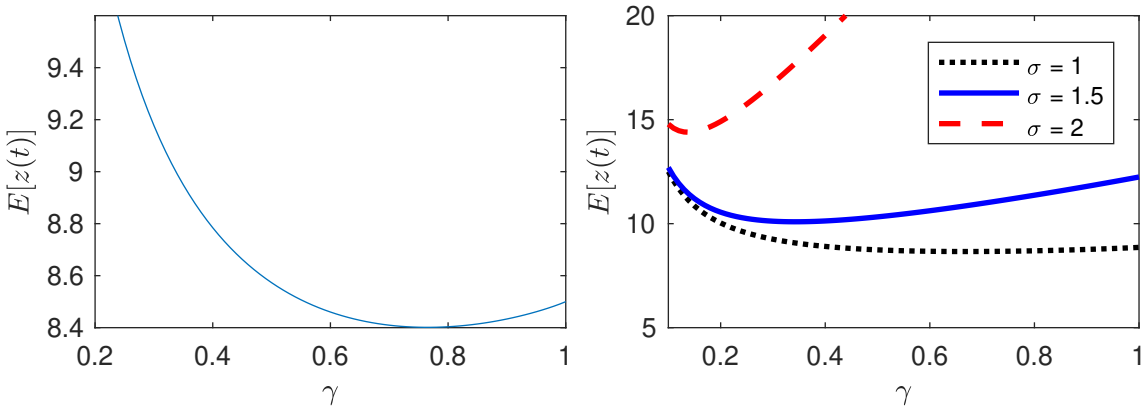
$$W_i = (\beta - T_i)^+, \tag{5.22}$$

where the parameter β is chosen by numerical line search. With this policy, $T + W = \max(\beta, T)$ and it follows from (5.21) that the policy achieves end-to-end average AoI

$$\mathrm{E}[z] = \mathrm{E}[\hat{x}] + \frac{\mathrm{E}[\max(\beta^2, T^2)]}{2\,\mathrm{E}[\max(\beta, T)]} + \mathrm{E}[T]. \tag{5.23}$$

For completeness, the effectiveness of waiting is demonstrated in section 5.5 by some numerical evaluations of the lazy sampling policy. We will see that lazy sampling becomes important when the variance of the computation time T becomes large. Before presenting these results, we comment on the connection of this lazy sampling model to the lazy updating model in [128, 52].

In [128, 52], the random variable T represented the delivery time of a fresh update (say through a network) to the monitor. Fresh updates were generated at will and W represented the waiting time prior to generating the next fresh update. A key element of this system was the tight coupling of waiting and update generation. In this setting, the intuition behind β-minimum waiting was that if the prior delivery time was small, the age at the monitor would be small and it would be a waste of network resources to deliver an update when the age reduction afforded by the update would be



(a) $T \sim \exp(1)$ **(b)** $T \sim$ Log-Normal$(1, e^{\sigma^2} - 1)$

Figure 5.5.1: Average age at the monitor vs the sampling rate γ for the β-minimum policy for different distribution of computation time T. Total offered load by source updates is $\rho = 1$, with $\rho_1 = \rho_2 = 0.5$. Notice that $\gamma = 1$ is the zero-wait computation policy.

small.

In this work, updates are generated by an exogenous process that is beyond the control of the DP. Moreover, because updates are disseminated through a shared memory publication process, the age processes of updates in shared memory are essentially uncoupled from the update sampling/processing policy implemented by the DP. In particular, any time the DP reader fetches a sample pair from the memory, the update age of that pair has expected value $\mathrm{E}[\hat{x}]$, which is just the average age in the shared memory. Nevertheless, even though DP operation is uncoupled from the age process in shared memory, the β-minimum waiting policy is effective. In particular, it reduces the expected value of $y(t)$, the age process at the input to the DP. What is happening is that the waiting policy mitigates the deleterious effect of high-variance computation times T on the sampling policy at the DP reader. We note that Theorem 6 went unrecognized in [128, 52]. Specifically, Theorem 6 shows that no matter what policy is used, the output always lags the input by $\mathrm{E}[T]$ in terms of average age.

5.5 Numerical Evaluation

In this section, we examine some numerical examples of the performance β-minimum waiting policy, simply to remind the reader of the benefits of waiting. Fig. 5.5.1 illustrates age performance with respect to variance in the computation time with probability distributions $\exp(1)$ (Fig. 5.5.1(a)), and Log-Normal$(1, e^{\sigma^2} - 1)$ (Fig. 5.5.1(b)). The log-normal distributed computation times T has PDF [129],

$$f_T(t) = \frac{e^{-(\ln(t)-b)^2/2\sigma^2}}{\sqrt{2\pi}\sigma t}, \qquad t > 0, \tag{5.24}$$

with free parameters b and $\sigma > 0$. In our numerical evaluations, we consider a given distribution on T such that the computation time is normalized to $\mathrm{E}[T] = 1$. In this regard, for Log-Normal distribution, for each σ, we set $b = -\sigma^2/2$ so that $\mathrm{E}[T] = 1$. By varying σ, we vary $\mathrm{Var}[T] = e^{\sigma^2} - 1$. These numerical results are largely similar to those in [128, 52]. In particular, the results remind the reader that zero-wait becomes increasingly sub-optimal when the computation time T has high variance. The choice of β specifies a sampling rate

$$\gamma = \frac{1}{\mathrm{E}[T] + \mathrm{E}[W]} = \frac{1}{\mathrm{E}[\max(\beta, T)]} \tag{5.25}$$

at the DP reader. We then plot the average age at the monitor as a function of γ. Because $\mathrm{E}[T] = 1$, the maximum update sampling rate is $\gamma = 1$, which corresponds to the zero-wait policy. As $\gamma \to 0$, the average age is increasingly dominated by the average inter-read time $1/\gamma$, because updates become too infrequent.

5.6 Conclusion

In this work, we focused on the problem of timely processing of updates from multiple sources. Specifically, we considered a model of a publish-subscribe system where a writer publishes updates from two independent sources in a shared memory and decision updates are derived by a decision

process by reading from the memory. The decision processing works independently of how the source updates are recorded in the memory. Even though the decision processing operates without knowledge of the ages of updates in the shared memory, its reading policy is still able to improve the end-to-end decision update timeliness.

Part III

Synchronization Primitives & AoI

CHAPTER 6
TIMELY MOBILE ROUTING - THEORY

6.1 Introduction

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. 6.1.1. 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.

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. Thus, in this chapter, our

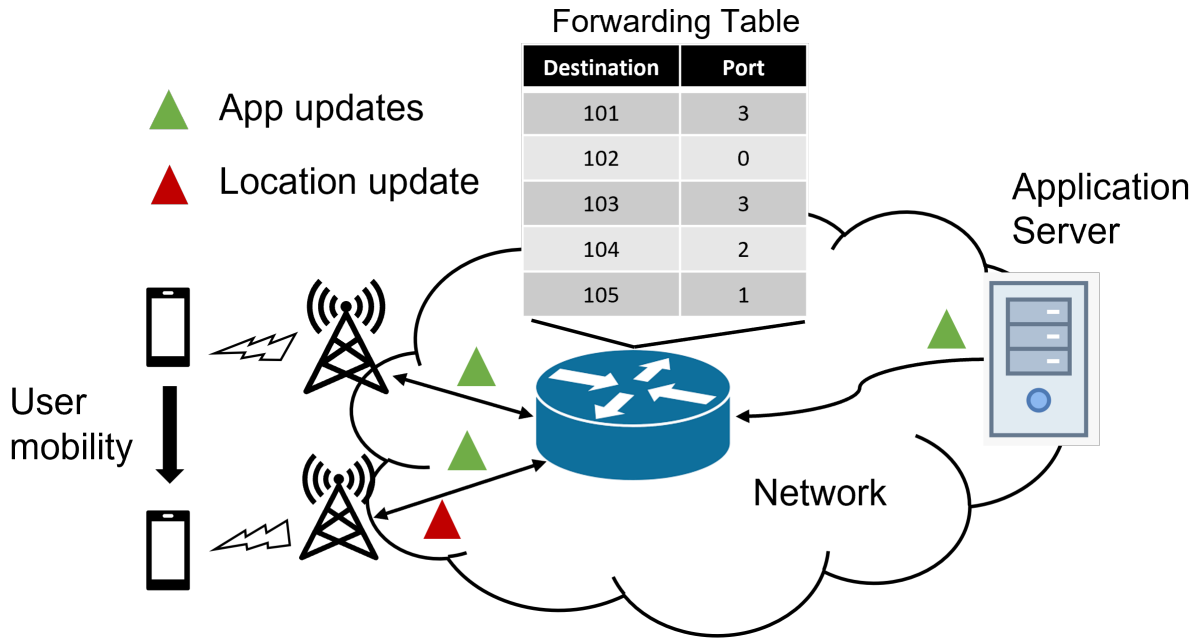


Figure 6.1.1: Packet forwarding application with mobile users

objective is to analyse and compare RCU and RWL when used as synchronization primitives over FIB in terms of their impact on location update and app update age.

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.

6.1.1 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.

6.1.2 Chapter Outline and Contributions

In 6.2, 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 6.1. We show how the app update age process and location update age process are coupled through the FIB. In 6.3 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 lock-less 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.

6.2 AoI Evaluation of App Updates Using SHS

6.2.1 RCU and RWL: SHS Framework

We now describe the SHS framework for RCU and RWL considering the packet forwarding example. An overview of SHS can be found in Chapter 2, Section 2.1. 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.

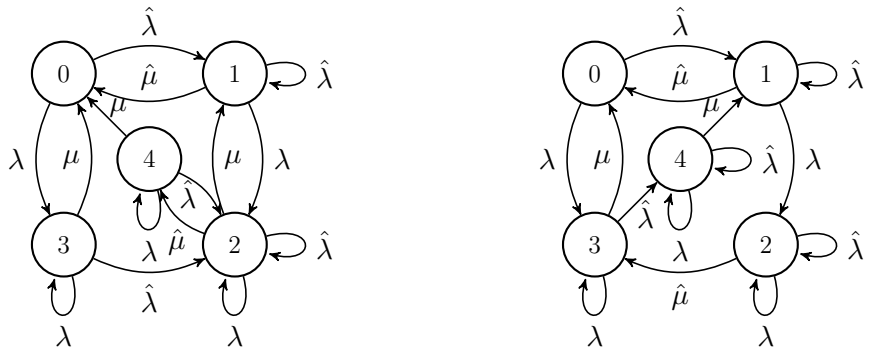
2. The age state of the application update process that initiates client read requests is $\mathbf{x}(t) = [x_0(t)\ x_1(t)]$, where $x_0(t)$ and $x_1(t)$ are the ages of the most recent application updates at the reader and at the mobile terminal (i.e. the destination monitor) respectively.

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.



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

Figure 6.2.1: SHS Markov chain for **(a)** RCU mechanism and for **(b)** RWL mechanism.

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. 6.2.1(a) and 6.2.1(b) respectively. The SHS transition reset maps for RCU and RWL are shown in Tables 6.2.1(a) and 6.2.1(b). In each table, a transition l from state q_l to q_l' occurs at rate $\lambda^{(l)}$ with age reset maps

$$\mathbf{x}' = \mathbf{x}\mathbf{A}_l, \qquad \hat{\mathbf{x}}' = \hat{\mathbf{x}}\hat{\mathbf{A}}_l. \tag{6.1}$$

Equation (6.1) highlights how the the app update process $\mathbf{x}(t)$ and location update process $\hat{\mathbf{x}}(t)$ are coupled only through the state 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.

6.2.2 RCU and RWL: SHS Transitions

Here we describe the SHS transitions for both RCU and RWL that are enumerated in Tables 6.2.1(a) and 6.2.1(b). For each collection of transitions, we focus on the age state process ($\mathbf{x}(t)$ or $\hat{\mathbf{x}}(t)$) that

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 6.2.1: SHS transitions for tracking age in Markov chains of Fig. 6.2.1 for (a) RCU and (b) RWL.

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.

6.2.3 RCU: SHS Age Analysis

For the SHS analysis, we employ the normalized rates

$$\hat{\rho} = \hat{\lambda}/\hat{\mu}, \quad \beta = \lambda/\hat{\mu}, \quad \sigma = \mu/\hat{\mu}. \tag{6.2}$$

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. 6.2.1(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

$$\boldsymbol{\pi} = C_\pi^{-1}[\sigma \ \hat{\rho}\sigma \ \beta\hat{\rho} \ \tfrac{\beta\sigma}{\hat{\rho}+\sigma} \ \tfrac{\beta\hat{\rho}}{\hat{\rho}+\sigma}], \tag{6.3a}$$

$$C_\pi = (1+\hat{\rho})(\beta+\sigma). \tag{6.3b}$$

We now apply Theorem 1 to the SHS reset maps in Table 6.2.1(a). With the shorthand notations

$$\lambda^* = \lambda + \hat{\lambda}, \qquad \mu^* = \mu + \hat{\mu}, \tag{6.4}$$

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

$$\lambda^*\hat{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_6 + \mu\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{13} + \mu\hat{\mathbf{v}}_4\hat{\mathbf{A}}_{14}, \tag{6.5a}$$

$$(\lambda^*+\hat{\mu})\hat{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\hat{\mathbf{v}}_0\hat{\mathbf{A}}_1 + \hat{\lambda}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_2 + \mu\hat{\mathbf{v}}_2\hat{\mathbf{A}}_{15}, \tag{6.5b}$$

$$(\lambda^*+\mu^*)\hat{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_3 + \hat{\lambda}\hat{\mathbf{v}}_3\hat{\mathbf{A}}_4 + \hat{\lambda}\hat{\mathbf{v}}_4\hat{\mathbf{A}}_5 + \lambda\hat{\mathbf{v}}_1\hat{\mathbf{A}}_9 + \lambda\hat{\mathbf{v}}_2\hat{\mathbf{A}}_{10}, \tag{6.5c}$$

$$(\lambda^*+\mu)\hat{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \lambda\hat{\mathbf{v}}_0\hat{\mathbf{A}}_8 + \lambda\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{11}, \tag{6.5d}$$

$$(\lambda^*+\mu)\hat{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\mu}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_7 + \lambda\hat{\mathbf{v}}_4\hat{\mathbf{A}}_{12}. \tag{6.5e}$$

From Table 6.2.1(a) we see that $\hat{\mathbf{A}}_8, \hat{\mathbf{A}}_9, \ldots, \hat{\mathbf{A}}_{15}$ are all identity matrices. It follows that (6.5) simplifies to

$$\lambda^*\hat{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_6 + \mu\hat{\mathbf{v}}_3 + \mu\hat{\mathbf{v}}_4, \tag{6.6a}$$

$$(\lambda^*+\hat{\mu})\hat{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\hat{\mathbf{v}}_0\hat{\mathbf{A}}_1 + \hat{\lambda}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_2 + \mu\hat{\mathbf{v}}_2, \tag{6.6b}$$

$$(\hat{\lambda}+\mu^*)\hat{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_3 + \hat{\lambda}\hat{\mathbf{v}}_3\hat{\mathbf{A}}_4 + \hat{\lambda}\hat{\mathbf{v}}_4\hat{\mathbf{A}}_5 + \lambda\hat{\mathbf{v}}_1, \tag{6.6c}$$

$$(\lambda^*+\mu)\hat{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \lambda\hat{\mathbf{v}}_0 + \lambda\hat{\mathbf{v}}_3, \tag{6.6d}$$

$$(\lambda^* + \mu)\hat{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\mu}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_7 + \lambda\hat{\mathbf{v}}_4. \tag{6.6e}$$

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 (6.5) 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:

$$\lambda^*\bar{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\bar{\mathbf{v}}_1\mathbf{A}_6 + \mu\bar{\mathbf{v}}_3\mathbf{A}_{13} + \mu\bar{\mathbf{v}}_4\mathbf{A}_{14}, \tag{6.7a}$$

$$(\lambda^* + \hat{\mu})\bar{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\bar{\mathbf{v}}_0\mathbf{A}_1 + \hat{\lambda}\bar{\mathbf{v}}_1\mathbf{A}_2 + \mu\bar{\mathbf{v}}_2\mathbf{A}_{15}, \tag{6.7b}$$

$$(\lambda^* + \mu^*)\bar{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\bar{\mathbf{v}}_2\mathbf{A}_3 + \hat{\lambda}\bar{\mathbf{v}}_3\mathbf{A}_4 + \hat{\lambda}\bar{\mathbf{v}}_4\mathbf{A}_5 + \lambda\bar{\mathbf{v}}_1\mathbf{A}_9 + \lambda\bar{\mathbf{v}}_2\mathbf{A}_{10}, \tag{6.7c}$$

$$(\lambda^* + \mu)\bar{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \lambda\bar{\mathbf{v}}_0\mathbf{A}_8 + \lambda\bar{\mathbf{v}}_3\mathbf{A}_{11}, \tag{6.7d}$$

$$(\lambda^* + \mu)\bar{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\mu}\bar{\mathbf{v}}_2\mathbf{A}_7 + \lambda\bar{\mathbf{v}}_4\mathbf{A}_{12}. \tag{6.7e}$$

For these equations, we note from Table 6.2.1(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 (6.6) 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 (6.6) and the simplified version of (6.7) 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

$$\mathrm{E}[\hat{\Delta}] = \sum_{q=0}^{4} \hat{v}_{q1}, \qquad\qquad \mathrm{E}[\Delta] = \sum_{q=0}^{4} \bar{v}_{q1}. \tag{6.8}$$

6.2.4 RWL: SHS Age Analysis

The RWL Markov chain in Fig. 6.2.1(b) has stationary probabilities $\boldsymbol{\pi}$ with normalization constant C_π given by

$$\boldsymbol{\pi} = [\pi_0 \ \pi_1 \ \pi_2 \ \pi_3 \ \pi_4] = C_\pi^{-1} \begin{bmatrix} \sigma(\hat{\rho} + \sigma + \beta\sigma) \\ \hat{\rho}\sigma(\beta + \hat{\rho} + \sigma) \\ \beta\hat{\rho}\sigma(\beta + \hat{\rho} + \sigma) \\ \beta\sigma(1 + \beta + \hat{\rho}) \\ \beta\hat{\rho}(1 + \beta + \hat{\rho}) \end{bmatrix}^\top \tag{6.9a}$$

$$C_\pi = \beta\hat{\rho}(1 + \beta + \hat{\rho}) + \sigma(1 + \beta)(1 + \hat{\rho})(\sigma + \beta + \hat{\rho}). \tag{6.9b}$$

The age balance equations based on the SHS reset maps shown in Table 6.2.1(b) for RWL location updates are:

$$\lambda^* \hat{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \hat{\mu}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_6 + \mu\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{13}, \tag{6.10a}$$

$$(\lambda^* + \hat{\mu})\hat{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \hat{\lambda}\hat{\mathbf{v}}_0\hat{\mathbf{A}}_1 + \hat{\lambda}\hat{\mathbf{v}}_1\hat{\mathbf{A}}_2 + \mu\hat{\mathbf{v}}_4\hat{\mathbf{A}}_{14}, \tag{6.10b}$$

$$(\lambda^* + \hat{\mu})\hat{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \hat{\lambda}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_3 + \lambda\hat{\mathbf{v}}_1\hat{\mathbf{A}}_9 + \lambda\hat{\mathbf{v}}_2\hat{\mathbf{A}}_{10}, \tag{6.10c}$$

$$(\lambda^* + \mu)\hat{\mathbf{v}}_3 = \mathbf{1}\bar{\pi}_3 + \hat{\mu}\hat{\mathbf{v}}_2\hat{\mathbf{A}}_7 + \lambda\hat{\mathbf{v}}_0\hat{\mathbf{A}}_8 + \lambda\hat{\mathbf{v}}_3\hat{\mathbf{A}}_{11}, \tag{6.10d}$$

$$(\lambda^* + \mu)\hat{\mathbf{v}}_4 = \mathbf{1}\bar{\pi}_4 + \hat{\lambda}\bar{\mathbf{v}}_3\hat{\mathbf{A}}_4 + \hat{\lambda}\bar{\mathbf{v}}_4\hat{\mathbf{A}}_5 + \lambda\bar{\mathbf{v}}_4\hat{\mathbf{A}}_{12}. \tag{6.10e}$$

For the app updates described by the age process $\mathbf{x}(t)$, there is a set of SHS equations identical to (6.10), 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 (6.8).

The SHS analysis of RCU and RWL corresponding to Equations (6.3) through (6.10) 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 the SHS models of these non-preemptive systems, the states of Markov chains in Fig. 6.2.1 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 6.2.1 except transitions $l = 10, 11, 12$ in both Table 6.2.1(a) and Table 6.2.1(b) are deleted. The age balance equations then can be obtained from Theorem 1 in the same way that we derived (6.5). We don't explicitly enumerate these age balance equations here; however, in section 6.3, we numerically compare the performance of preemptive and non-preemptive systems.

6.3 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 [130]. 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

$$\hat{\mu}_{\text{RWL}} = \hat{\mu}_{\text{RCU}} = \hat{\mu}. \tag{6.11}$$

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 [42]. 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

$$\hat{\rho} = \frac{\hat{\lambda}}{\hat{\mu}}, \quad \beta = \frac{\lambda}{\hat{\mu}}, \quad \sigma_{\text{RWL}} = \frac{\mu_{\text{RWL}}}{\hat{\mu}}, \quad \sigma_{\text{RCU}} = \frac{\mu_{\text{RCU}}}{\hat{\mu}}, \tag{6.12}$$

we now present some results from numerically evaluating (6.6), (6.7), (6.10) 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. 6.3.1, 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. 6.3.1(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. 6.3.1(b). In this case, larger

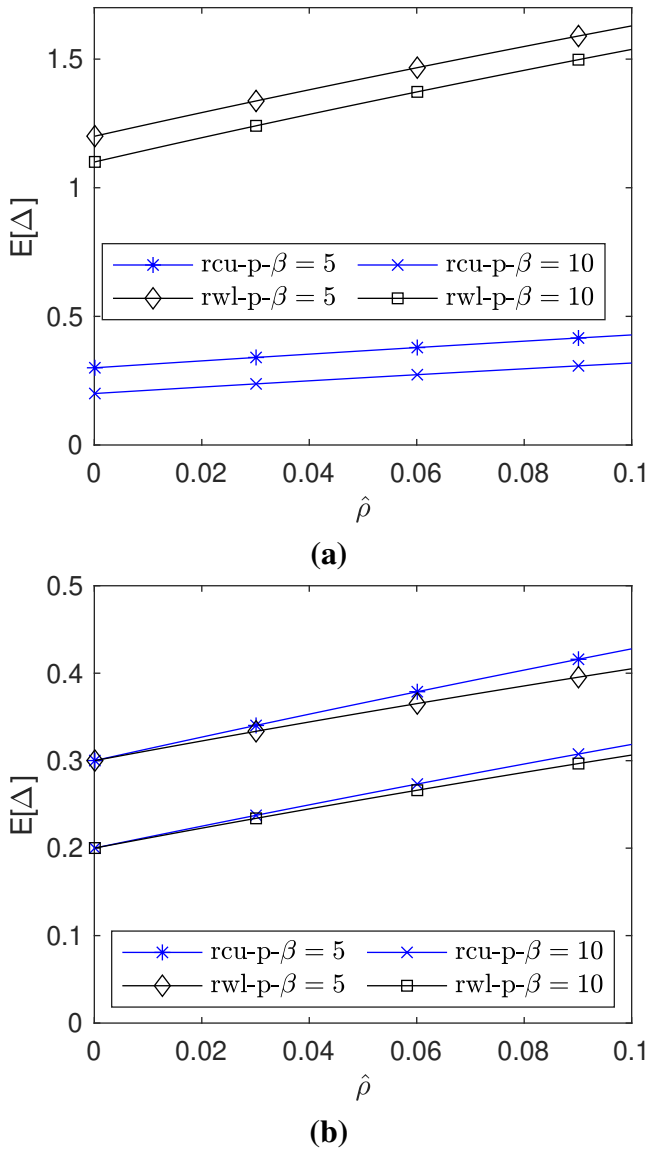


Figure 6.3.1: AoI at mobile client when using RCU preemption (rcu -p) and RWL preemption (rwl-p) as a function of normalized write request rate $\hat{\rho} = \hat{\lambda}/\hat{\mu}$, against different values of normalized read rate $\beta = \lambda/\hat{\mu}$ and $\sigma_{\text{RCU}} = 10$ with **(a)** $\sigma_{\text{RWL}} = 1$, and **(b)** $\sigma_{\text{RWL}} = 10$.

$\hat{\rho}$ 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. 6.2.1, 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 in each of these states. Thus, the probability that an app update is delivered under RCU is

$$P_{\text{RCU}} = (\pi_0 + \pi_3)\frac{\mu}{\lambda^* + \mu}. \tag{6.13}$$

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

$$P_{\text{RWL}} = (\pi_0 + \pi_1\frac{\hat{\mu}}{\hat{\mu} + \lambda} + \pi_2\frac{\hat{\mu}}{\hat{\mu} + \lambda} + \pi_3)\frac{\mu}{\lambda^* + \mu}. \tag{6.14}$$

Fig. 6.3.2 shows that P_{RCU} and P_{RWL} in (6.13) and (6.14) decrease as a function of normalized write request rate. In comparing Figs 6.3.1 and 6.3.2, we see that for both RCU and RWL that the average age $\mathrm{E}[\Delta]$ becomes worse as the delivery probability decreases.

Fig. 6.3.3 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. 6.3.3 that preemption mechanisms generally reduce AoI.

In Fig. 6.3.4, we observe that the age of location updates in the memory is $\mathrm{E}[\hat{\Delta}] \approx 1/\hat{\rho}$ for both RCU and RWL. This demonstrates that essentially all location updates are promptly stored in memory and that $\mathrm{E}[\hat{\Delta}]$ 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

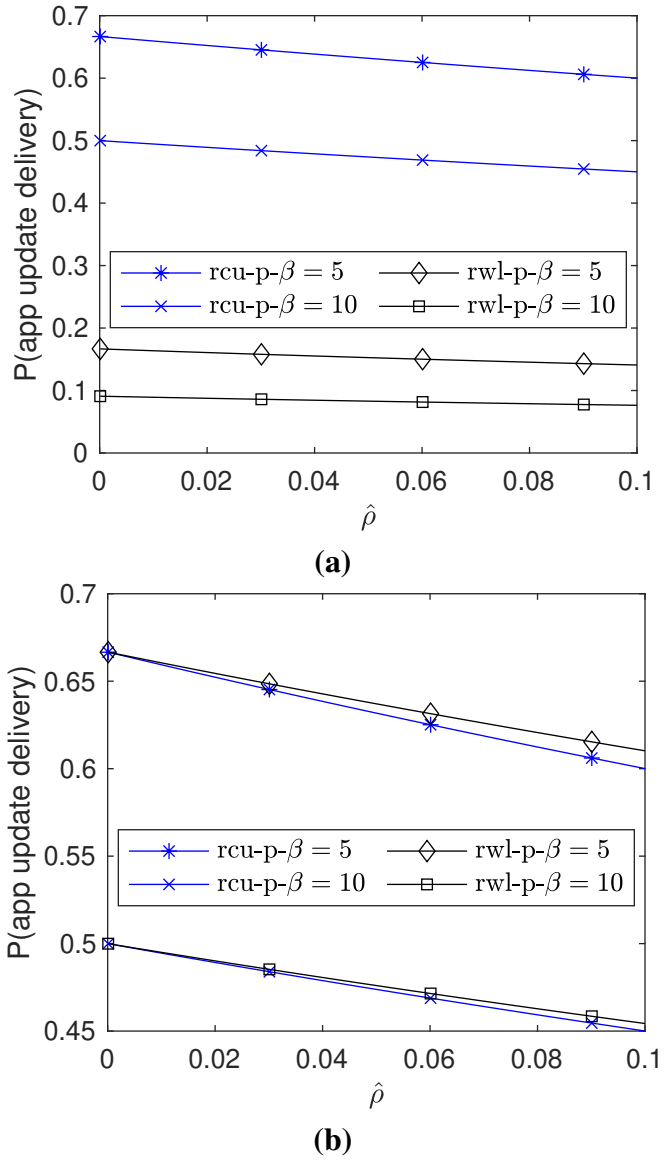


Figure 6.3.2: Probability that an app update arriving at router is delivered correctly when $\sigma_{\text{RCU}} = 10$ and when **(a)** $\sigma_{\text{RWL}} = 1$ and **(b)** $\sigma_{\text{RWL}} = 10$.

of location updates would be better described using metrics such as Age of Incorrect Information [131] or Age of Synchronization [132, 62] that account for whether the current update is correct.

6.4 Conclusion

This work 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

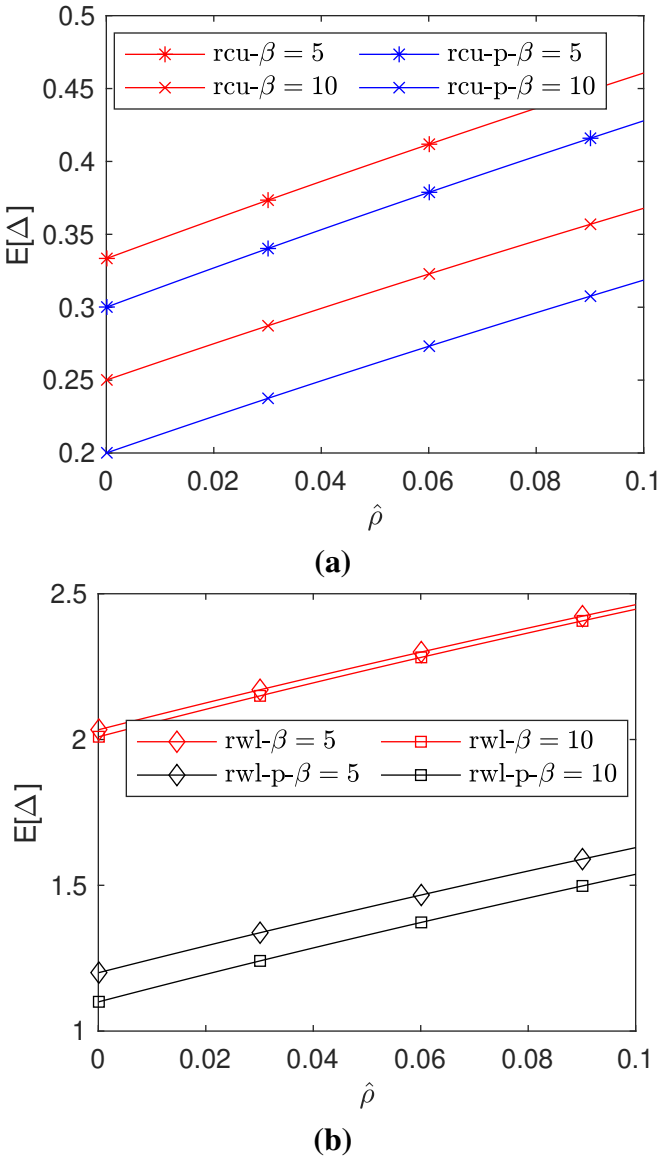


Figure 6.3.3: AoI performance with and without preemption for **(a)** RCU with $\sigma_{\text{RCU}} = 10$, and **(b)** RWL with $\sigma_{\text{RWL}} = 1$.

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 resulting in increased age at the mobile terminal.

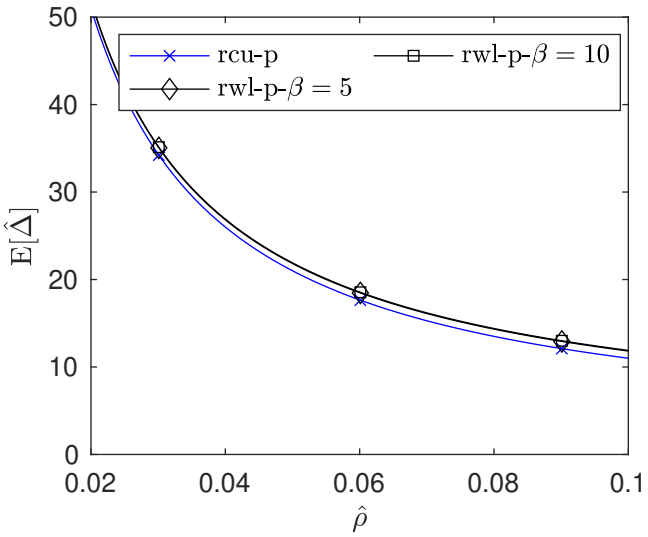


Figure 6.3.4: AoI at memory when $\sigma_{\mathrm{RCU}} = 10$ and $\sigma_{\mathrm{RWL}} = 1$.

CHAPTER 7

TIMELY MOBILE ROUTING - AN EXPERIMENTAL STUDY

7.1 Introduction

In this work, we design and implement a packet forwarding application as introduced in Section 6.1 on a high-speed testbed employing the Data Plane Development Kit (DPDK)[1] [133] packet processing framework and use Age of Information (AoI) metric to evaluate two key issues that influence timeliness: 1. batch packet admission procedures of DPDK that cause input queueing, and 2. synchronization primitives that regulate concurrent access to the Forwarding Information Base (FIB). We now present these two challenges in detail.

7.1.1 Impact of Input Queueing

From the outset of AoI analysis of updating systems, the value of "bufferless" mechanisms that discard old updates and/or give priority to fresher updates has been recognized [3, 134]. In the practical context of DPDK, this would correspond to DPDK input rings that hold just a single packet. However, because DPDK aims to maximize packet throughout, it uses large rings to absorb traffic bursts and does not support small buffer configurations. While this helps to avoid packet dropping (and consequent TCP retransmissions), larger buffers will also contribute to buffer bloat latency. In our testbed evaluation of AoI for feasible DPDK configurations (section 7.3), we will see that age can increase with offered load because the update packets become stale while queued in the ring. Additionally, we will observe how DPDK employing batch processing to increase packet throughout penalizes the timeliness of update packets.

[1] DPDK is an open-source software project managed by the Linux Foundation and is widely used in data centers and core routers.

7.1.2 Impact of Synchronization Primitives

As observed in Chapter 6 (Section 6.1), the FIB is typically implemented as a concurrent hash table in which the contention between readers and writers is usually handled either by RWL or RCU. Both RCU and RWL-based data structures have been bench-marked using stand-alone stress tests [103, 104, 105, 106, 108, 109, 110], but, when used and implemented in the networking stack, their performance raises many concerns and questions [135, 136]. For example, authors in [135] studied RCU and RWL and argued that lock contention in a Memcached application accounts for around a third of the overall kernel overhead and significantly contributes to delay and latency variation.

In the context of timeliness, the age performance of RWL and RCU needs to be better understood in practical systems, especially when these are used in fundamental networking data structures. The aim of this chapter is, thus, to quantitatively understand and analyze the impact of RCU and RWL on the timely updating of shared memory and how this, in turn, affects timely routing of information updates. To our knowledge, this is the first quantitative experimental study of the two widely used synchronization primitives concerning the AoI performance metric.

In section 7.3, we show that our AoI experimental results are consistent with the literature that RCU takes advantage of its light read-side primitives to generally outperform RWL. However, at low packet sending rates, this difference is negligible. The caveat of using RCU, however, is that each write makes a copy of the shared-object, which means that the memory footprint of the code is larger and also requires a complex garbage collection mechanism [102].

7.2 Experiment Design and Testbed

This section describes the experimental setup used to evaluate the layer 2 packet forwarding application depicted in Fig. 6.1.1. We note that AoI evaluation requires time-based computations across machines. Since the age of an update packet is based on a timestamp inserted by the sender, calculation of the age of an update at the receiver requires synchronized clocks at the sender and receiver. However, the accuracy of the NTP protocol supported by the testbed is around 1ms, which

is too coarse to measure delays on a microsecond scale.[2] Our experimental workaround is to place the sender and receiver functionality on the same machine.

Fig. 7.2.1 shows a block diagram of a testbed architecture with two machines (Source and Forwarder) that implements the system shown in Figure 6.1.1. The sender thread in the Source acts as the application server that sends time-stamped app updates (data packets) to multiple mobile users. This sender thread also emulates mobile user movement by sending time-stamped location updates (control packets). Each update carries a user ID indicating a location change for that user. The receive thread in the Source acts as the mobile users receiving app update packets. Each time-stamped update carries a user ID that enables the Source to track the app update age process of each user. Although the scale of the experiment is small, using only two machines allows us to focus on the primary bottlenecks we have identified: compute bottleneck due to synchronization primitives, and queuing bottleneck at the sender.

At the Forwarder, the FIB is implemented as a hash table for fast lookup. The destination user ID acts as a key, and the hash function translates this key into a hash index that points to an *address tuple*. This address tuple consists of a MAC address and a timestamp as shown in Fig. 7.2.1. A traditional FIB would store a next hop MAC address for a destination user ID and update this MAC address to reflect a new point of attachment for the mobile user. Since the next hop MAC address (Source machine) is same for all users in our testbed implementation, old and new address entries in the FIB are distinguished by the timestamp in the address tuple. Upon receiving a new control packet from the Source, the control process in the Forwarder updates the timestamp in the address tuple for the corresponding user ID.

To ensure reproducibility of the experiments, we use a trace file that consists of rows of type *<type, userID>*, where a type indicates whether a packet sent from the Source is a control or data packet. The order of packet types in the trace is decided by a pseudo-random sequence of coin flips such that the control data ratio (CDR) parameter specifies the ratio of control and data packets.

The user IDs for both data and control packets are selected from a Zipf distribution with exponent

[2]We note this is merely a limitation of our testbed. Sub-microsecond timing accuracy is feasible, although not generally used in network routers [137].

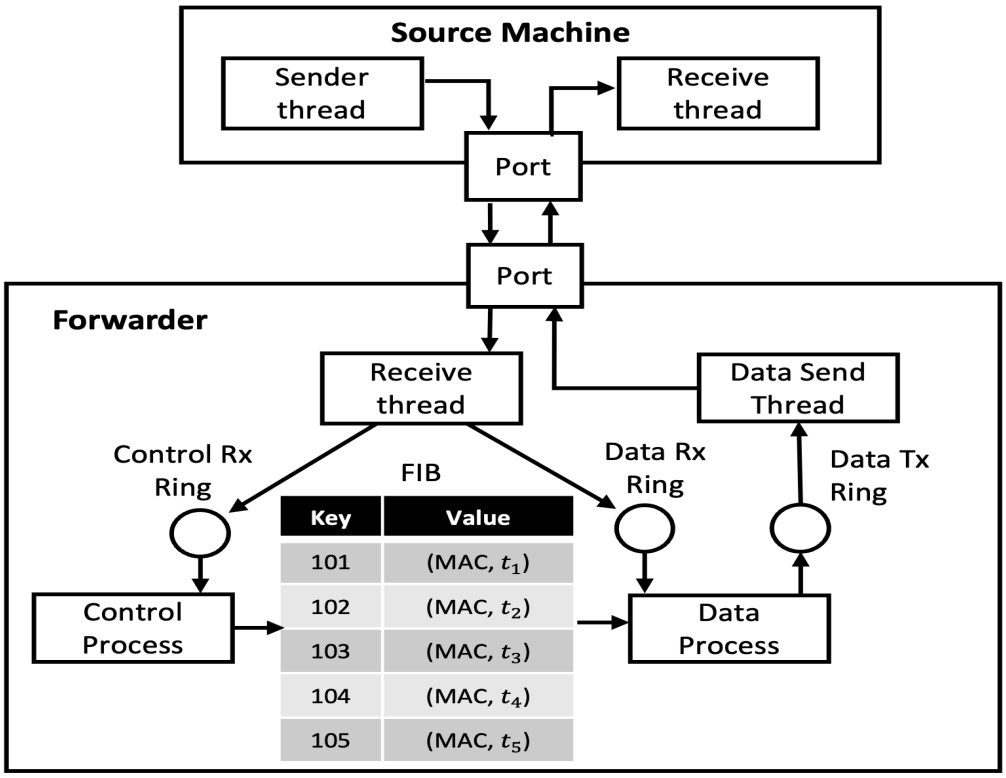


Figure 7.2.1: Packet forwarding testbed: The Source machine emulates the app update sender and receivers as well as their location update senders. In the Forwarder, the FIB stores the key-value pair as user ID (101, 102, . . .) and address tuple while the control and data processes contend for FIB access.

1 on a set of 1000 user addresses. To eliminate randomness of the packet preparation time, packets are read from the trace file and stored in the memory of the sender thread of the Source before the experiment is run. The sender thread sends both control and data packets from the same interface due to hardware and software limitations and to enforce a packet trace sequence.

At the Forwarder, the receive thread pulls packets from the receive ring of the NIC and moves control and data packets to their respective Rx rings. The control process retrieves the control packets from the control Rx ring and updates the FIB, thus acting as a *writer*. The data process retrieves data packets from the data Rx ring and, acting as the FIB *reader*, addresses each data packet via a lookup of the user ID carried by the packet. Each data process read of the FIB returns with an address tuple. The corresponding timestamp in the tuple is inserted in the header of the data packet which is then sent to Data Tx Ring. These modified data packets, which represent app

updates, are then sent back to the Source.

For each user ID, an address tuple is said to be *fresh* if its timestamp is the same as that of the last sent address update (control packet) from the Source. That is, an address is fresh if the mobile user has not sent a subsequent location update. In our experiments, the freshness of the address determines the status of a data packet received by receive thread. Specifically, received data packets with a fresh address are classified as correctly received app updates and serve to reduce the app update age of the corresponding mobile user. On the other hand, a received data packet with an address that is not fresh is classified as misaddressed and regarded as lost in transit.

We note that buffer overflow events (i.e. packet drops) occur at the control (data) Rx ring when the control (data) process at the Forwarder fails to keep up with its incoming packet stream. A dropped control packet signifies that a user movement has not been updated in the FIB, resulting in misaddressed data packets. A dropped data packet indicates that an app update has not been received at the mobile user, hence increasing that user's app update age.

7.3 Testbed Results

Experiments are executed on the COSMOS experimental networking testbed [24]. We employ DPDK, a set of data plane libraries and network interface card (NIC) drivers to support fast packet processing in user space [133]. The machines use Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz (24 cores, hyper-threading and turbo-boost turned off) with 192GB RAM evenly distributed on 2 NUMA nodes. Each thread in our experiment is pinned to each core in a single NUMA node which in turn is pinned to a single socket. A Mellanox ConnectX-4 Lx 25GbE network interface card is connected to NUMA node 0. We run our program on Ubuntu 18.04.6 LTS, and DPDK 21.08. The Source and Forwarder are connected via a dedicated switch with 25Gbps Ethernet links.

Table 7.3.1 summarizes three sets of experiments performed in this study. In the testbed, data and control packets are both 60 bytes long and the Source can pump these packets at a maximum rate of $R_{\max} = 14$ Mpps (million packets per second). In our experiments, our results are shown as a function of the offered load R pps. Specifically, the sender thread in the Source feeds the

Experiment	Data pkts	Ctrl pkts	Users	FIB lookup
Baseline	47996440	0	1	No
Routing, CDR 0.01	39996600	400000	1000	Yes
Routing, CDR 0.1	39996600	3999800	1000	Yes

Table 7.3.1: All experiments share the following DPDK configurations: (1) Source-Tx burst size 32, Tx ring size 64, Rx burst size 64, Rx ring size 4096. (2) Forwarder-Tx/Rx burst size 64, Tx/Rx ring size 4096.

pre-prepared packet trace using a token bucket rate control mechanism and a maximum data burst size of B packets (in our experiments, $B = 32$). At time $t_0 = 0$, the bucket is initialized with $N_0 = 0$ tokens and tokens then accumulate at a rate of R tokens/s. The sender thread requests to place packets on the Tx ring of the Source NIC at times $t_0, t_1, \ldots$ such that at time t_i with N_i tokens, the sender thread calls the *eth_tx_burst* function to offer $K_i = \min(B, N_i)$ packets to the NIC. At time t_{i+1}, *eth_tx_burst* returns that a *batch* of L_i packets were admitted to the NIC and thus $N_{i+1} = N_i - L_i + (t_{i+1} - t_i)R$ tokens are available to repeat this process until the entire packet trace is admitted to the Source NIC.

Since no packets are dropped at the Source Tx ring, $L_i < K_i$ indicates that *eth_tx_burst* call filled the ring. Also note that while the *eth_tx_burst* execution time $\tau_i = t_{i+1} - t_i$ is random[3], the process self-adjusts to offer packets at rate R pps for all $R < R_{\max}$. Finally, we note that hardware limitations dictate that all L_i packets in a batch are recorded with the same timestamp t_i, and this timestamp is inserted by the CPU. While this sending process is not the usual Poisson update process in analytical studies, it does offer a repeatable characterization of AoI performance under variable offered load.

We note that high-speed packet IO such as DPDK use large batches by default, leading to a trade-off between bursty high throughput and precise packet generation. Cases where users might not require a bursty traffic, such as in generating Poisson stream of packets are difficult to emulate reliably, especially at high sending rates. Packet data cannot be directly sent to the NIC, but can be placed in a DMA memory region and retrieved asynchronously by the NIC, causing unwanted jitter [138]. Further, calling the eth_tx_burst() function to place packets on the output ring takes some

[3]We observe that τ_i and L_i appear to be weakly but positively correlated.

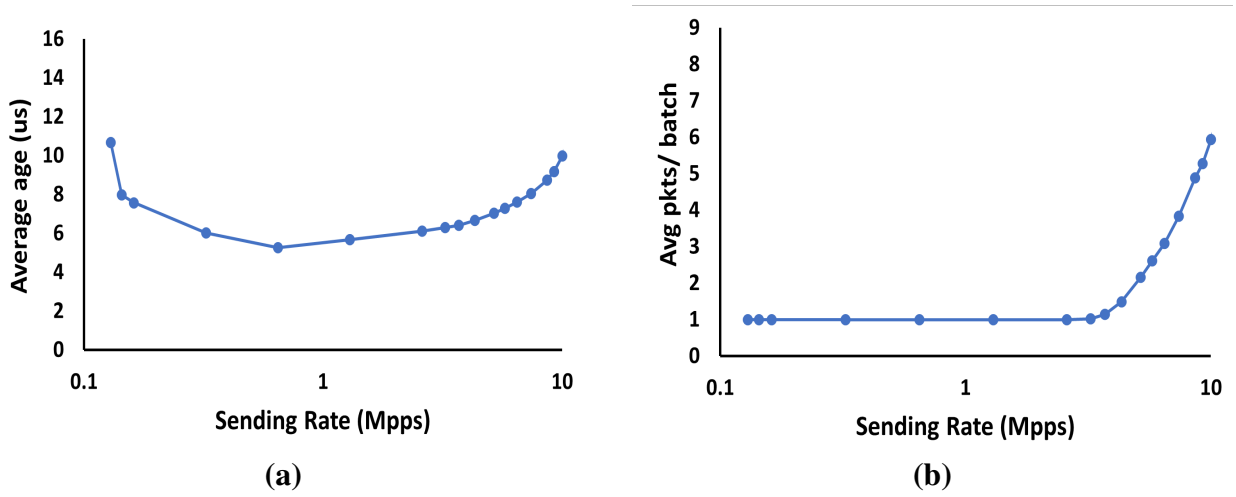


Figure 7.3.1: Baseline experiment

random time to return, which also depends on the number of packets offered.

For Poisson packet arrivals, a pure software approach would wait for pre-configured pseudorandom times between sending individual packets. Implementing a close approximation to exponential inter-arrivals is not a problem at low packet rates where the aforementioned random system delays are negligible compared to inter-packet times. However, Poisson arrival emulation becomes difficult at rates approaching the limitations of the testbed hardware and the software framework running on it. When packet delays from the system are a significant fraction of the average inter-arrival time, the precision of Poisson traffic pattern remains a concern.

7.3.1 Baseline Experiment

To understand the rate control mechanism, we performed a baseline experiment with a single immobile user. No control packets were sent and the FIB addressing mechanism at the Forwarder was bypassed so that all packets were immediately sent back to the Source. In Fig. 7.3.1(a), we see that for $R < 1$ Mpps, the average age initially declines with R (as expected) because updates become less infrequent. However, perhaps unexpectedly, we see that as R becomes large, the average age grows. This is a consequence of rate control. Fig. 7.3.1(b) presents the average batch size for all nonzero batch sizes. For $R < 4$ Mpps, tokens accumulate slowly and each *eth_tx_burst* call offers either zero or one packet. Thus each admitted batch has only a single packet. However,

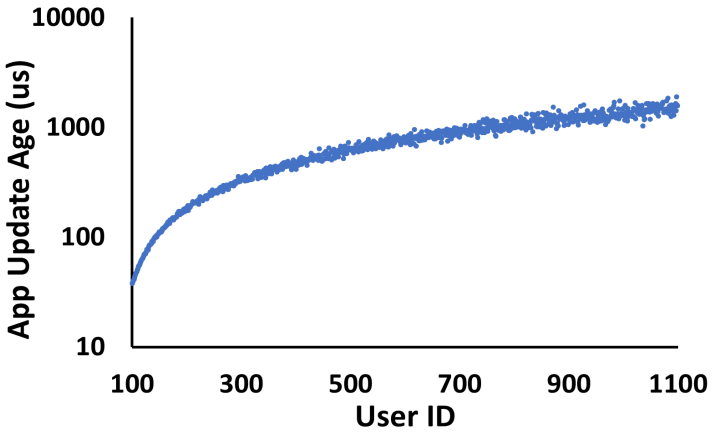


Figure 7.3.2: Average app update age for each user for sending rate 10 Mpps when RWL is used.

for $R > 4$ Mpps, the average batch size grows with R, and the growing average age reflects the input queueing induced by the batch admission procedure. In short, processing more packets in a batch increases throughput, but this is not necessarily favorable to timeliness.

7.3.2 Routing Experiments

We now examine the effect of the FIB access mechanism (RCU or RWL) on the average age of app updates for a set of 1000 mobile users. In these experiments, the packet stream, with sending rates ranging from $R = 1$ to $R = 10$ Mpps, represents the aggregated updating processes of all 1000 users. Compared to the prior baseline experiment with one user, the packet sending rates associated with a particular user are scaled by the Zipf distribution probabilities. Average update ages of 1-10 μs seen in the one-user baseline experiment become 5-10 ms in averaging over all 1000 users; Fig. 7.3.2 shows the average app update age for each user for $R = 10$ Mpps under the RWL construct.

A subset of experimental results is shown in Fig. 7.3.3 for CDR $= 0.01$ and CDR $= 0.1$. We note that increasing the CDR stresses the system in two ways. First, it increases FIB access contention between readers and the writer. Second, increasing the CDR corresponds to increased change in mobile users location, thus, increasing the likelihood that app updates are misaddressed.

In Figs. 7.3.3(a)-(b) we see for CDR $= 0.01$ that the Forwarder performs reasonably well at

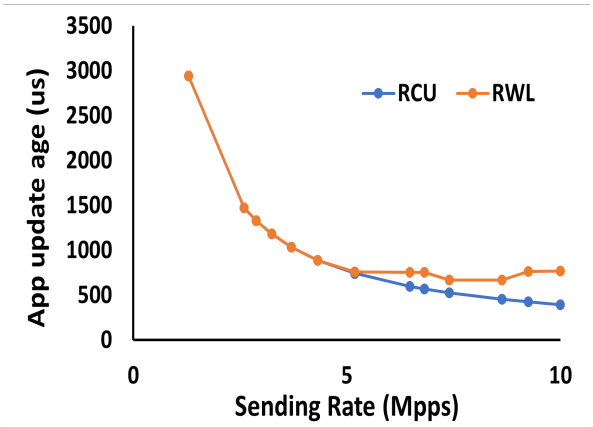
all packet rates. Fig. 7.3.3(a) shows the average app update age generally decreasing with the packet sending rate, with RCU outperforming RWL at high packet rates. This is consistent with relatively low rates of dropped data packets in Fig. 7.3.3(b). However, at higher sending rates, mutual exclusion between reads and write in RWL slows the data process to handle the incoming packets on the data Rx ring of the forwarder, resulting in increased data packet drops at higher sending rates, see Fig. 7.3.3(b).

For CDR $= 0.1$, Figs. 7.3.3(c)-(e) reflect the increased stress of a high CDR. In particular, the results reinforce the fact that neither RCU nor RWL is good in update-heavy scenarios as RCU writes are heavy and RWL enforces mutual exclusion. These mechanisms effectively slow the control processing so that the control Rx ring is quickly filled at a higher control packet rates and the dropping rate of control packets is high. As a consequence, the FIB is updated with stale control updates, increasing the rate of misaddressed data packets, as seen in Fig. 7.3.3(e), and increasing the app update age in Fig. 7.3.3(c), as compared to CDR $= 0.01$. The number of misaddressed packets was calculated based on the total number of received packets on Source machine that carried timestamp older than the last sent control timestamp for that user ID. Note that in Fig. 7.3.3(e), we plot misaddressed packets only for $R < 4$ Mpps. This is because at $R > 4$ Mpps, we observed significant data packet drops at the Forwarder, resulting in fewer received packets at the Sender. Calculating misaddressed packets based on these fewer packets gave us statistically incorrect number.

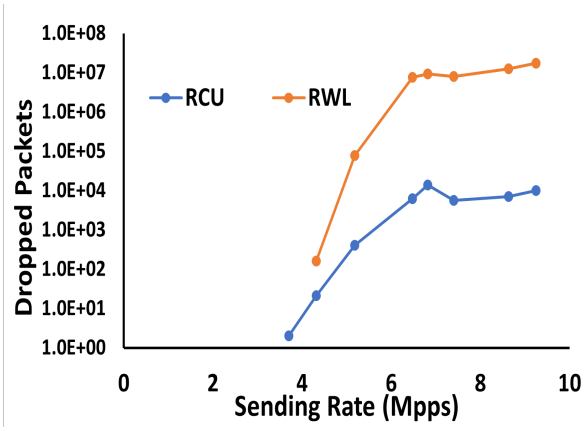
RCU is read-friendly, as is evident in Fig. 7.3.3(d), with fewer data packet drops. By contrast, RWL reads are frequently locked out at higher write request rates, This slows data packet processing under RWL and increases the data packet drops at the data Rx ring. Consequently, Fig. 7.3.3(c) shows that the average age under RWL follows the classic pattern of updating systems: age initially decreases with the update rate but eventually increases as the system becomes congested [3]. In short, updating should be fast, but not too fast.

7.4 Conclusion

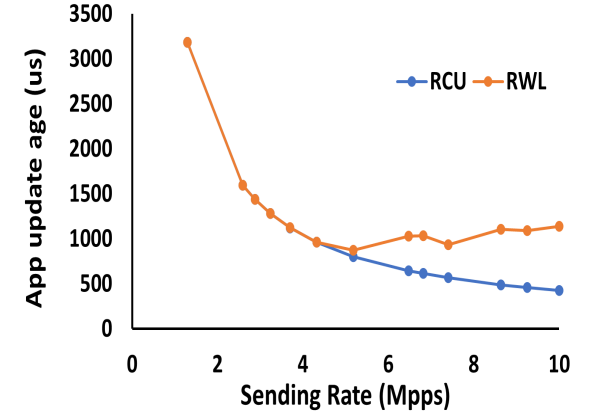
In this work, we designed and implemented a DPDK-based packet forwarding experiment. We quantitatively evaluated the performance of the Readers-Writer Lock (RWL) and (lock-less) Read-Copy-Update (RCU) synchronization primitives, in terms of the Age-of-Information (AoI) performance metric. Even in a relatively simple one-forwarder system, this initial study revealed complex interactions between FIB synchronization mechanisms and packet queueing. This work highlights how more work is needed on optimizing packet processing frameworks such as DPDK for updating systems.



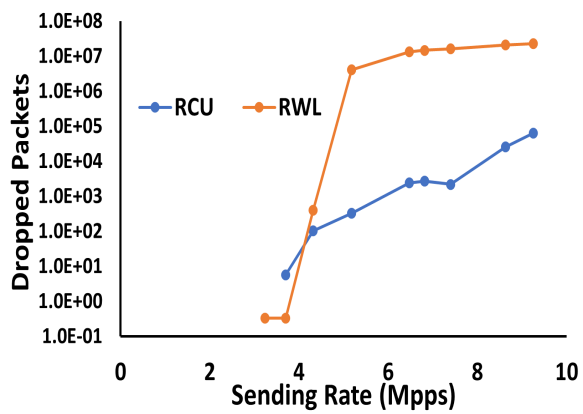
(a) **CDR** $= 0.01$



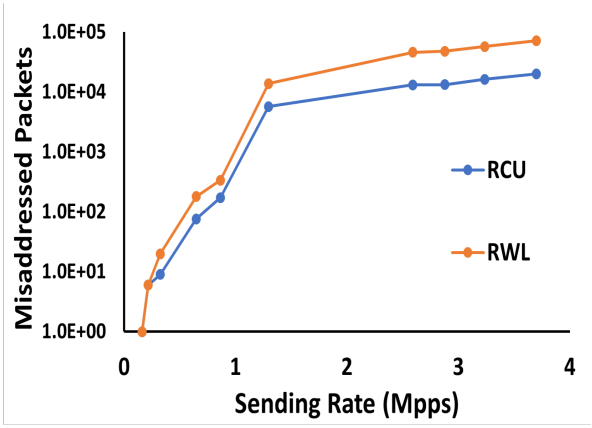
(b) **CDR** $= 0.01$



(c) **CDR** $= 0.1$



(d) **CDR** $= 0.1$



(e) **CDR** $= 0.1$

Figure 7.3.3: Results from packet forwarding testbed (Fig. 7.2.1) with control/data Rx ring sizes $= 64$, data tx ring size $= 1024$. The plots depict age performance of Read-Copy-Update (RCU) and Readers-Writer Lock (RWL) as a function of the sending rate R Mpps.

CHAPTER 8
AGE-MEMORY TRADE-OFF IN RCU

8.1 Introduction

Consider a scenario of a Visual Simultaneous Localization and Mapping (SLAM) system [139], which constructs a map of an environment in real-time while simultaneously determining the location of a mobile device within that map. For seamless interaction with the real world, it is desirable to run SLAM systems on mobile phones. In a typical SLAM workflow, incoming images are processed to track the device's location, and this location information is then incorporated into a global map, with ongoing optimization of the map structure. For timely accuracy, SLAM systems must promptly process incoming camera streams, accessing the latest images in real-time. Although SLAM systems adopt a modular approach with concurrent modules handling specific tasks such as image processing, location tracking, map updating, and global map optimization, there is a tight coupling between modules. All modules operate on the global map, implemented as a shared data structure, and engage in computationally intensive operations, frequently accessing and updating the map [140, 141].

RCU is well-suited to applications such as Visual SLAM because it enables a module to read the freshest copy of a data item. When a module performs a complex operation, it places a read-lock on a data item to ensure it will be available and unchanged during the read operation. When the RCU writer wishes to update the data item, the write operation creates a fresher version/copy. As soon as the write is committed, this fresher copy is returned to subsequent read requests. However, each prior copy is retained in memory until all of its read-locks have been released. Therefore, from a timeliness perspective, more frequent updating of data items in the memory provides the latest information to readers but this results in memory overhead by increasing the number of data copies created.

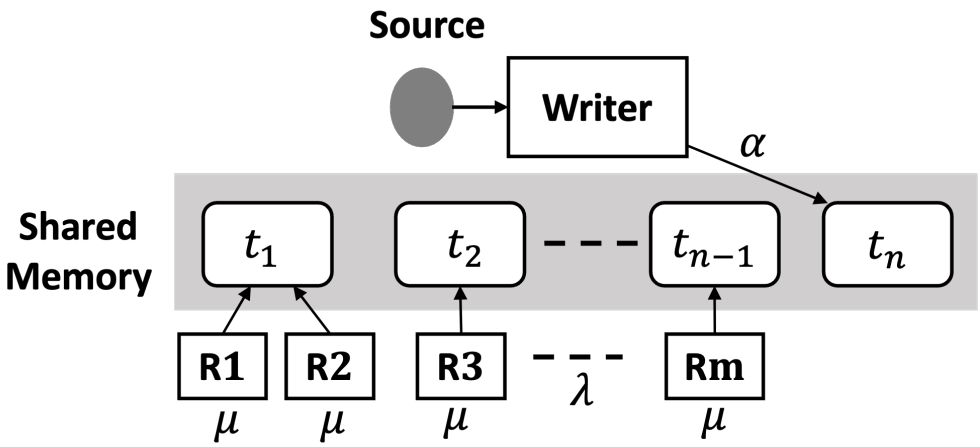


Figure 8.2.1: *Memoryless RCU model:* On behalf of an external source, a writer updates the shared memory at rate α with timestamped updates, denoted by timestamps $t_1, t_2, \ldots$. Read requests $R_1, R_2, \ldots, R_m$ access the version of the source update with the freshest timestamp. These read requests are generated at rate λ and have a mean read time of $1/\mu$.

While Visual SLAM serves as an illustrative example, the broader motivation in this chapter is to explore the trade-off between memory usage and update age in real-time systems. RCU, as a widely used synchronization primitive, is the focal point of our study in the following ways: 1. We investigate the memory footprint of concurrent updates in RCU and provide an upper bound on the average number of active[1] updates in the system, and 2. we analytically explore a trade-off between memory footprint and the age of updates, particularly in case of unbounded number of concurrent updates.

8.2 System Model and Main Results

In this work, we focus on a class of systems (see Fig. 8.2.1) in which a source generates time-stamped *updates*, which are stored in shared memory. The writer queries this source for fresh measurements to update the memory, creating a new copy therein. Concurrently, a reader serves clients' requests for these measurements by accessing the memory. Multiple 'old' readers may concurrently access distinct versions of data copies, depending upon the time of their request. It is noteworthy that the most recent read request will consistently retrieve the latest update from the

[1] An update is active if there is at least one reader reading that update.

memory.

To analyze RCU, we assume the writer starts writing a fresh update as soon as it finishes its previous write, without regard for the number of update copies in the grace period. With respect to memory consumption (i.e. the number of copies created), this is a worst-case analysis in that the writer is pushing to create as many copies as possible. In practice, the number of update versions is limited by physical memory; however, we ignore this constraint here. Instead, we employ a model that limits the creation of copies by constraining how fast the writer can write an update to memory. In this regard, we will sometimes call such a writing process as *unconstrained write process*. Specifically, we examine a system in which write operations to unlocked memory have independent exponential (α) service times. Since the writer receives a fresh update from the source immediately after publishing an update, there is a rate α Poisson point process of new updates being generated and written to memory.

Fig. 8.2.2 depicts the age process in shared memory as a function of time t. We assume an update 0 with initial age $\Delta(0)$ is in memory at time $t = 0$. Following the publication of update $n-1$ at time t_{n-1}, the writer queries the source for a fresh measurement. In response, the source generates an update n with time-stamp t_{n-1}. The writer receives this update instantly, begins writing to the shared memory, and subsequently publishes the new update at time t_n. The age $\Delta(t)$ at the shared memory increases linearly in time in the absence of any new update and is reset to a smaller value when an update is published. Thus, at time t_n, $\Delta(t)$ is reset to $W_n = t_n - t_{n-1}$. This continues for all subsequent updates and therefore, the age process $\Delta(t)$ exhibits a sawtooth waveform shown in Fig. 8.2.2.

We further assume that the read requests form a rate λ Poisson process, and each request's service/read time is an independent exponential (μ) random variable. Furthermore, we assume that memory is reclaimed when the last reader, holding the reference to a particular update, completes its service time. We refer to this as the *memoryless RCU* model since both write initiations and read requests are memoryless Poisson point processes.

Since read requests arrive as a Poisson process and the reads have exponential holding/service

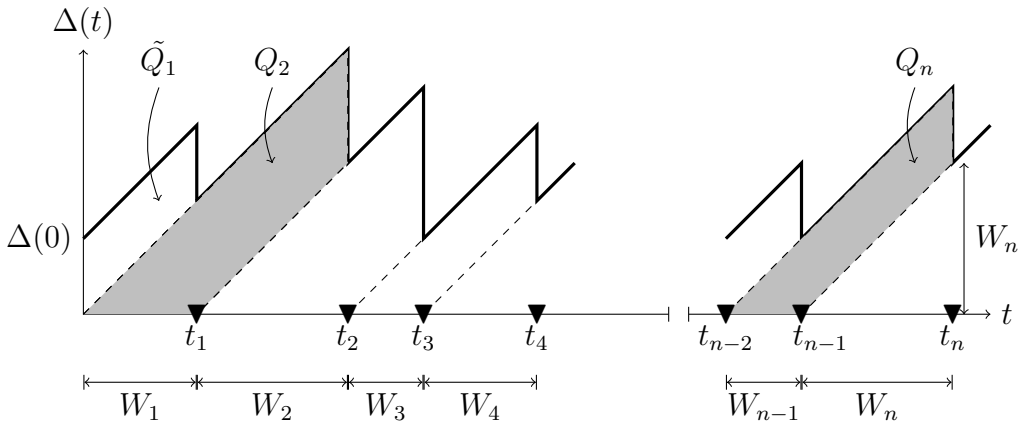


Figure 8.2.2: Example evolution of age at shared memory in the unconstrained write model. Updates are published in memory at times marked ▼.

times independent of the number of concurrent read requests of an update, the birth-death process of read locks is an M/M/∞ queue. However, from the perspective of the birth-death process of update copies in memory, the RCU system is complicated because update n is tagged by a number of read requests that depends on the write time of update $n+1$. This implies that the service/active time of an update depends upon the inter-arrival time of the next update; this is not an M/M/∞ queue.

8.2.1 Main Result

Let $N(t), t \geq 0$ denote the stochastic process of the number of active updates at time t. When each update has a fixed size in memory, $N(t)$ is proportional to the memory footprint of the RCU updating process. Theorem 7 desribes the memory footprint $\mathrm{E}[N]$ and the average age $\mathrm{E}[\Delta]$ of an update in memory in terms of the system parameters λ, μ and α.

Theorem 7. *For the memoryless RCU model in which updates are written as a rate α Poisson process, and read requests arrive as a rate λ Poisson process with independent exponential (μ) service times:*

(a) The memory footprint $\mathrm{E}[N]$ satisfies

$$\mathrm{E}[N] = 1 + \sum_{k=1}^{\infty} \sum_{j=0}^{\infty} \frac{b_k^j e^{-b_k}}{j!} \left(\frac{j}{\alpha/\mu + j} \right). \tag{8.1}$$

where $b_k = \lambda q^k/\mu$ *with* $q = \alpha/(\alpha+\mu)$.

(b)

$$\mathrm{E}[N] \leq 1 + \sum_{k=1}^{\infty} \frac{q^k}{q^k + \alpha/\lambda} \tag{8.2}$$

$$\leq 1 + \lambda/\mu. \tag{8.3}$$

(c) The average age of the current update in memory is

$$\mathrm{E}[\Delta] = 2/\alpha. \tag{8.4}$$

8.3 Proof of Theorem 7

8.3.1 Proof of Theorem 7(a)

Consider an example of unconstrained write process as shown in Fig. 8.3.1 along with the corresponding age evolution. We inspect the system at an arbitrary time t. Relative to time t, we look backward in time and define update 0 to be the most recently published update. We refer to update 0 as the current update. We also set our clock such that update 0 is published at time $S_0 = 0$. Further, we use index $k \geq 0$ to denote the update published k writes prior to update 0. We denote publication time of update k by S_k and thus the S_k are indexed backward in time, i.e., $\ldots, S_{n+1} < S_n < S_{n-1} < \ldots, S_1 < S_0 = 0$. Following this notation, the writing time of update n is $W_n = S_{n-1} - S_n$. Recall that W_n are i.i.d. exponential (α) random variables. By the memoryless property of the exponential random variable, $Z = t - S_0$, the time elapsed since the last published update, is also $\exp(\alpha)$.

When the writer publishes update $k-1$ at time S_{k-1}, the grace period for update k starts, and the writer starts writing update $k-2$. At this time, S_{k-1}, there is a random number of residual reader locks on update k and the grace period for update k terminates when all these residual readers release their respective locks.

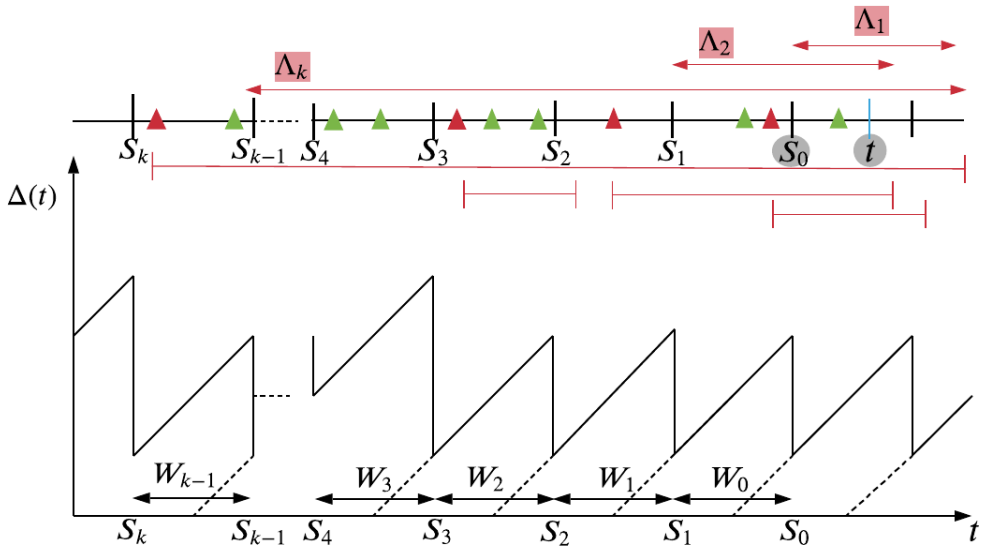


Figure 8.3.1: An example of the RCU read/write process (upper timeline) and the sample age evolution (shown only for illustration purpose) of update in memory (lower timeline). In the upper timeline: green triangles mark arrivals of read requests that finish before the next update is published; red triangles mark those reads that establish a grace period by holding a read lock after the next update is published; the red intervals beneath the upper timeline show the service times of such readers; the red arrows above the upper timeline (with labels Λ_k, Λ_2 and Λ_1) identify the grace periods of updates k, 2, and 1 that are active at time t.

For each past update $k > 0$, there is some probability that it remains in a grace period at time t. Given $W_{k-1} = w$, update k is the current update in the interval $(S_k, S_{k-1}) = (S_{k-1} - w, S_{k-1})$. In this length w interval, the number of read requests M_k is Poisson with $\mathrm{E}[M_k] = \lambda w$. Moreover, given $M_k = m$, the arrival times of the read requests are statistically identical to the set $\{S_{k-1} + U_1, \ldots, S_{k-1} + U_m\}$, where $U_1, \ldots, U_m$ is a set of i.i.d. uniform $(-w, 0)$ random variables [142]. These read requests will have i.i.d. exponential (μ) service times $X_1, X_2, \ldots, X_m$. The ith such read request releases its read-lock at time $S_{k-1} + Y_i$ where $Y_i = U_i + X_i$. Therefore, given $M_k = m$ and $W_{k-1} = w$ the last read-lock on update k is released at time

$$\Lambda_k = S_{k-1} + \max_{1 \le i \le m} Y_i. \tag{8.5}$$

We define L_{k-1} as the time elapsed since S_{k-1} up to time t. Then,

$$L_{k-1} = \sum_{j=0}^{k-2} W_j + Z. \tag{8.6}$$

The number of active updates at time t, N, is equal to the number of updates still in their respective grace periods at time t plus the current published update 0. To find $\mathrm{E}[N]$, we define E_k as the event that the grace period of update k has ended by time t. The conditional probability of event E_k that update k has finished its service by time t is

$$\begin{aligned}
\mathrm{P}[E_k \mid W_{k-1} = w, M_k = m] &= \mathrm{P}[\Lambda_k \leq S_{k-1} + L_{k-1} \mid W_{k-1} = w, M_k = m] \\
&= \mathrm{P}\left[\max_{1 \leq i \leq m} Y_i \leq L_{k-1} \mid W_{k-1} = w\right] \\
&= (\mathrm{P}[Y_i \leq L_{k-1} \mid W_{k-1} = w])^m,
\end{aligned} \tag{8.7}$$

where we have used the fact that the $Y_i = U_i + X_i$ remain i.i.d. under the condition $W_{k-1} = w$. We observe that L_{k-1} and X_i are independent of W_{k-1}. For fixed w, we also observe that U_i depends on W_{k-1} only to the extent that the event $W_{k-1} = w$ specifies that U_i is a uniform $(-w, 0)$ random variable. Hence, defining X to be exponential (μ) and U to be uniform $(-w, 0)$,

$$\mathrm{P}[E_k \mid W_{k-1} = w, M_k = m] = (\mathrm{P}[U + X \leq L_{k-1}])^m. \tag{8.8}$$

Lemma 9.

$$\mathrm{P}[U + X \leq L_{k-1}] = 1 - a_w q^k = \epsilon(k, w) \tag{8.9}$$

where $q = \alpha/(\alpha + \mu)$ *and* $a_w = (1 - e^{-\mu w})/\mu w$.

The proof, an elementary probability exercise, appears in the Appendix 8.A. It then follows from (8.8) and (8.9) that

$$\mathrm{P}[E_k \mid W_{k-1} = w] = \sum_{m=0}^{\infty} \mathrm{P}[E_k \mid W_{k-1} = w, M_k = m] P_{M_k|W_{k-1}}(m \mid w),$$

$$= \sum_{m=0}^{\infty} \frac{1}{m!} \epsilon(k,w)^m (\lambda w)^m e^{-\lambda w} = \exp\left(-b_k(1-e^{-\mu w})\right), \tag{8.10}$$

where $b_k = \lambda q^k/\mu$. Since, W_{k-1} is exponential (α), it follows from (8.10) that

$$\mathrm{P}[E_k] = \int_0^{\infty} P(E_k \mid W_{k-1} = w) f_{W_{k-1}}(w)\, dw = \alpha \int_0^{\infty} e^{-b_k(1-e^{-\mu w})} e^{-\alpha w}\, dw. \tag{8.11}$$

With the substitution $y = e^{-\mu w}$, we obtain

$$\mathrm{P}[E_k] = \frac{\alpha e^{-b_k}}{\mu} \int_0^1 y^{\alpha/\mu - 1} e^{b_k y}\, dy. \tag{8.12}$$

A Taylor series expansion of $e^{b_k y}$ yields

$$\begin{aligned} \mathrm{P}[E_k] &= \frac{\alpha e^{-b_k}}{\mu} \int_0^1 y^{\alpha/\mu - 1} \sum_{j=0}^{\infty} \frac{(b_k y)^j}{j!}\, dy \\ &= \frac{\alpha e^{-b_k}}{\mu} \sum_{j=0}^{\infty} \frac{b_k^j}{j!} \int_0^1 y^{\alpha/\mu + j - 1}\, dy, \\ &= \frac{\alpha e^{-b_k}}{\mu} \sum_{j=0}^{\infty} \frac{b_k^j}{j!(\alpha/\mu + j)}. \end{aligned} \tag{8.13}$$

It follows that

$$\mathrm{P}[E_k^c] = 1 - \mathrm{P}[E_k] = \sum_{j=0}^{\infty} \frac{b_k^j e^{-b_k}}{j!} \left(\frac{j}{\alpha/\mu + j}\right). \tag{8.14}$$

Now let I_k be the indicator random variable for the event E_k^c that update k is active at time t. Therefore, the number of active updates is $N = 1 + \sum_{k=1}^{\infty} I_k$. Hence,

$$\mathrm{E}[N] = 1 + \mathrm{E}[\sum_{k=1}^{\infty} I_k] = 1 + \sum_{k=1}^{\infty} \mathbb{P}(E_k^c). \tag{8.15}$$

Theorem 7(a) follows from (8.14) and (8.15).

8.3.2 Proof of Theorem 7(b)

To verify Theorem 7(b), let J_k denote a Poisson (b_k) random variable. We observe that (8.1) can be written as

$$\mathrm{E}[N] = 1 + \sum_{k=1}^{\infty} \mathrm{E}[\frac{J_k}{\alpha/\mu + J_k}]. \tag{8.16}$$

Since $x/(\alpha/\mu + x)$ is a concave function, using Jensen's inequality and the fact that $\mathrm{E}[J_k] = b_k = \lambda q^k/\mu$, we obtain

$$\mathrm{E}[N] \leq 1 + \sum_{k=1}^{\infty} \frac{\mathrm{E}[J_k]}{\alpha/\mu + \mathrm{E}[J_k]} = 1 + \sum_{k=1}^{\infty} \frac{q^k}{\alpha/\lambda + q^k}. \tag{8.17}$$

Since $q^k \geq 0$, it follows from (8.17) that

$$\mathrm{E}[N] \leq 1 + \sum_{k=1}^{\infty} \frac{\lambda}{\alpha} q^k = 1 + \frac{\lambda q}{\alpha(1-q)} = 1 + \frac{\lambda}{\mu}. \tag{8.18}$$

8.3.3 Proof of Theorem 7(c)

Fig. 8.2.2 represents a sample age evolution in the unconstrained write model with W_n denoting the exponential (α) write time of the nth update. We represent the area under sawtooth waveform as the concatenation of the polygon areas $\tilde{Q_1}, Q_2, \ldots, Q_n, \ldots$. The average age is $\Delta = \mathrm{E}[Q_n]/\,\mathrm{E}[W_n]$ where

$$Q_n = \frac{(W_{n-1} + W_n)^2}{2} - \frac{W_n^2}{2} = \frac{W_{n-1}^2 + W_{n-1}W_n}{2}. \tag{8.19}$$

Since $\mathrm{E}[W_n] = 1/\alpha$ and $\mathrm{E}[W_n^2] = 2/\alpha^2$,

$$\mathrm{E}[Q_n] = \mathrm{E}[W_{n-1}^2]/2 + \mathrm{E}[W_{n-1}]\,\mathrm{E}[W_n] = 2/\alpha^2 \tag{8.20}$$

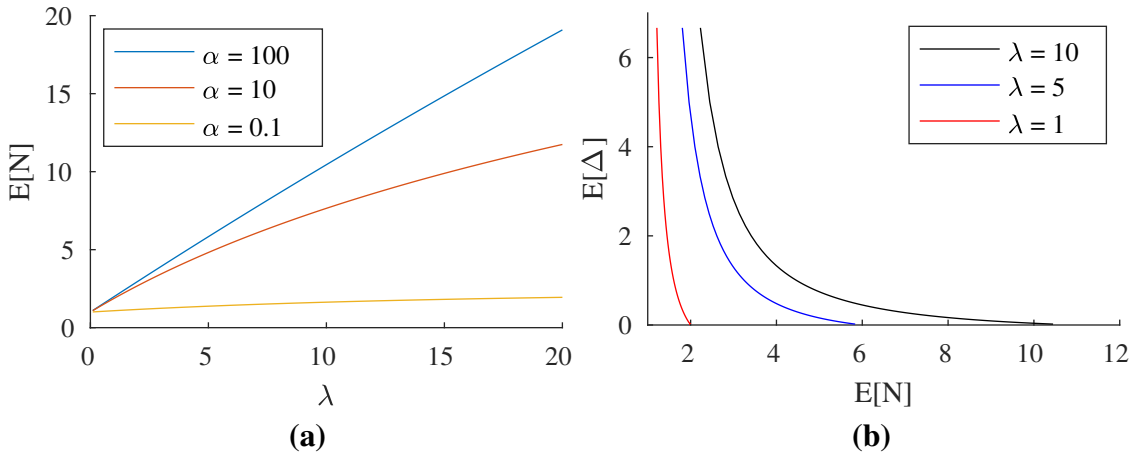


Figure 8.4.1: (a) Memory footprint in RCU as a function of read arrival rate λ. (b) Trade-off between the average age Δ and $\mathrm{E}[N]$ as a function of writing rate α. In both (a) and (b), the read service rate is $\mu = 1$.

and the claim follows.

8.4 Numerical Evaluation and Discussion

From Theorem 7, we see that the average age $\mathrm{E}[\Delta]$ of the current update in the memory is monotonically decreasing with the writing rate α. Fig. 8.4.1(b) plots age-memory trade-off over $\alpha \in (0, \infty)$, showing that minimal average age at the readers is achieved when updates are written as fast as possible, but this is at the expense of an increased memory footprint. However, for a fixed write rate α, the memory footprint is an increasing function of the read request rate λ as shown in Fig. 8.4.1(a). Both the analysis and numerical evaluation highlight the trade-off between age and memory observed in the RCU mechanism.

Fig. 8.4.2 plots $\mathrm{E}[N]$, and the upper bounds (8.2) and (8.3) as a function of α for $\mu = 1$ and various λ. We observe that the upper bound (8.2) is tight for all α. Further, notice that as $\alpha \to \infty$, the expected number of updates for different values of λ approach the upper bound in (8.3), albeit at different rates.

We now give some intuition for the upper bound to $\mathrm{E}[N]$ in Theorem 7(b). For $\alpha \gg \lambda$, each update is tagged with zero or one reads. As $\alpha \to \infty$, an untagged update expires in expected time $1/\alpha \to 0$, as it is replaced by the next update. On the other hand, a tagged update enters

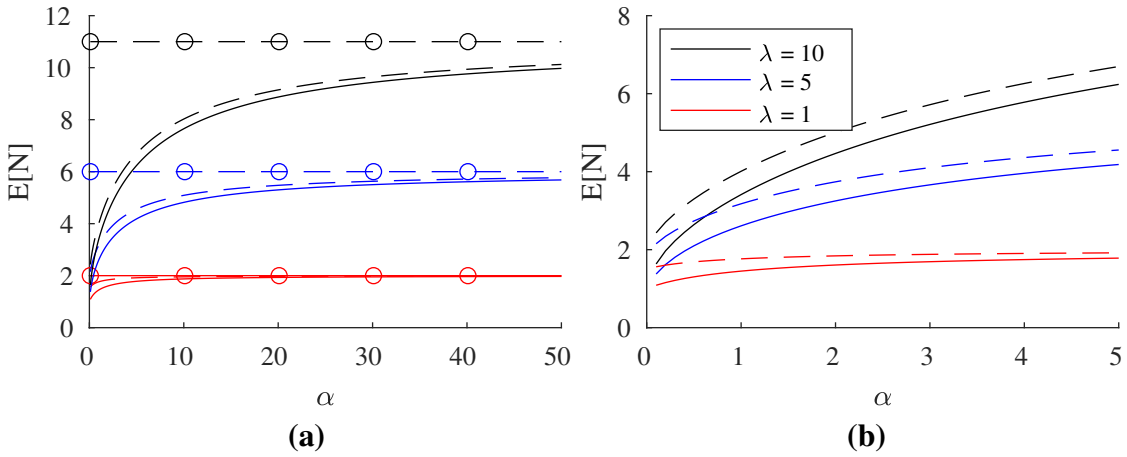


Figure 8.4.2: (a) The expected number of active updates are written at rate α. The black, blue and red curves are when $\lambda/\mu = 10$, $\lambda/\mu = 5$, and $\lambda/\mu = 1$ respectively; the read service rate is $\mu = 1$. (b) Zoomed in version of (a).

a grace period with duration corresponding to the exponential (μ) service time required by its read. Hence tagged updates have a one-to-one correspondence with the reads in the system. The number of tagged updates is described by the M/M/∞ queue process with arrival rate λ and service rate μ that characterizes the number of reads in the system. Therefore, in the limiting case of $\alpha \to \infty$, the number N' of tagged updates in the system follows a Poisson distribution $\mathrm{P}[N' = n] = (\lambda/\mu)^n e^{-\lambda/\mu}/n!$, for $n \geq 0$. Furthermore, there is always one untagged update that is perpetually being replaced. Hence, the number of updates in the system is $N = 1 + N'$ and the average number of updates holding a read lock is $\mathrm{E}[N] = \mathrm{E}[1 + N'] = 1 + \lambda/\mu$.

We conclude this discussion by relating memory size and the read rate μ, a relationship contingent upon the interpretation of μ within the context of update operations. One interpretation links μ to the time needed for copy operations. Under this view, if the update size is represented by M, then M will be proportional to the average read time $1/\mu$. Consequently, according to (8.3), the average number of update copies will scale linearly with M, resulting in memory usage proportional to M^2. In an alternate interpretation, we regard $1/\mu$ as the average duration of a read lock's holding time. In this scenario, readers undertake computational tasks where the read lock necessitates access to the entire object for calculations. However, the actual read time might be negligible but the reader has to navigate through different sections of the data structure. Here, the object size has no direct

implication on μ, instead μ varies based on the complexity of the computations performed. As such, the memory consumption in this case grows only linearly with object size M.

8.5 Conclusion

In this work, we explored the trade-off between memory footprint and update age in the context of RCU, particularly relevant for applications with sizable updates operating within the constraints of memory-constrained mobile devices. The central question is whether frequent updating can induce excessive memory consumption. Theorem 7 provides a reassuring finding — given finite average service/read time $1/\mu$ and read request rate λ, the average number of updates in the system is finite.

APPENDIX

8.A Proof of Lemma 9

Lemma 9

$$\mathrm{P}[U + X \leq L_{k-1}] = 1 - a_w q^k = \epsilon(k, w) \tag{8.21}$$

where $q = \alpha/(\alpha + \mu)$ and $a_w = (1 - e^{-\mu w})/\mu w$.

Proof. Since the W_j and Z are i.i.d. exponential (α) random variables, (8.6) implies L_{k-1} has a Gamma distribution with PDF

$$f_{L_{k-1}}(l) = \frac{\alpha}{\Gamma(k)}(\alpha l)^{k-1} e^{-\alpha l} \mathbf{1}_{\{l \geq 0\}}. \tag{8.22}$$

Since $Y = U + X$, where $U \sim$ Uniform$(-w, 0)$ and $X \sim \exp(\mu)$, the PDF of Y is

$$\begin{aligned} f_Y(y) &= \int_{-\infty}^{\infty} f_X(x) f_U(y - x)\, dx \\ &= \int_{-\infty}^{\infty} \mu e^{-\mu x} \mathbf{1}(x \geq 0) \frac{1}{w} \mathbf{1}(-w \leq y - x \leq 0)\, dx. \end{aligned} \tag{8.23}$$

Resolving the indicator functions in (8.23) yields

$$f_Y(y) = \begin{cases} \frac{1}{w}(1 - e^{-\mu(w+y)}), & -w \leq y \leq 0, \\ \frac{1}{w}(e^{-\mu y} - e^{-\mu(w+y)}), & y \geq 0. \end{cases} \tag{8.24}$$

This implies

$$\mathrm{P}[Y \leq L_k] = \int_{l=0}^{\infty} f_{L_k}(l) \int_{-w}^{l} f_Y(y)\, dy\, dl = I_1 + I_2 \tag{8.25}$$

such that

$$
\begin{aligned}
I_1 &= \int_0^\infty f_{L_k}(l) \int_{-w}^{0} f_Y(y)\, dy\, dl \\
&= \int_0^\infty f_{L_k}(l) \left[\int_{-w}^{0} \frac{1}{w}\, dy - \frac{1}{w} \int_{-w}^{0} (1 - e^{-\mu(w+y)})\, dy \right] dl \\
&= 1 + (e^{-\mu w} - 1)/(\mu w), \qquad (8.26) \\
I_2 &= \int_0^\infty f_{L_k}(l) \int_0^{l} f_Y(y)\, dy\, dl, \\
&= \underbrace{-\frac{1}{\mu w} \int_{l=0}^{\infty} f_{L_k}(l)(e^{-\mu l} - 1)\, dl}_{I_3} + \underbrace{\frac{1}{\mu w} \int_{l=0}^{\infty} f_{L_k}(l)(e^{-\mu(w+l)} - e^{-\mu w})\, dl.}_{I_4} \qquad (8.27)
\end{aligned}
$$

Solving integrals I_3 and I_4:

$$
\begin{aligned}
I_3 &= \frac{1}{\mu w} \left[\int_{l=0}^{\infty} f_{L_k}(l) e^{-\mu l} dl - \int_{l=0}^{\infty} f_{L_k}(l) dl \right] \\
&= \frac{1}{\mu w} \left[\int_{l=0}^{\infty} \frac{\alpha}{\Gamma(k)} (\alpha y)^{k-1} e^{-(\alpha+\mu)l} dl - 1 \right] \\
&= \frac{1}{\mu w} \left[(\frac{\alpha}{\alpha+\mu})^k - 1 \right], \qquad (8.28) \\
I_4 &= \frac{1}{\mu w} \left[\int_{l=0}^{\infty} f_{L_k}(l) e^{-\mu(w+l)} dl - \int_{l=0}^{\infty} f_{L_k}(l) e^{-\mu w} dl \right] \\
&= \frac{1}{\mu w} \left[e^{-\mu w} \int_{l=0}^{\infty} \frac{\alpha}{\Gamma(k)} (\alpha y)^{k-1} e^{-(\alpha+\mu)l} dl - e^{-\mu w} \right] \\
&= \frac{e^{-\mu w}}{\mu w} \left[(\frac{\alpha}{\alpha+\mu})^k - 1 \right]. \qquad (8.29)
\end{aligned}
$$

Combining (8.26), (8.27), (8.28), and (8.29), we have,

$$
\mathrm{P}[Y \le L_k \mid W_{k-1} = w] = 1 - \frac{1 - e^{-\mu w}}{\mu w} \left(\frac{\alpha}{\alpha+\mu} \right)^k. \qquad (8.30)
$$

Recalling $q = \alpha/(\alpha+\mu)$ and $a_w = (1 - e^{-\mu w})/(\mu w)$, the lemma follows. □

Part IV

Timely and Energy-Efficient Multi-Step Update Processing

CHAPTER 9
TIMELY AND ENERGY-EFFICIENT MULTI-STEP UPDATE PROCESSING

9.1 Introduction

This chapter explores systems where source updates require multiple sequential processing stages. We model and analyze various system designs under both parallel and series server configurations. In parallel setups, multiple processors execute all computation steps independently, while in series (pipeline) configurations, each processor performs a specific step in sequence. We also address the occurrence of wasted power, which arises when processing efforts do not lead to a reduction in age. This happens when a fresher update finishes first in parallel servers or when a server preempts processing due to a fresher update in pipeline setups. We formulate and solve optimization problems for a special case where updates require two computational steps, and determine the optimal service rates for each step when the system is subject to a power constraint.

9.1.1 Contributions and Chapter Outline

Section 9.2 introduces the system parameters and the power consumption model. In Section 9.3, we examine the series server setup, where we first formulate the optimization problem for a general tandem queue with two servers model. We then explore various preemptive and non-preemptive tandem queue models, establishing the relationship between the parameters of the general optimization problem and specific tandem queue configurations.

Similar to Section 9.3, in Section 9.4, we formulate a general optimization problem for parallel server setups, focusing on optimizing update processing step rates when two servers operate in parallel. We begin by analyzing a baseline model where the two servers function independently and introduce a novel approach using the Stochastic Hybrid Systems (SHS) methodology to derive a system of linear equations for calculating the average age at the monitor.

Additionally, we propose three heuristic policies designed to improve age performance in parallel server setups: Parallel Coordinated Alternating Freshness (P-CAF), Synchronized Freshness (SF), and Parallel Shared Intermediate Update (P-SIU).

These policies leverage information about each server's current stage of processing to optimize update handling. For each of these heuristic policies, we optimize the computational step rates to achieve minimal age under the given power constraint.

In Section 9.5, we conduct a comparative analysis of age performance across the various series and parallel server models introduced in Sections 9.3 and 9.4. Finally, in Section 9.6 we conclude by discussing several open problems that emerge from this work and propose directions for future research.

9.2 System Model Overview

We assume that processing source updates involves a sequence of two computational steps. Each step i, with $i \in \{1, 2\}$, involves a random computational workload C_i, measured in CPU cycles. The total CPU demand for a source update is thus $C_1 + C_2$ cycles. Each computational step i is executed at a constant processing frequency f_i (CPU cycles per unit time), resulting in an execution time $T_i = C_i/f_i$. The average service rate for step i is then given by $\mu_i = f_i/\operatorname{E}[C_i]$. We assume that the execution time for step i is an independent exponential random variable with rate μ_i i.e., $T_i \sim \exp(\mu_i)$. The update processing occurs on a multi-processor machine, which we model as a queuing system with multiple servers.

9.2.1 Processor Speed and Power Consumption Model

Power dissipation in digital CMOS circuits is primarily attributed to dynamic power, short circuit losses, and transistor leakage currents [143]. Among these, dynamic power consumption is currently the main component in high-performance microprocessors. Dynamic power, driven by the periodic

switching of capacitors, can be approximated by the well-known formula

$$P = AC_L V^2 f, \tag{9.1}$$

where A and C_L denote the Activity Factor (AF) and loading capacitance, respectively, V is the supply voltage, and f is the clock frequency [144]. According to alpha-power law MOS model [145], $f \propto V^{\alpha_c - 1}$, where α_c, also called as velocity saturation index, is a technology dependent factor, typically ranging between 1 and 2. This implies $V \propto f^{1-\alpha_c}$, and consequently, the power consumption is $P \propto f^{\alpha}$ where $\alpha = (1+\alpha_c)/(\alpha_c - 1) \geq 3$. According to [146], for a 25 μm technology, α_c is likely to be in range $[1.3, 1.5]$. For our numerical evaluations, we fix the velocity saturation index at $\alpha_c = 1.5$, which corresponds to $\alpha = 5$.

9.3 Problem Formulation: Sequential Servers

We consider two servers in series, where each server handles one computation step in the update processing sequence as illustrated by Fig. 1.2 with $n = 2$. The output of Server 1 is forwarded to Server 2, which delivers the processed update to the monitor instantaneously upon completing step 2. Each Server i operates at a constant frequency f_i.

The total power consumption at the two servers is limited by a power budget. Let P represent the total power budget. Then the sum of power consumption at Server 1 and 2 should be smaller or equal to constant P. We assume that an idle processor consumes negligible power. Let p_i represent the probability Server i is busy, then we have the following constraint:

$$p_1 f_1^{\alpha} + p_2 f_2^{\alpha} \leq P. \tag{9.2}$$

Given that $\mu_i = f_i / \operatorname{E}[C_i]$, and assuming $\operatorname{E}[C_1] = \operatorname{E}[C_2] = \operatorname{E}[C]$, we can express processing frequency f_i as $\mu_i \operatorname{E}[C]$. Substituting this into the power constraint in (9.2), we obtain:

$$p_1 \mu_1^{\alpha} + p_2 \mu_2^{\alpha} \leq \frac{P}{\operatorname{E}[C]^{\alpha}}. \tag{9.3}$$

In this tandem queue setup, the busy probabilities p_1 and p_2 depend on the service rates and queuing discipline at both servers. To elucidate the dependence of p_1 and p_2 on μ_1 and μ_2, we re-write the power constraint (9.3) as:

$$p_1(\mu_1, \mu_2)\mu_1^\alpha + p_2(\mu_1, \mu_2)\mu_2^\alpha \leq \frac{P}{\mathrm{E}[C]^\alpha}. \tag{9.4}$$

The age at the monitor, denoted by $\Delta_{\text{queue}}(\mu_1, \mu_2)$, is a function of the service rates μ_1 and μ_2, and is influenced by the queuing discipline at each server. Our objective is to minimize the age $\Delta_{\text{queue}}(\mu_1, \mu_2)$ at the monitor by controlling the service rates μ_1 and μ_2, subject to the power constraint (9.4). The optimization problem is thus formulated as:

$$\begin{aligned}
&\text{minimize} && \Delta_{\text{queue}}(\mu_1, \mu_2) && \text{(9.5a)}\\
&\text{subject to} && p_1(\mu_1, \mu_2)\mu_1^\alpha + p_2(\mu_1, \mu_2)\mu_2^\alpha \leq \frac{P}{\mathrm{E}[C]^\alpha}, && \text{(9.5b)}\\
& && \mu_1, \mu_2 \geq 0. && \text{(9.5c)}
\end{aligned}$$

To solve the multi-variable optimization problem (9.5), our strategy is to exploit the relationship between μ_1 and μ_2. Define $\rho = \mu_1/\mu_2$, where $\rho \geq 0$, representing the total offered load from Server 1 to Server 2. With this definition, the busy probabilities p_1 and p_2 can be expressed as functions of ρ i.e., $p_1(\mu_1, \mu_2) \equiv p_1(\rho)$ and $p_2(\mu_1, \mu_2) \equiv p_2(\rho)$. Next, substituting $\mu_1 = \rho\mu_2$ into the constraint in (9.4) yields:

$$\mu_2^\alpha \Big(\rho^\alpha p_1(\rho) + p_2(\rho)\Big) \leq \frac{P}{\mathrm{E}[C]^\alpha}, \tag{9.6}$$

which provides the upper bound on the service rate of Server 2 as:

$$\mu_2 \leq \frac{1}{\mathrm{E}[C]} \left(\frac{P}{\rho^\alpha p_1(\rho) + p_2(\rho)}\right)^{1/\alpha}. \tag{9.7}$$

Since ρ determines μ_1 relative to μ_2, the age at the monitor can be expressed as a function of μ_2 and

ρ, $\Delta_{\text{queue}}(\mu_2, \rho)$. Hence, the optimization problem (9.5) can be reformulated as

$$\text{minimize} \quad \Delta_{\text{queue}}(\mu_2, \rho) \tag{9.8a}$$

$$\text{subject to} \quad \mu_2 \leq \frac{1}{\mathrm{E}[C]} \left(\frac{P}{\rho^\alpha p_1(\rho) + p_2(\rho)} \right)^{1/\alpha}, \tag{9.8b}$$

$$\mu_2 \geq 0, \text{and } \rho \geq 0. \tag{9.8c}$$

Observe that the technology fixes α, and the power constraint P and CPU demand C are system parameters. With ρ fixed, $\rho^\alpha p_1(\rho)$ is the relative fraction of the energy budget utilized at Server 1 and $p_2(\rho)$ is the relative fraction of energy budget at Server 2. For this fixed ρ, the service rates μ_2 and $\mu_1 = \rho\mu_2$ yields an average age at the monitor. We will see that this age is typically minimized by choosing μ_2 as large as possible subject to the upper bound (9.8b). What remains is choosing the right value of ρ. A larger ρ keeps Server 2 busier with fresh arrivals by using more energy at Server 1 but this may be wasting the effort of Server 1. On the other hand, if ρ is smaller, then the system may not be not feeding enough updates to Server 2. For a given system design choice, finding an optimal ρ^* allows us to determine the corresponding optimal service rate μ_2^* using the right side of (9.8b), which in turn yields the optimal age $\Delta_{\text{queue}}(\mu_2^*, \rho^*)$.

In the following , we describe variations on non-preemptive and preemptive queues on each server and provide analytical expressions for $p_1(\rho)$, $p_2(\rho)$, and $\Delta_{\text{queue}}(\mu_2, \rho)$. We assume a generate-at-will with zero-wait scenario at Server 1 such that it can generate a fresh (age zero) update whenever it wishes. However, we consider variations on service disciplines at Server 2. There may be a single queue to save updates from Server 1 when Server 2 is busy. Since, the queuing (if any) is only at Server 2, we name our sub-models based on the queuing discipline at Server 2.

Since Server 1 employs a generate-at-will with zero-wait strategy and has memoryless service times, its departure process is a Poisson process with rate μ_1. Consequently, the inter-arrival times of updates at Server 2 follow an exponential distribution with parameter μ_1. The service time at Server 2 is also exponential, with rate μ_2.

We adopt Kendall's notation to denote the queuing discipline at Server 2, following the con-

vention used in the AoI literature [134, 4]. For example, an M/M/1/1 submodel implies a queueing system that blocks and clears a new arrival while Server 2 is busy. We use the notation M/M/1* to indicate preemption in service at Server 2, and M/M/1/2* to denote a system with a waiting room having an update capacity of 1, with preemption in waiting. We now describe the analysis of these models in detail.

9.3.1 M/M/1*

In this model, Server 1 generates a fresh update immediately upon completing the processing of the previous update. The update is then passed to Server 2 at a rate μ_1. Server 2 employs preemption in service, allowing a new arrival from Server 1 to preempt an update currently being serviced at Server 2. Consequently, an update departing from Server 1 immediately enters service at Server 2, and any preempted update at Server 2 is discarded. Since there is no queuing at Server 2, it is either idle or actively serving an update. Since, we've assumed memoryless service times, the fraction of time Server 2 is busy is

$$p_2(\rho) = \frac{\mu_1}{\mu_1 + \mu_2} = \frac{\rho}{1+\rho}. \tag{9.9}$$

Notably, Server 1 remains perpetually busy, i.e.

$$p_1(\rho) = 1. \tag{9.10}$$

This setup is analogous to the line network studied in [27, 32], where it was demonstrated that the age at the monitor for a two-server line network, applicable to our model as well, is given by:

$$\Delta_{\text{M/M/1}^*}(\mu_1, \mu_2) = \frac{1}{\mu_1} + \frac{1}{\mu_2}, \tag{9.11}$$

Alternatively, we can express the age in terms of μ_2 and ρ as

$$\Delta_{\text{M/M/1}^*}(\mu_2, \rho) = \frac{1}{\mu_2}\left(1 + \frac{1}{\rho}\right). \tag{9.12}$$

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$
1	$0 \to 1$	μ_1	$[x_0, 0, x_1, x_1]$
2	$1 \to 0$	μ_2	$[x_2, x_1, x_2, x_3]$
3	$1 \to 2$	μ_1	$[x_0, 0, x_2, x_1]$
4	$2 \to 2$	μ_1	$[x_0, 0, x_2, x_1]$
5	$2 \to 1$	μ_2	$[x_2, x_1, x_3, x_3]$

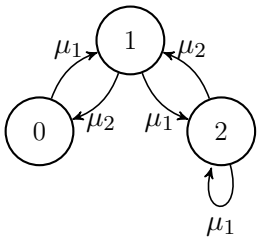


Figure 9.3.1: The SHS transition maps and Markov Chain corresponding to $M/M/1/2^*$ model.

9.3.2 M/M/1/2*

Server 1 generates a fresh update as soon as it finishes processing the previous update. This means that Server 1 is always busy, thus $p_1(\rho) = 1$. The step 1 update is then sent to the waiting room of Server 2, which has a capacity of 1. In this waiting room, a new arrival from Server 1 preempts any existing update. Server 2 sits idle if its waiting room is empty. The age of processed update can be analysed using the SHS Markov chain and table of state transitions depicted in Fig. 9.3.1. The continuous state age vector is $\mathbf{x} = [x_0, x_1, x_2, x_3]$, where x_0 is the age of the processed update at the monitor, x_1 and x_2 are the ages of the update at Server 1 and Server 2 respectively, and x_3 is the age of the update at Server 2's waiting room. The discrete state is $\mathcal{Q} = \{0, 1, 2\}$, where state 0 corresponds to Server 2 being idle, and states 1 and 2 correspond to Server 2 being busy with no update in the queue and one update waiting in the queue, respectively.

We now describe SHS transitions enumerated in the table in Fig. 9.3.1.

- $l = 1$: Server 1 finishes step 1, sends the update to idle Server 2. Server 2 receives an update of age x_1, thus $x_2' = x_1$. A fresh update is generated at Server 1, thus $x_1' = 0$. Age at the monitor remains unchanged, hence $x_0' = x_0$.

- $l = 2$: Server 2 finishes step 2, and delivers the update to monitor, making $x_0' = x_2$. The waiting room is empty, and Server 2 waits for an update from Server 1, resulting in no change in x_2.

- $l = 3, 4$: Update from Server 1 arrives in the waiting room and preempts the update (if any), resetting the age in the waiting room to $x_3' = x_1$. Server 1 generates a fresh update, hence

$x_1' = 0$.

- $l = 5$: Server 2 finishes step 2 and delivers update to the monitor, resulting in $x_0' = x_2$. Since there is an update waiting in Server 2's buffer with age x_3, Server 2 starts processing this update, thus age at Server 2 is reset to the age of update in the waiting room i.e., $x_2' = x_3$.

The Markov chain in Fig. 9.3.1 has stationary probabilities $\boldsymbol{\pi}$ with normalization constant C_π given by

$$\boldsymbol{\pi} = [\pi_0 \ \pi_1 \ \pi_2] = C_\pi^{-1}[1 \ \rho \ \rho^2], \tag{9.13a}$$

$$C_\pi = 1 + \rho + \rho^2. \tag{9.13b}$$

The probability that Server 2 is busy is thus,

$$p_2(\rho) = \pi_1 + \pi_2 = \frac{\rho(1+\rho)}{1+\rho+\rho^2}. \tag{9.14}$$

We now use Theorem 1 to solve for

$$\bar{\mathbf{v}} = [\bar{\mathbf{v}}_0 \ \bar{\mathbf{v}}_1 \ \bar{\mathbf{v}}_2], \tag{9.15}$$

where $\mathbf{v}_q = [v_{q0} \ v_{q1} \ v_{q2} \ v_{q3}], \forall q \in \mathcal{Q}$. This yields

$$\mu_1 \bar{\mathbf{v}}_0 = \mathbf{1}\bar{\pi}_0 + \mu_2 \bar{\mathbf{v}}_1 \mathbf{A}_2, \tag{9.16a}$$

$$(\mu_1 + \mu_2)\bar{\mathbf{v}}_1 = \mathbf{1}\bar{\pi}_1 + \mu_1 \bar{\mathbf{v}}_0 \mathbf{A}_1 + \mu_2 \bar{\mathbf{v}}_2 \mathbf{A}_5, \tag{9.16b}$$

$$(\mu_1 + \mu_2)\bar{\mathbf{v}}_2 = \mathbf{1}\bar{\pi}_2 + \mu_1 \bar{\mathbf{v}}_2 \mathbf{A}_4 + \mu_1 \bar{\mathbf{v}}_1 \mathbf{A}_3. \tag{9.16c}$$

The age at the monitor, $\Delta_{\text{M/M/1/2*}}$, is then calculated as $\Delta_{\text{M/M/1/2*}} = v_{0,0} + v_{1,0} + v_{2,0}$. Some algebra

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$
1	$0 \to 1$	μ_1	$[x_0, 0, x_1]$
2	$1 \to 1$	μ_1	$[x_0, 0, x_2]$
3	$1 \to 0$	μ_2	$[x_2, x_1, x_2]$

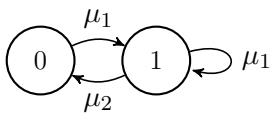


Figure 9.3.2: The SHS transition maps and Markov Chain corresponding to $M/M/1/1$ model.

yields:

$$\Delta_{\text{M/M/1/2*}}(\mu_1, \mu_2) = \frac{2}{\mu_1} + \frac{2\mu_1^2}{\mu_2(\mu_1^2 + \mu_1\mu_2 + \mu_2^2)} + \frac{(\mu_2 + 2\mu_1)(\mu_1^2 + 3\mu_1\mu_2 + \mu_2^2)}{(\mu_1 + \mu_2)^4}. \tag{9.17}$$

Furthermore, by substituting $\mu_1 = \rho\mu_2$ in (9.17), we obtain

$$\Delta_{\text{M/M/1/2*}}(\mu_2, \rho) = \frac{1}{\mu_2}\Big(\frac{2}{\rho} + \frac{2\rho^2}{1 + \rho + \rho^2} + \frac{(1 + 2\rho)(1 + 3\rho + \rho^2)}{(1 + \rho)^4}\Big). \tag{9.18}$$

We now make an observation on the age expression in (9.17) and age expression derived in [147, Theorem 1, Equation (9)]. The authors in [147] studied age performance in edge computing scenario, where a source generates a packet at will with zero wait. The source packet is sent to the edge server for computation. The transmission time is assumed exponential with rate λ. The edge server is modelled as a service facility with one packet waiting capacity with preemption in waiting. Computing time at edge server is assumed exponential(μ). Thus we notice that the end-to-end model in [147] is same as the M/M/1/2* model, with λ equivalent to μ_1 and μ equivalent to μ_2. Then the age expression in (9.17) can be shown to be identical to the independently derived result in [147], where the authors instead employed the sawtooth waveform analysis.

9.3.3 M/M/1/1

In this model as well, Server 1 generates a fresh source update as soon as it finishes processing the previous one. Server 1 then sends updates to Server 2 at rate μ_1. The service facility at Server 2 operates under a non-preemptive First-Come-First-Serve discipline with no waiting queue. If Server 2 is busy when a new update arrives, the new update is discarded. Consequently, Server 2

only accepts updates when it is idle. The probability that Server 2 is occupied is

$$p_2(\rho) = \frac{\mu_1}{\mu_1 + \mu_2} = \frac{\rho}{1+\rho}. \tag{9.19}$$

Since Server 1 is always busy,

$$p_1(\rho) = 1. \tag{9.20}$$

The age at the monitor for M/M/1/1 model $\Delta_{\text{M/M/1/1}}$ can be described by the SHS Markov chain and table of state transitions shown in Fig. 9.3.2. The continuous age state vector is $\mathbf{x} = [x_0, x_1, x_2]$, where x_0 is the age of the processed update at the monitor, x_1 and x_2 are ages of the update at Server 1 and Server 2 respectively. For this model, discrete states are $\mathcal{Q} = \{0, 1\}$, where 0 and 1 correspond to Server 2 being idle and busy respectively. The SHS transitions are self-explanatory. Employing Theorem 1, we calculate age at the monitor as $\Delta_{\text{M/M/1/1}} = v_{0,0} + v_{1,0}$. Some algebraic manipulation gives:

$$\Delta_{\text{M/M/1/1}}(\mu_1, \mu_2) = \frac{2}{\mu_1} + \frac{2}{\mu_2}, \tag{9.21}$$

or equivalently,

$$\Delta_{\text{M/M/1/1}}(\mu_2, \rho) = \frac{2}{\mu_2}\Big(1 + \frac{1}{\rho}\Big). \tag{9.22}$$

9.3.4 Synchronous Sequential Service (SSS)

In this model, servers work synchronously, meaning Server 1 generates a fresh update after Server 2 finishes step 2 on previous update. Consequently, processing on source update starts when both servers are idle. Here, only one server is busy at any given time. The age analysis for this model can be approached using either the sawtooth waveform method or the SHS method. For consistency with previous analyses, we apply the SHS method to evaluate the AoI at the monitor.

Fig. 9.3.3 illustrates SHS Markov Chain and table of state transitions for synchronous servers model. The continuous age state vector is $\mathbf{x} = [x_0, x_1, x_2]$, where x_0 is the age of the processed update at the monitor, and x_1 and x_2 are the ages of the update at Server 1 and Server 2 respectively.

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$
1	$1 \to 2$	μ_1	$[x_0, x_1, x_1]$
2	$2 \to 1$	μ_2	$[x_2, 0, x_2]$

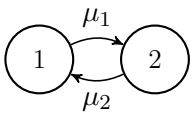


Figure 9.3.3: The SHS transition maps and Markov chain corresponding to Synchronous Sequential Servers (SSS) model.

For this model, discrete states are $\mathcal{Q} = \{1, 2\}$, where 1 and 2 correspond to Server 1 and Server 2 being busy respectively. We skip explaining the SHS transitions due to space constraints, however we do note that unlike in M/M/1/2* and M/M/1/1 models, $x_1' = 0$ occurs at the transition corresponding to μ_2. The Markov Chain in Fig. 9.3.3 has stationary probabilities

$$\pi_1 = \frac{\mu_2}{\mu_1 + \mu_2}, \quad \text{and} \quad \pi_2 = \frac{\mu_1}{\mu_1 + \mu_2}. \tag{9.23}$$

The probabilities that servers 1 and 2 are busy are then

$$p_1(\rho) = \frac{1}{1+\rho}, \quad \text{and} \quad p_2(\rho) = \frac{\rho}{1+\rho}. \tag{9.24}$$

The age at the monitor, $\Delta_{\text{sync}}(\mu_1, \mu_2)$, is calculated as $v_{10} + v_{20}$, resulting in

$$\Delta_{\text{SSS}}(\mu_1, \mu_2) = \frac{1}{\mu_1} + \frac{1}{\mu_2} + \frac{1}{\mu_1 + \mu_2}\left(1 + \frac{\mu_1}{\mu_2} + \frac{\mu_2}{\mu_1}\right). \tag{9.25}$$

Alternatively, the age (9.25) can be expressed in terms of μ_2 and ρ as:

$$\Delta_{\text{SSS}}(\mu_2, \rho) = \frac{1}{\mu_2}\Big(2 + \frac{1}{\rho} + \frac{1}{\rho(1+\rho)}\Big). \tag{9.26}$$

9.4 Problem Formulation: Parallel Servers

The power consumption in a system with two parallel servers is more complex than in a system with servers arranged in series. The parallel system's states can involve both servers in either step 1 or step 2, or the servers can be executing different steps. If we denote the state space of the parallel

system by $\mathcal{Q}$, then $\mathcal{Q} = \mathcal{U} \cup \mathcal{V}$, where $\mathcal{U}$ is the set of states where at least one server is in stage 1, and $\mathcal{V}$ is the set of states where at least one server is in stage 2.

Our aim here is to express the average power consumption for each stage by summing the power consumption across the relevant states in the Markov chain, weighted by the stationary probabilities of those states. Let $\pi_q, q \in \mathcal{Q}$ represent the stationary probability of the Markov Chain with discrete state space $\mathcal{Q}$. Let $r_u(\mu_1, \alpha)$ be the power consumption associated with stage 1 execution in state $u \in \mathcal{U}$. The average power consumption during stage 1 execution across all relevant states is then $\sum_{u\in\mathcal{U}} \bar{\pi}_u r_u(\mu_1, \alpha)$. Similarly, let $r_v(\mu_1, \alpha)$ denote the power consumption in stage 2 for any state $v \in \mathcal{V}$. The average power consumption during stage 2 execution is then $\sum_{v\in\mathcal{V}} \bar{\pi}_v r_v(\mu_2, \alpha)$.

Thus, the total power consumption in parallel servers should satisfy the following constraint:

$$\sum_{u\in\mathcal{U}} \bar{\pi}_u r_u(\mu_1, \alpha) + \sum_{v\in\mathcal{V}} \bar{\pi}_v r_v(\mu_2, \alpha) \leq P. \tag{9.27}$$

Our objective is to minimize age $\Delta_{\text{parallel}}(\mu_1, \mu_2)$ at the monitor by optimizing the service rates μ_1 and μ_2, subject to the power constraint (9.27). The optimization problem is thus formulated as:

$$\begin{aligned}
&\text{minimize} && \Delta_{\text{parallel}}(\mu_1, \mu_2) && \text{(9.28a)}\\
&\text{subject to} && \sum_{u\in\mathcal{U}} \bar{\pi}_u r_u(\mu_1, \alpha) + \sum_{v\in\mathcal{V}} \bar{\pi}_v r_v(\mu_2, \alpha) \leq P. && \text{(9.28b)}\\
& && \mu_1, \mu_2 \geq 0. && \text{(9.28c)}
\end{aligned}$$

In the following discussion, we derive explicit analytical expressions for $\Delta_{\text{parallel}}(\mu_1, \mu_2)$, identify the system state set $\mathcal{Q}$, the stationary probabilities π_q, and the state subsets $\mathcal{U}$ and $\mathcal{V}$. Additionally, we will specify the corresponding power consumption functions $r_u(\mu_1, \alpha)$ and $r_v(\mu_2, \alpha)$. To solve the optimization problem in (9.28), we employ an approach similar to that used in Section 9.3. Specifically, we define the ratio $\rho = \mu_1/\mu_2$ and derive an upper bound for the service rate μ_2.

It is important to note that in parallel server systems, the quantity $\rho = \mu_1/\mu_2$ does not always accurately represent the total offered load from step 1 to step 2. This is because one server may

transition from stage 1 to stage 2 while the other remains in stage 1. Moreover, as we will see in certain parallel server policies discussed later, even within a single server, the concept of offered load from steps 1 to step 2 can break down, diverging from its conventional interpretation in general queueing theory. Therefore, we refrain from referring to ρ as the offered load in parallel setup. Instead, we treat ρ purely as the ratio between the service rates of step 1 and step 2.

9.4.1 Parallel SSS (P-SSS)

In this mode, two identical servers process updates independently, in parallel. Each server works on a separate update and executes both computation steps. After processing an update, a server immediately starts processing a fresh update. The total service time for an update is a random variable

$$T = T_1 + T_2 = C_1/f_1 + C_2/f_2. \tag{9.29}$$

With $T_i \sim \exp(\mu_i)$, the service time T is a two-parameter hypoexponential distribution with parameters μ_1 and μ_2.

The age analysis for parallel server setup is non-trivial even with two parallel servers. The issue is that unlike series server setup, not every update delivery by servers will reset the monitor age i.e. not every update delivery is "useful". The issue arises due to the variability in processing times. A server might take longer to process an update, resulting in the second server, which is working on a fresher update, completing its task sooner and resetting the monitor's age. When the older update from the first server eventually arrives, it does not reset the monitor's age.

When Server $i, i \in \{1, 2\}$, sends an update with age x_i to the monitor, the monitor accepts a processed update only if it is fresher than its current update. Consequently, the resulting age at the monitor, denoted by x_0, is updated as follows:

$$x_0' = \min(x_0, x_i). \tag{9.30}$$

Therefore, in the SHS analysis, it is essential to track variables such as $\min(x_0, x_1)$, $\min(x_0, x_2)$,

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$
1	$(1,1) \to (2,1)$	μ_1	$[x_0, x_1, x_2, x_3, x_4, x_5]$
2	$(1,1) \to (1,2)$	μ_1	$[x_0, x_1, x_2, x_3, x_4, x_5]$
3	$(1,2) \to (2,2)$	μ_1	$[x_0, x_1, x_2, x_3, x_4, x_5]$
4	$(2,1) \to (2,2)$	μ_1	$[x_0, x_1, x_2, x_3, x_4, x_5]$
5	$(2,1) \to (1,1)$	μ_2	$[x_3, 0, x_2, 0, x_5, 0]$
6	$(2,2) \to (1,2)$	μ_2	$[x_3, 0, x_2, 0, x_5, 0]$
7	$(1,2) \to (1,1)$	μ_2	$[x_4, x_1, 0, x_5, 0, 0]$
8	$(2,2) \to (2,1)$	μ_2	$[x_4, x_1, 0, x_5, 0, 0]$

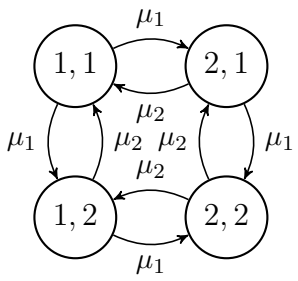


Figure 9.4.1: The SHS transition maps and Markov Chain corresponding to Parallel Sequential Synchronous Service (P-SSS) model.

and $\min(x_0, x_1, x_2)$ to accurately account for the delivery of fresh updates and the discarding of outdated ones. This approach to tracking age variables is inspired by the methodology proposed in [148]. We now proceed to describe the SHS analysis for the Parallel SSS model in detail.

The age of processed update for Parallel SSS model can be analyzed using the SHS Markov chain and table of state transitions shown in Fig. 9.4.1. The discrete state set is $\mathcal{Q} = \{(1,1), (2,1), (1,2), (2,2)\}$, where each tuple $(i,j) \in \mathcal{Q}$ represents the stage of Server 1 and Server 2, respectively. The continuous state age vector is $\mathbf{x} = [x_0, x_1, x_2, x_3, x_4, x_5]$, where x_0 is the age at monitor, x_1 and x_2 are the ages of the update at Server 1 and Server 2 respectively, $x_3 = \min(x_0, x_1)$, $x_4 = \min(x_0, x_2)$, and $x_5 = \min(x_0, x_1, x_2)$.

The SHS transitions are enumerated in the table in Fig. 9.4.1 and can be understood as follows:

- $l = 1, 2, 3, 4$: In these transitions, the servers only change stages; one of the servers finishes stage 1 and begins stage 2. Consequently, there is no reset in the age of updates at servers 1 and 2. Since no update is delivered to the monitor, the age at the monitor remains unchanged. As a result, the age variables x_3, x_4, and x_5 also remain unchanged.

- $l = 5, 6$: These transitions occur when Server 1 finishes processing and delivers the update to the monitor. Server 1 generates a fresh update, consequently, $x_1' = 0$. Since Server 2 continues to work on its update, $x_2' = x_2$. The age at the monitor is reset according to $\min(x_0, x_1)$, which is tracked by age variable x_3. Hence, $x_0' = x_3$. Since

$x'_1 = 0$, thus $x'_3 = 0$. The transition for age variable x_4 is more complex. We have $x'_4 = \min(x'_0, x'_2) = \min(x_3, x_2) = \min(\min(x_0, x_1), x_2) = \min(x_0, x_1, x_2) = x_5$. Further, $x'_5 = \min(x'_0, x'_1, x'_2) = \min(x_3, 0, x_2) = 0$.

- $l = 7, 8$: These transitions occur when Server 2 finishes processing and delivers the update to the monitor. Now Server 2 generates a new update, thus $x'_2 = 0$. Since Server 1 continues to work on its update, $x'_1 = x_1$. The age at the monitor is reset according to $\min(x_0, x_2)$, which is tracked by age variable x_4. Hence, $x'_0 = x_4$. Since $x'_2 = 0$, $x'_4 = 0$. Next, we have $x'_3 = \min(x'_0, x'_1) = \min(x_4, x_1) = \min(\min(x_0, x_2), x_1) = \min(x_0, x_1, x_2) = x_5$. Additionally, $x'_5 = \min(x'_0, x'_1, x'_2) = \min(x_4, x_1, 0) = 0$.

The Markov chain in Fig. 9.4.1 has stationary probabilities $\boldsymbol{\pi}$

$$\boldsymbol{\pi} = [\pi_{(1,1)}, \pi_{(1,2)}, \pi_{(2,1)}, \pi_{(2,2)}] = \frac{1}{(1+\rho)^2}[1, \rho, \rho, \rho^2]. \tag{9.31}$$

The age balance equations are

$$2\mu_1 \bar{\mathbf{v}}_{1,1} = \mathbf{1}\bar{\pi}_{1,1} + \mu_2 \bar{\mathbf{v}}_{2,1}\mathbf{A}_5 + \mu_2 \bar{\mathbf{v}}_{1,2}\mathbf{A}_7, \tag{9.32a}$$

$$(\mu_1 + \mu_2)\bar{\mathbf{v}}_{2,1} = \mathbf{1}\bar{\pi}_{2,1} + \mu_1 \bar{\mathbf{v}}_{1,1}\mathbf{A}_1 + \mu_2 \bar{\mathbf{v}}_{2,2}\mathbf{A}_8, \tag{9.32b}$$

$$(\mu_1 + \mu_2)\bar{\mathbf{v}}_{1,2} = \mathbf{1}\bar{\pi}_{1,2} + \mu_1 \bar{\mathbf{v}}_{1,1}\mathbf{A}_2 + \mu_2 \bar{\mathbf{v}}_{2,2}\mathbf{A}_6, \tag{9.32c}$$

$$2\mu_2 \bar{\mathbf{v}}_{2,2} = \mathbf{1}\bar{\pi}_{2,2} + \mu_1 \bar{\mathbf{v}}_{1,2}\mathbf{A}_4 + \mu_1 \bar{\mathbf{v}}_{1,2}\mathbf{A}_3. \tag{9.32d}$$

Solving these equations, the age at the monitor can be calculated as $\mathrm{E}[x_3] = v_{(1,1),0} + v_{(2,1),0} + v_{(1,2),0} + v_{(2,2),0}$.

$$\Delta_{\text{P-SSS}}(\mu_1, \mu_2) = \frac{1}{\mu_1} + \frac{1}{\mu_2} + \frac{1}{4(\mu_1+\mu_2)}\left(1 + \frac{\mu_1}{\mu_2} + \frac{\mu_2}{\mu_1}\right) + \frac{1}{4}\left(\frac{\mu_1\mu_2(\mu_1+2\mu_2)(\mu_2+2\mu_1)}{(\mu_1+\mu_2)^5}\right). \tag{9.33}$$

Setting $\rho = \mu_1/\mu_2$, we get

$$\Delta_{\text{P-SSS}}(\mu_2, \rho) = \frac{1}{\mu_2}\left(1 + \frac{1}{\rho} + \frac{1}{4(1+\rho)}\left(1 + \rho + \frac{1}{\rho}\right)\right) + \frac{1}{4\mu_2}\left(\frac{\rho(1+2\rho)(2+\rho)}{(1+\rho)^5}\right). \tag{9.34}$$

The age expression in (9.33) exhibits symmetry in μ_1 and μ_2 because each server operates independently with its service time being a convolution of two exponential random variables with rates μ_1 and μ_2. Swapping these rates leaves the distribution of a server's service time unchanged, reflecting the symmetry in the age expression.

Further, observe that as $\mu_1 \to \infty$, $\Delta_{\text{P-SSS}} \to 1.25/\mu_2$. This result aligns with the following intuition: When $\mu_1 \to \infty$ and μ_2 is finite , step 1 is almost instantly completed, and each server delivers an update with an average age $1/\mu_2$. If there were only one server, this would correspond to an average age of $2/\mu_2$ at the monitor, since each update is delivered on average after a duration of $1/\mu_2$. However, if a single server were running step 2 at twice the speed ($2\mu_2$), the age at the monitor would be $1/\mu_2$ as $\mu_1 \to \infty$. However, we are not simply running one server at double speed; rather, we are running a system with two parallel servers. Each server processes an update that is slightly older, resulting in an average age of $1.25/\mu_2$ rather than $1/\mu_2$.

For P-SSS model, the sets $\mathcal{U}$ and $\mathcal{V}$ are defined as $\mathcal{U} = \{(1,1),(1,2),(2,1)\}$ and $\mathcal{V} = \{(1,2),(2,1),(2,2)\}$, respectively. The total power consumption for step 1 is then

$$\sum_{u\in\mathcal{U}} \pi_u r_u(\mu_1, \alpha) = \mathrm{E}[C]^\alpha(\pi_{(1,1)}(2\mu_1^\alpha) + \pi_{(1,2)}\mu_1^\alpha + \pi_{(2,1)}\mu_1^\alpha). \tag{9.35}$$

Since $\pi_{(1,2)} = \pi_{(2,1)}$, it follows from (9.35) that

$$\sum_{u\in\mathcal{U}} \pi_u r_u(\mu_1, \alpha) = \mathrm{E}[C]^\alpha(\pi_{(1,1)} + \pi_{(1,2)})(2\mu_1^\alpha). \tag{9.36}$$

Similarly, the total power consumption for step 2 is

$$\sum_{v\in\mathcal{V}} \pi_v r_v(\mu_2, \alpha) = \mathrm{E}[C]^\alpha(\pi_{(1,2)}\mu_2^\alpha + \pi_{(2,1)}\mu_2^\alpha + \pi_{(2,2)}(2\mu_2^\alpha)),$$

$$= \mathrm{E}[C]^{\alpha}(\pi_{(1,2)} + \pi_{(2,2)})(2\mu_2^{\alpha}). \tag{9.37}$$

Thus, it follows from (9.27), (9.36) and (9.37) that the total power consumption for an update processing in the P-SSS model is constrained as

$$\mathrm{E}[C]^{\alpha}\Big((\pi_{(1,1)} + \pi_{(1,2)})(2\mu_1^{\alpha}) + (\pi_{(1,2)} + \pi_{(2,2)})(2\mu_2^{\alpha})\Big) \leq P. \tag{9.38}$$

Using (9.31) and the fact that $\rho = \mu_1/\mu_2$, some algebraic manipulations on (9.38) yields

$$\mu_2^{\alpha}\Big(\frac{\rho^{\alpha}}{1+\rho} + \frac{\rho}{1+\rho}\Big) \leq \frac{P}{2\,\mathrm{E}[C]^{\alpha}}. \tag{9.39}$$

Based on (9.39), the upper bound on the service rate of step 2 in P-SSS model is

$$\mu_2 \leq \frac{1}{\mathrm{E}[C]}\left(\frac{P}{2}\frac{1+\rho}{\rho(\rho^{\alpha-1}+1)}\right)^{1/\alpha}. \tag{9.40}$$

We then minimize $\Delta_{\text{P-SSS}}(\mu_2, \rho)$ in (9.34) subject to the constraint on μ_2 given in (9.40).

9.4.2 Parallel Coordinated Alternating Freshness (P-CAF)

In this policy, only one server is allowed to work on stage 2 of update processing at a time. When both servers i and j are in stage 1, they process the same fresh update concurrently. If Server i transitions to stage 2, then Server j restarts stage 1 with a fresh update. If Server j reaches stage 2 with its fresher update before Server i completes its processing, then Server i will abort its current task and restart in stage 1 with a fresh update. This mechanism ensures that the update in stage 1 is always the freshest.

We analyze the age performance of P-CAF policy using SHS. The Markov state space is defined as $\mathcal{Q} = \{1, 2\}$, where state 1 corresponds to both servers being in stage 1, while state 2 indicates that one server is in stage 2 and the other in stage 1. The continuous age vector is $\mathbf{x} = [x_0, x_1, x_2]$, where x_0 corresponds to age at the monitor, and x_1 denotes the age of update currently in stage 1,

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$
1	$1 \to 2$	$2\mu_1$	$[x_0, 0, x_1]$
2	$2 \to 2$	μ_1	$[x_0, 0, x_1]$
3	$2 \to 1$	μ_2	$[x_2, 0, x_2]$

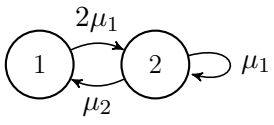


Figure 9.4.2: The SHS transition maps and Markov Chain corresponding to Parallel Coordinated Alternating Freshness (P-CAF) model.

and x_2 denotes the age of update being processed in stage 2.

Note that, in contrast to the P-SSS parallel server model where both servers work independently and may deliver stale updates, the P-CAF policy allows coordination among the servers to ensure that only the freshest update that has completed two steps of processing is delivered. This coordination eliminates the possibility of useless deliveries, making every delivery useful and minimizing age. Mathematically, the analysis is simplified since we don't need to track age variables such as $\min(x_i, x_3)$, as the policy inherently guarantees that the update delivered from stage 2 is always the freshest one. By design, the update in stage 2 is always the latest to reach this stage, and any fresher update will be in stage 1, awaiting its turn to enter stage 2.

The SHS Markov chain and table of state transitions are shown in Fig. 9.4.2.

- $l = 1$: Transition from state 1 to state 2 at rate $2\mu_1$. In state 1, both servers are in stage 1. The time until one server finishes stage 1 is the minimum of two independent exponential distributions with rate μ_1, resulting in a departure rate of $2\mu_1$ from state 1. Upon transition, one server moves to stage 2, so $x_2' = x_1$, while the other server restarts in stage 1 with a fresh update, thus $x_1' = 0$. The age at the monitor, x_0, remains unchanged since no update has been delivered, so $x_0' = x_0$.

- $l = 2$: Server in stage 1 finishes service to reach stage 2, and the server in stage 2 (with an older update) restarts in stage 1 with a fresh update. The age of the update in stage 1 is reset to 0, so $x_1' = 0$. The age of the update now being processed in stage 2 is updated to the age of the previous update in stage 1, so $x_2' = x_1$. The age at the monitor, x_0, remains unchanged, so $x_0' = x_0$.

- $l = 3$: Server in stage 2 finishes service and delivers the processed update to the monitor. The age at the monitor is updated to the age of the update that was in stage 2, so $x_0' = x_2$. The server that finished in stage 2 as well as the server that was in stage 1 restart at stage 1, and both servers now work on the same fresh update in stage 1, thus $x_1' = 0$.

The Markov Chain in Fig. 9.4.2 has stationary probabilities

$$\pi_1 = \frac{1}{1+\rho}, \quad \text{and} \quad \pi_2 = \frac{\rho}{1+\rho}. \tag{9.41}$$

The age balance equations are

$$2\mu_1[v_{10}\ v_{11}\ v_{12}] = [\pi_1\ \pi_1\ \pi_1] + \mu_2[v_{22}\ 0\ v_{22}], \tag{9.42a}$$

$$(\mu_1+\mu_2)[v_{20}\ v_{21}\ v_{22}] = [\pi_2\ \pi_2\ \pi_2] + \mu_1[v_{20}\ 0\ v_{21}] + 2\mu_1[v_{10}\ 0\ v_{11}]. \tag{9.42b}$$

Solving the set of equations in (9.42), we can obtain v_{qj}. The age at the monitor is $\Delta_{\text{P-CAF}} = v_{10}+v_{20}$, which gives us:

$$\Delta_{\text{P-CAF}} = \frac{1}{\mu_2}\Big(\frac{3}{2(1+\rho)} + \frac{2\rho}{1+2\rho} + \frac{1+\rho+\rho^2}{\rho(1+\rho)^2}\Big). \tag{9.43}$$

For P-CAF policy, $\mathcal{U} = \{1, 2\}$. Since there is only one state corresponding to at least one server in stage 2, the set $\mathcal{V}$ is a singleton set i.e., $\mathcal{V} = \{2\}$. The power consumption in executing step 1 is then

$$\sum_{u\in\mathcal{U}} \pi_u r_u(\mu_1, \alpha) = \mathrm{E}[C]^\alpha(\pi_1(2\mu_1^\alpha) + \pi_2\mu_1^\alpha). \tag{9.44}$$

Similarly, the power consumed in executing step 2 is

$$\sum_{v\in\mathcal{V}} \pi_v r_v(\mu_2, \alpha) = \pi_2\mu_2^\alpha\, \mathrm{E}[C]^\alpha. \tag{9.45}$$

It follows from (9.27), (9.44) and (9.45) that the total power consumption for an update processing

l	$q_l \to q_l'$	$\lambda^{(l)}$	$\mathbf{x}\mathbf{A}_l$
1	$1 \to 2$	$2\mu_1$	$[x_0, x_1]$
2	$2 \to 1$	$2\mu_2$	$[x_1, 0]$

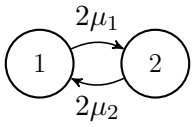


Figure 9.4.3: The SHS transition maps and Markov Chain corresponding to Parallel Shared Intermediate Update (P-SIU) model.

in is constrained as

$$\pi_1(2\mu_1^\alpha) + \pi_2(\mu_1^\alpha + \mu_2^\alpha) \leq \frac{P}{\mathrm{E}[C]^\alpha}. \tag{9.46}$$

Simplifying (9.46) yields

$$\mu_1^\alpha + \pi_1\mu_1^\alpha + \pi_2\mu_2^\alpha \leq \frac{P}{\mathrm{E}[C]^\alpha}. \tag{9.47}$$

With $\rho = \mu_1/\mu_2$ and using (9.41), the upper bound on μ_2 can be obtained from (9.47),

$$\mu_2 \leq \frac{1}{\mathrm{E}[C]}\left(\frac{P(1+\rho)}{\rho + \rho^\alpha(2+\rho)}\right)^{1/\alpha}. \tag{9.48}$$

We minimize (9.43) subject to the constraint on μ_2 in (9.48).

9.4.3 Parallel Shared Intermediate Updates (P-SIU)

This policy leverages server-to-server communication to share intermediate processing results, enabling parallel processing of updates. Under the P-SIU policy, both servers begin by working on stage 1 of the same update. When one server completes stage 1 and transitions to stage 2, it shares the intermediate result of stage 1 with the other server. Consequently, both servers then start working on the intermediate update in stage 2 simultaneously. When either one of the update completes stage 2 then both servers restart with a fresh update.

The P-SIU policy effectively transforms the parallel server system into a system that behaves like a single server. In this equivalent single-server system, the service times for stage 1 and stage 2 are exponential with rate $2\mu_1$ and $2\mu_2$. This simplification arises because the policy ensures that both servers are always working in parallel on the same update, whether in stage 1 or stage 2.

The discrete state space of the system is defined as $\mathcal{Q} = \{1, 2\}$, where state 1 corresponds to

both servers in stage 1, and state 2 correspond to both servers in stage 2. The continuous age vector is $\mathbf{x} = [x_0, x_1]$, where x_0 is the age of the update at the monitor, and x_1 is the age of the update being processed by the servers. The SHS Markov Chain and transitions are illustrated in Fig. 9.4.3 and are self-explanatory. Additionally, the Markov Chain in Fig. 9.4.3 has stationary probabilities

$$\pi_1 = \frac{1}{1+\rho}, \quad \text{and} \quad \pi_2 = \frac{\rho}{1+\rho}. \tag{9.49}$$

The age at the monitor is $\Delta_{\text{P-SIU}} = v_{10} + v_{20}$, which is expressed as:

$$\Delta_{\text{P-SIU}} = \frac{1}{\mu_2}\Big(1 + \frac{1}{2\rho}(1 + \frac{1}{1+\rho})\Big). \tag{9.50}$$

For the P-SIU policy, we have $\mathcal{U} = \{1\}$ and $\mathcal{V} = \{2\}$, and thus the power consumption for executing step 1 in P-SIU policy will be

$$\sum_{u\in\mathcal{U}} \pi_u r_u(\mu_1, \alpha) = \pi_1(2(\mathrm{E}[C]\mu_1)^\alpha). \tag{9.51}$$

Similarly, the power consumed in executing step 2 is

$$\sum_{v\in\mathcal{V}} \pi_v r_v(\mu_2, \alpha) = \pi_2(2(\mathrm{E}[C]\mu_2)^\alpha). \tag{9.52}$$

It follows from (9.27), (9.51) and (9.52) that the total power consumption for an update processing in P-SIU is constrained as

$$\pi_1(2\mu_1^\alpha) + \pi_2(2\mu_2^\alpha) \leq \frac{P}{\mathrm{E}[C]^\alpha}. \tag{9.53}$$

With $\rho = \mu_1/\mu_2$ and using the stationary probabilities given in (9.49), we derive the following upper bound on μ_2 from (9.53):

$$\mu_2 \leq \frac{1}{\mathrm{E}[C]}\left(\frac{P}{2}\frac{1+\rho}{\rho(\rho^{\alpha-1}+1)}\right)^{1/\alpha}. \tag{9.54}$$

Finally, we minimize $\Delta_{\text{P-SIU}}(\mu_2, \rho)$ as given in (9.50), subject to the constraint in (9.54).

9.5 Numerical Evaluation

In this section, we address the optimization problem presented in (9.8) and (9.28) for the various queueing models identified in Section 9.3 and Section 9.4. To illustrate the methodology, we begin with a detailed analysis of the M/M/1* system. This example will demonstrate the process of solving the optimization problem by substituting the upper bound on μ_2 into the objective function $\Delta(\mu_2, \rho)$. While we provide a detailed analysis for the M/M/1* model as an illustrative example, the optimization process for other models follows a similar approach. To avoid redundancy, we do not present explicit analyses for each model. Instead, we present the results of the numerical evaluation for all models, which have been derived using the same methodology.

To solve the M/M/1*, we use (9.9), (9.10) and (9.12). With $\alpha = 3$, the optimization problem in (9.8) can then be reformulated as:

$$\text{minimize} \quad \frac{1}{\mu_2}\left(1 + \frac{1}{\rho}\right) \tag{9.55a}$$

$$\text{subject to} \quad \mu_2 \leq \frac{1}{\mathrm{E}[C]}\left(\frac{P}{\rho^3 + \frac{\rho}{1+\rho}}\right)^{1/3}, \tag{9.55b}$$

$$\mu_2 \geq 0, \text{and } \rho \geq 0. \tag{9.55c}$$

We aim to solve (9.55) with respect to the variable ρ. Substituting μ_2 from the constraint (9.55b), into the objective function (9.55a), we get

$$\text{minimize} \quad f_{\text{M/M/1}^*}(\rho), \tag{9.56}$$

where

$$f_{\text{M/M/1}^*}(\rho) = \frac{\mathrm{E}[C]}{P^{1/3}}\left(\rho^3 + \frac{\rho}{1+\rho}\right)^{1/3}\left(1 + \frac{1}{\rho}\right). \tag{9.57}$$

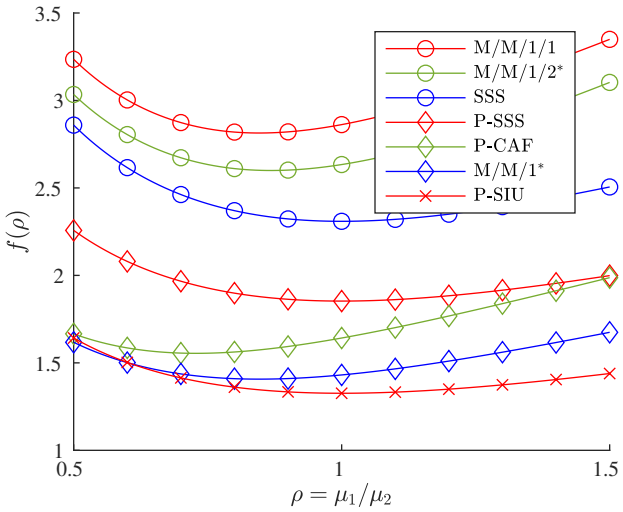


Figure 9.5.1: Plot of objective function of constrained optimization as a function of ρ. Here $P = 8, \mathrm{E}[C] = 1$, and $\alpha = 5$.

Taking derivative of $f(\rho)$ in (9.57) with respect to ρ, and setting $f'(\rho) = 0$, we obtain

$$\rho^3(1+\rho) = 2/3. \tag{9.58}$$

Solving (9.58) gives the optimal $\rho^* = 0.7279$. Now $\rho^* < 1$, implies that Server 2 should operate faster than Server 1, as a slow Server 2 becomes a bottleneck in update processing.

In general, for a fixed P, $\mathrm{E}[C]$ and α, let $\mu_2^* = g_{\text{model}}(\rho)$, where for each model in servers in series configuration, μ_2^* is given by the equivalent right side of (9.8b) and for parallel models, μ_2^* is given by the right side of (9.40), (9.48), and (9.54). Then the objective function for each model will be

$$f_{\text{model}}(\rho) = \frac{\Delta_{\text{model}}(\rho)}{g_{\text{model}}(\rho)}. \tag{9.59}$$

Figure 9.5.1 illustrates the plot of the objective function $f(\rho)$ in (9.59) as a function of ρ for all models studied in this work. First, we observe that for all models, the optimal $\rho^* \leq 1$. This indicates that Server 2 should be faster. For instance, a faster Server 2 in M/M/1/2* suggests that the queue at Server 2 is quickly being served, which is favorable for minimizing the age. The optimal ρ^* for M/M/1/1 is the same as that in M/M/1* because $p_1(\rho)$ and $p_2(\rho)$ are the same, and the age $\Delta_{\text{M/M/1/1}}(\mu_2, \rho)$ in (9.22) is simply double the age $\Delta_{\text{M/M/1*}}(\mu_2, \rho)$ in (9.12). Further, for the SSS,

P-SSS and P-SIU models, $\rho^* = 1$ suggests that step 1 and step 2 processing should be done at the same rate. This makes sense due to the symmetry: a slower service at step 1 would delay step 2, and a slower service at step 2 would keep the system waiting to generate a new update. Both scenarios are sub-optimal for minimizing age.

We observe that, across all considered models, the optimal ρ^* is independent of the power constraint P. The illustrative example of M/M/1* mathematically justifies this, as minimizing the objective function (9.57) will be independent of P. A more conceptual reasoning is as as follows. In this work, we have considered a restrictive class of systems where increasing μ_2 and $\mu_1 = \rho\mu_2$ improves age performance. In the examined systems, when ρ is fixed, then increasing the service rate at Server 2 is always age reducing as is evident from (9.12), (9.18), (9.22) (9.26), (9.34), (9.43) and (9.50). Therefore, the optimal μ_2 should be as large as possible while ensuring that the energy consumed by servers 1 and 2 satisfies the power constraint .

We note that this independence of ρ^* from P might not hold for all systems. For instance, consider a system where Server 1 generates at will with zero wait and serves at rate μ_1, and updates are queued at Server 2. The performance of Server 2 is known if the updates arrive fresh [3]. However, in our scenario, updates arrive with some age from Server 1, making the system more complex. A longer inter-arrival time between updates can slightly empty the queue at Server 2, but the updates arrive with more age, as the inter-arrival times reflect the age of the updates. Hence, it is not straightforward to say that increasing μ_2 and $\mu_1 = \rho\mu_2$ will always minimize age, and as such there could be some optimal service rates ratio ρ^* which could depend on the power budget P.

Fig. 9.5.2 numerically compares optimal age performance of all the models in terms of power constraint P. As expected, increasing P leads to a larger optimal μ_2^* resulting in a decrease in age due to the faster service rate. It is apparent from Fig. 9.5.2 that preemption in service yields better age performance among all servers in series models, which aligns with the existing view in the AoI literature that preemption of old updates by new ones is always beneficial. An interesting and somewhat surprising finding is that synchronous service at servers performs better than asynchronous service, indicating that having a single update in the system being serviced

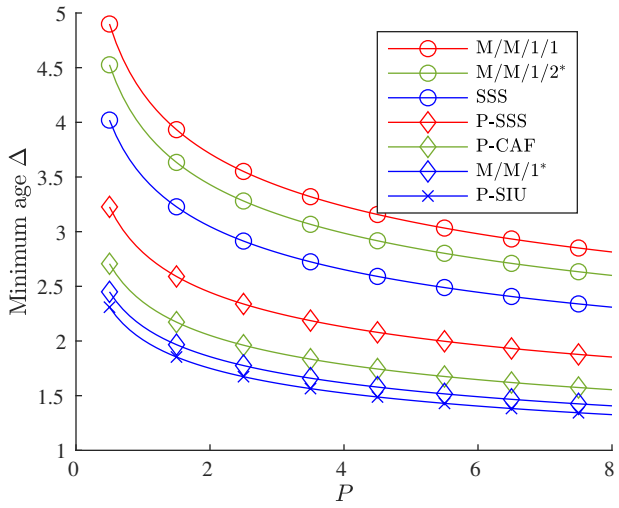


Figure 9.5.2: Optimal age $\Delta(\mu_2^*, \rho^*)$ for servers in series and parallel setups under power constraint P. Here, $\alpha = 5$ and, $\mathrm{E}[C] = 1$.

is more advantageous than having multiple updates in progress. This observation makes sense upon further reflection: synchronous servers prevent updates from lingering in the waiting queue at Server 2 (as in the M/M/1/2* model), or causing Server 2 to be idle more frequently which occurs when updates are frequently discarded (as in the M/M/1/1 model).

Fig. 9.5.2 demonstrates that the P-SSS model achieves better age performance compared to the SSS model, highlighting the advantages of parallel processing over sequential processing. P-SSS can be viewed as a parallelized version of SSS, where independent servers operate in a manner similar to SSS but with each server consuming half of the total power budget ($P/2$). Additionally, the superior performance of P-CAF and P-SIU compared to P-SSS further underscores the benefits of incorporating additional information about the state of the other server.

9.6 Open Problems: A Discussion

A central theme emerging from this work is the concept of "wasted power" in the context of parallel server systems. To understand this concept, imagine a scenario where a supervisor assigns the same unsolved problem to multiple researchers, each working independently to find a solution. If one researcher has a breakthrough and solves the problem first, the efforts of the others who were also working diligently but did not finish first, might seem wasted. However, from the outset, it's

impossible to predict which researcher will be successful first. Thus, while some of the work may not directly contribute to the final solution, it is not necessarily wasted since it increases the overall probability of success.

We define a server's work as "useless" if the update being processed is older than the age at the monitor. That is, if a server had full knowledge of the state of the monitor – specifically, whether the monitor has a fresher update than the server – the server would choose not to continue its current processing. Among parallel server setups, it appears that there is no wasted power in the P-SIU and P-CAF policies. Both of these policies involve full sharing of processing stages between servers. At any given moment, no server is working on update older than what the monitor has, ensuring that all efforts contribute to reducing the age at the monitor.

In contrast, the P-SSS setup exhibits clear wasted effort. In this setup, the two servers work independently on processing updates, without knowledge of the other's progress. If the server working on the older update had information that the other server has already delivered a fresher update, it would refrain from continuing its work, recognizing that its efforts are redundant. However, in the P-SSS setup, this information is not shared, leading to situations where one server's work becomes unnecessary and ultimately wasted.

In series server setups, the effort of Server 1 is considered wasted if Server 2 continues working on an older update despite the availability of a fresh update that has completed step 1. In the M/M/1* model, wasted effort is effectively avoided, as Server 2 always prioritizes the freshest update from Server 1. If Server 2 is busy when Server 1 finishes processing a new update, Server 2 preempts its current task and immediately begins processing the fresher update from Server 1. This mechanism ensures that Server 2's efforts are always directed toward the most recent data, preventing any wasted power.

However, wasted power does occur in the M/M/1/1 and M/M/1/2* models. In M/M/1/1, if Server 2 is busy when Server 1 completes processing, the update from Server 1 is discarded and also Server 2 is not necessarily working on the freshest update from Server 1. In M/M/1/2*, Server 1 can deliver an update that is then queued at Server 2. However, this queued update may be preempted

and discarded from a new arrival from Server 1. As a result, the effort and energy expended by Server 1 on the now-discarded update are rendered useless. Similar to the M/M/1/1 model, Server 2 in this setup is not always working on the freshest update from Server 1.

In the Synchronous Sequential Service (SSS) model, there doesn't appear to be direct wasted effort as seen in other setups. However, there are still significant opportunities for optimization. In SSS, if Server 1 takes an unusually long time to complete stage 1, it may be more efficient to discard the current update and start processing a new one. Sometimes, when the SSS delivers an update too quickly then the newly generated update is almost identical to the one just delivered. Thus, the effort spent on processing and delivering the second update may not provide significant value. Studying threshold based waiting strategies in SSS similar to that in [52] is an area of future research.

For other models, our analysis assumes a generate-at-will scenario with zero-wait at servers. In the existing literature on optimal waiting strategies [52, 53], it is typically assumed that there is just a single update in the service facility. Upon delivery of this update, the decision to wait is then considered. However, our system is more complex, as we allow multiple updates to be in process simultaneously, either in parallel or in sequential servers. Consequently, the optimality (or even desirability) of the known non-zero wait strategies, such as setting a threshold based on prior service time, is unresolved. While we acknowledge that waiting strategies could be studied and potentially employed, this investigation is beyond the scope of our current work, and is a topic of for future research.

In our current analysis, the optimization of service rates in parallel models is performed under the constraint that both servers operate at the same service rates, denoted as μ_1 and μ_2. However, this approach could be further refined by considering a more granular optimization involving four variables: $\mu_{11}, \mu_{12}, \mu_{21}, \mu_{22}$, where μ_{ij} represents Server i working on stage j.

Moreover, an even more sophisticated approach would involve studying an online policy where service rates are dynamically adjusted based on the current state of the system. For instance, in the system, when one server transitions to stage 2, should the other server now working in stage 1

with a fresher update work faster than before? Such an adaptive strategy could be studied using a Markov Decision Process (MDP) framework, where the optimal service rate is chosen based on the system's state at any given time.

Another promising avenue for future research lies in designing optimal policies for updates that require more than two processing steps. Moreover, the current work has a natural extension where where each processing step has a general service time distribution, rather than the exponential service times assumed here. The current SHS methodology has a limitation of being applicable to systems with memoryless regimes. Developing a novel SHS analysis for a general service time will not only be useful to this work, but in general to the AoI community.

9.7 Conclusion

This work explored the timely processing of updates that require a sequence of computational steps. We specifically examined the parallel and series configurations of servers deployed for update processing, with a focus on understanding the age-power trade-off in the special case of two-step update processing. To achieve this, we formulated and solved optimization problems that determine the optimal service rates for each step, constrained by a total power budget, to minimize the average age. Our optimization problem formulations were applied to various servers in series (tandem queue) models, as well as to different policies in parallel servers. Our analysis revealed that synchronous sequential execution generally outperforms asynchronous sequential execution. Additionally, we observed that parallel servers tend to outperform pipelines of servers (servers in series) in terms of AoI.

Part V

Future Work

CHAPTER 10
CONCLUSIONS AND FUTURE WORK

In this thesis, we studied the challenges of ensuring timely writing, reading, and processing of updates in producer-consumer systems using shared memory as a means of information dissemination. Throughout the thesis, we focused on a consistent model: source updates are written into the memory (the source-writer pair acts as producers), clients require access to these updates for computation, and the reader fulfills clients' requests by retrieving updates from memory (the reader-client pair acts as consumers). After processing/computing on the source update, the client outputs a processed update.

While shared memory offers an efficient mechanism for data sharing, it also introduces significant bottlenecks, including the potential for readers to access stale updates due to discrepancies in the timing between when updates are written and when they are read, as well as delays introduced by synchronization primitives. These bottlenecks impact the timeliness of all three operations involved in the producer-consumer paradigm: storing, retrieving, and processing. In this thesis, we proposed optimization strategies for writer, reader and client operations with the aim of making them suitable for timely status updating systems. In the subsequent sections, we elucidate the identified bottlenecks, corresponding research problems, proposed solutions, and future work.

10.1 On Efficient and Timely Memory Access

In Chapter 3, our the focus was on optimizing read operations. One significant bottleneck lies in the cost associated with sampling/reading memory , where the reader's aim to reduce update age may be outweighed by the sampling cost. We addressed the research problem of determining the optimal instances for memory sampling, considering the trade-off between age reduction and sampling cost. In Chapter 3, we assumed that the Reader could always ascertain the memory state through inexpensive timestamp retrievals. This effectively meant that the Reader was immediately aware of

any new writes to the memory, simplifying the representation of the system state to a tuple (x, y), where x denoted the age of the object in memory, and y denoted the age of the update at the client's input. Under this assumption, we established that the decision to sample only occurred during the time slots when the memory was updated, or equivalently, when the age of the update was zero. This known memory state assumption reduced the decision-making process to considering only the state $(0, y)$. Consequently, the optimal sampling policy exhibited an elegant threshold structure characterized by an optimal threshold Y_0^*, where the Reader would sample in state $(0, y)$ if $y \geq Y_0^*$, otherwise, it would remain idle in all other states.

Future Work: Timely Reading with Multiple Sources

Consider a system where multiple source updates are published by multiple writers and each source update is mapped to a particular reader. The question is which Reader gets to read in a slot such that the average age at all Readers is minimized.

An RMAB problem involves N independent bandit arms, where each arm represents an option or action that can be taken. When an arm is pulled, its state changes according to Markovian transition distributions, and when not pulled, the state evolves according to a different transition rule. The evolution of each arm's state is thus "restless", meaning it continues to evolve even when the arm is not selected. With respect to the problem of scheduling among Readers, the age at each Reader evolves similarly to a restless bandit, whether or not the Reader samples the memory. Consequently, such a problem can be studied with a Restless Multi-Armed Bandit (RMAB) problem framework. The goal of the RMAB framework will be to find a scheduling policy that selects one out of N Readers in each time slot such that the total time average cost of the system is minimized.

Whittle's Index is a heuristic policy used to solve a restless multi-armed bandit problem. In our physical system, where multiple source updates are published and an oracle must decide which reader to schedule for reading its corresponding source update, the charge acts as the Lagrangian. If we have only one source-reader pair, setting the charge to zero ensures that the reader always reads immediately upon receiving a new update. However, when multiple readers are involved

and multiple source updates arrive simultaneously, memory access collisions occur, necessitating a decision on which reader should read. In such cases, there might be an optimal policy to decide which Reader gets to read and potentially a Whittle's Index policy, if the systems satisfies a special property called indexability.

10.2 On Timely Processing of Source Updates

In Chapter 5, our focus was directed towards optimizing client operations, particularly concerning the timely processing of updates from multiple sources. We established a model where a writer publishes updates from two independent sources into shared memory, and decision updates are subsequently derived by a Decision Process (DP) through memory reading. Despite the DP operating independently of how the source updates are recorded in memory, our analysis showed that a lazy sampling policy employed by DP Reader can notably improve the timeliness of decision updates as it offsets the negative impact of high variance computation times.

Future Work: Age-Dependent Computation

An inherent limitation of our current system model arises when stale or redundant updates are read from memory. Despite this, the DP continues with computation on the stale update, even when the resulting decision update does not significantly reduce the age at the monitor. It would be more efficient for the DP to discard stale updates and wait an optimized time before reading the memory. Consequently, we propose the exploration of an age-dependent computation model in future work. This involves analyzing the age-dependent computation model, attempting to design an optimal computation policy, and subsequently comparing this new model with the existing lazy computation policy. Such investigations will facilitate the development of more effective and efficient computation policies in producer-consumer systems.

10.3 On the Impact of Synchronization Primitives

Chapters 6, 7, and 8 explored the impact of synchronization primitives on timely updating. We modeled and developed a packet forwarding experiment 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 experimental platform. While timeliness of the location updates in the table is desirable, excessive updating can be at the expense of timely reading of the table.

In the case of RCU, we also studied the memory footprint associated with lockless operation. For RCU, our conclusions regarding age were consistent with the general conclusion of prior work [100, 135, 149] that RCU tends to improve latency (because of non-blocking readers and no mutual exclusion among readers and writers), but this improvement is at the expense of using more memory. From a timeliness perspective, frequent updating keeps information fresher, however, this increases the memory overhead associated with more updates in grace periods.

Future Work: Loosely-Coupled Source and Writer

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 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 [128, 52] 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. As such, the future work will constitute characterizing the impact of RCU and RWL on average age of source updates in memory in a loosely-coupled source-writer setting.

Future Work: Age-Latency Trade-off

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 in RWL, the reader might have to wait to access 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. This setup is reminiscent of the cache updating system studied in [150], where a user retrieves a file either from a local cache or directly from a remote source. The authors explored the trade-off between obtaining a file from a limited capacity cache, which might contain an older version, and directly retrieving a fresh copy from the remote source, at the cost of additional transmission time. They determined whether a file should be stored in the cache and established the corresponding update rate to minimize the overall age at the user.

The two systems, however, differ in a key aspect. The work in [150] assumes that the remote server always has the freshest file available. In contrast, with a shared memory system, it is possible that the writer is locked out, preventing it from updating the shared memory with a fresher update. As a result, the main memory could be outdated, adding another layer of complexity to the age-latency trade-off in such systems.

We observe that RCU and RWL in general admit a combinatorial explosion of system models 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.

Future Work: Memory Usage in RCU

With respect to RCU, we found that for a fixed updating (writing) rate, the memory footprint grows with the rate of read requests. When memory is scarce, this can be problematic. There are many potential solutions. One way the memory usage can be regulated is by limiting the update rate α of the writer, albeit at the expense of high AoI. Alternatively, the memory footprint can be reduced by controlling the read request rate λ.

Consider a setting in which an application (such as SLAM) processes an update in a number of modules that can be executed either concurrently or sequentially. In this setting, sequential operation can effectively reduce the read request rate λ through various methods. For example, the modules may generate individual read requests, but the sequential execution of the modules will slow the overall update processing rate. In an alternate approach that utilizes local copies, the application makes a single read to store the current data item in a local copy. This local copy is then utilized by the subsequent module executions. Although a caveat can be that for large objects in the memory, the reads take longer. If a process maintains a local copy, subsequent steps circumvent this read latency, but the data used will be stale if the main memory has been updated.

Another approach is to consider the constrained RCU mechanism (RCU-C) that restricts the total number of update versions in the memory to be at most two (old and new). This could work almost as well as unconstrained RCU, particularly at low updating rates. However, at higher write rates, the copy constraint acts similarly to a read lock, in that the writer will enter a write-pending

state waiting for the grace period of the old copy to expire. An analytical model of this effect would characterize the tradeoff between age and memory consumption in RCU-C updating systems.

REFERENCES

[1] M. C. Potter, B. Wyble, C. E. Hagmann, and et al., "Detecting meaning in RSVP at 13 ms per picture," *Attention, Perception, and Psychophysics*, vol. 76, pp. 270–279, 2014.

[2] S. Kaul, M. Gruteser, V. Rai, and J. Kenney, "Minimizing Age of Information in Vehicular Networks," in *IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2011.

[3] S. Kaul, R. Yates, and M. Gruteser, "Real-time status: How often should one update?" In *Proc. IEEE INFOCOM*, Mar. 2012, pp. 2731–2735.

[4] 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.

[5] 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.

[6] E. W. Dijkstra, "Cooperating sequential processes," in *The origin of concurrent programming: from semaphores to remote procedure calls*, Springer, 2002, pp. 65–138.

[7] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "User-level interprocess communication for shared memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 2, pp. 175–198, 1991.

[8] S. Shi, C.-H. Hsu, K. Nahrstedt, and R. Campbell, "Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming," in *Proceedings of the 19th ACM international conference on Multimedia*, 2011, pp. 103–112.

[9] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, 1990, pp. 168–176.

[10] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for MapReduce," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, SIAM, 2010, pp. 938–948.

[12] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995.

[13] M. K. Aguilera, N. Ben-David, I. Calciu, R. Guerraoui, E. Petrank, and S. Toueg, "Passing messages while sharing memory," in *Proceedings of the 2018 ACM symposium on principles of distributed computing*, 2018, pp. 51–60.

[14] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui, "Shared memory vs message passing," Tech. Rep., 2003.

[15] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "Message passing versus distributed shared memory on networks of workstations," in *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, IEEE, 1995, pp. 37–37.

[16] J. Nelson *et al.*, "Latency-Tolerant Software Distributed Shared Memory," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 291–305.

[17] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A high-performance, heterogeneous MPI," in *2006 IEEE International Conference on Cluster Computing*, IEEE, 2006, pp. 1–9.

[18] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken, "Low-latency communication on the IBM RISC System/6000 SP," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, 1996, 24–es.

[19] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[20] 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.

[21] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, Citeseer, vol. 509518, 1998, pp. 509–518.

[22] D. Buono and G. Mencagli, "Run-time mechanisms for fine-grained parallelism on network processors: The TILEPro64 experience," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 55–64.

[23] K. Jeffay, "The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems," in *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, 1993, pp. 796–804.

[24] D. Raychaudhuri *et al.*, "Challenge: COSMOS: A City-Scale Programmable Testbed for Experimentation with Advanced Wireless," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. New York, NY, USA: Association for Computing Machinery, 2020, ISBN: 9781450370851.

[25] 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.

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

[27] R. D. Yates, "Age of Information in a Network of Preemptive Servers," in *IEEE Conference on Computer Communications (INFOCOM) Workshops*, arXiv preprint arXiv:1803.07993, Apr. 2018, pp. 118–123.

[28] 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*, Apr. 2018, pp. 112–117.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] M. Moltafet, M. Leinonen, and M. Codreanu, "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.

[35] S. Kaul, R. Yates, and M. Gruteser, "Status Updates Through Queues," in *Conf. on Information Sciences and Systems (CISS)*, Mar. 2012.

[36] K. Fraser, "Practical lock-freedom," University of Cambridge, Computer Laboratory, Tech. Rep., 2004.

[37] 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.

[38] P. H. Gum, "System/370 extended architecture: facilities for virtual machines," *IBM Journal of Research and Development*, vol. 27, no. 6, pp. 530–544, 1983.

[39] Gottlieb, Grishman, Kruskal, McAuliffe, Rudolph, and Snir, "The NYU ultracomputer—Designing an MIMD shared memory parallel computer," *IEEE Transactions on computers*, vol. 100, no. 2, pp. 175–189, 1983.

[40] P. E. Mckenney *et al.*, "Read-Copy Update," in *In Ottawa Linux Symposium*, 2001, pp. 338–367.

[41] P. E. McKenney, *What is RCU? – "Read, Copy, Update"*, [Online]. Available from: https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html.

[42] 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.

[43] M. Bouzeghoub, "A framework for analysis of data freshness," in *Proceedings of the 2004 international workshop on Information quality in information systems*, 2004, pp. 59–67.

[44] J. Cho and H. Garcia-Molina, "Synchronizing a database to improve freshness," *ACM sigmod record*, vol. 29, no. 2, pp. 117–128, 2000.

[45] A. Behrouzi-Far, E. Soljanin, and R. D. Yates, "Data Freshness in Leader-Based Replicated Storage," in *2020 IEEE International Symposium on Information Theory (ISIT)*, 2020, pp. 1806–1811.

[46] J. Zhong, R. Yates, and E. Soljanin, "Minimizing Content Staleness in Dynamo-Style Replicated Storage Systems," in *Infocom Workshop on Age of Information*, arXiv preprint arXiv:1804.00742, Apr. 2018.

[47] B. Adelberg, H. Garcia-Molina, and B. Kao, "Applying update streams in a soft real-time database system," *SIGMOD Rec.*, vol. 24, no. 2, pp. 245–256, May 1995.

[48] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, C. A. Soules, and A. Veitch, "LazyBase: trading freshness for performance in a scalable database," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12, Bern, Switzerland: Association for Computing Machinery, 2012, pp. 169–182, ISBN: 9781450312233.

[49] K.-D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher, "A QoS-sensitive approach for timeliness and freshness guarantees in real-time databases," in *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, IEEE, 2002, pp. 203–212.

[50] K.-D. Kang, S. H. Son, and J. A. Stankovic, "Managing deadline miss ratio and sensor data freshness in real-time databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 10, pp. 1200–1216, 2004.

[51] X. Song and J. Liu, "Performance of multiversion concurrency control algorithms in maintaining temporal consistency," in *Proceedings., Fourteenth Annual International Computer Software and Applications Conference*, 1990, pp. 132–139.

[52] R. Yates, "Lazy is Timely: Status Updates by an Energy Harvesting Source," in *Proc. IEEE Int'l. Symp. Info. Theory (ISIT)*, Jun. 2015, pp. 3008–3012.

[53] Y. Sun, E. Uysal-Biyikoglu, R. Yates, C. E. Koksal, and N. B. Shroff, "Update or wait: How to keep your data fresh," in *Proc. IEEE INFOCOM*, Apr. 2016.

[54] B. Zhou and W. Saad, "Joint Status Sampling and Updating for Minimizing Age of Information in the Internet of Things," *IEEE Transactions on Communications*, vol. 67, no. 11, pp. 7468–7482, 2019.

[55] B. Zhou and W. Saad, "Optimal Sampling and Updating for Minimizing Age of Information in the Internet of Things," in *2018 IEEE Global Communications Conference (GLOBECOM)*, 2018, pp. 1–6.

[56] A. M. Bedewy, Y. Sun, S. Kompella, and N. B. Shroff, "Optimal Sampling and Scheduling for Timely Status Updates in Multi-Source Networks," *IEEE Transactions on Information Theory*, vol. 67, no. 6, pp. 4019–4034, 2021.

[57] Y. Sun, Y. Polyanskiy, and E. Uysal, "Sampling of the Wiener Process for Remote Estimation Over a Channel With Random Delay," *IEEE Transactions on Information Theory*, vol. 66, no. 2, pp. 1118–1135, 2020.

[58] M. Bastopcu and S. Ulukus, "Age of Information for Updates With Distortion: Constant and Age-Dependent Distortion Constraints," *IEEE/ACM Trans. Netw.*, vol. 29, no. 6, pp. 2425–2438, Dec. 2021.

[59] X. Wu, J. Yang, and J. Wu, "Optimal Status Update for Age of Information Minimization With an Energy Harvesting Source," *IEEE Transactions on Green Communications and Networking*, vol. 2, no. 1, pp. 193–204, Mar. 2018.

[60] F. Chiariotti *et al.*, "Query age of information: Freshness in pull-based communication," *IEEE Transactions on Communications*, vol. 70, no. 3, pp. 1606–1622, 2022.

[61] B. Yin *et al.*, "Only those requested count: Proactive scheduling policies for minimizing effective age-of-information," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 109–117.

[62] 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.

[63] R. Yates, P. Ciblat, M. Wigger, and A. Yener, "Age-Optimal Constrained Cache Updating," in *Proc. IEEE Int'l. Symp. Info. Theory (ISIT)*, Jun. 2017, pp. 141–145.

[64] A. M. Bedewy, Y. Sun, and N. B. Shroff, "Age-optimal information updates in multihop networks," in *Proc. IEEE Int'l. Symp. Info. Theory (ISIT)*, Jun. 2017, pp. 576–580.

[65] R. Talak, S. Karaman, and E. Modiano, "Minimizing age-of-information in multi-hop wireless networks," in *55th Annual Allerton Conference on Communication, Control, and Computing*, Oct. 2017, pp. 486–493.

[66] C. Kam, S. Kompella, G. D. Nguyen, J. E. Wieselthier, and A. Ephremides, "Modeling the age of information in emulated ad hoc networks," in *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, Oct. 2017, pp. 436–441.

[67] F. Chiariotti, O. Vikhrova, B. Soret, and P. Popovski, "Peak Age of Information Distribution for Edge Computing With Wireless Links," *IEEE Transactions on Communications*, vol. 69, no. 5, pp. 3176–3191, 2021.

[68] A. Sinha, S. Singhvi, P. D. Mankar, and H. S. Dhillon, *Peak Age of Information under Tandem of Queues*, 2024. arXiv: 2405.02705 [cs.IT].

[69] O. Vikhrova, F. Chiariotti, B. Soret, G. Araniti, A. Molinaro, and P. Popovski, "Age of Information in Multi-hop Networks with Priorities," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–6.

[70] C. Kam, S. Kompella, G. D. Nguyen, and A. Ephremides, "Effect of Message Transmission Path Diversity on Status Age," *IEEE Trans. Info. Theory*, vol. 62, no. 3, pp. 1360–1374, Mar. 2016.

[71] R. Talak and E. H. Modiano, "Age-Delay Tradeoffs in Queueing Systems," *IEEE Transactions on Information Theory*, vol. 67, no. 3, pp. 1743–1758, 2021.

[72] M. Fidler, J. P. Champati, J. Widmer, and M. Noroozi, "Statistical Age-of-Information Bounds for Parallel Systems: When Do Independent Channels Make a Difference?" *IEEE Journal on Selected Areas in Information Theory*, vol. 4, pp. 591–606, 2023.

[73] R. D. Yates, "Status updates through networks of parallel servers," in *Proc. IEEE Int'l. Symp. Info. Theory (ISIT)*, Jun. 2018, pp. 2281–2285.

[74] J. M. George and J. M. Harrison, "Dynamic control of a queue with adjustable service rate," *Operations research*, vol. 49, no. 5, pp. 720–731, 2001.

[75] T. B. Crabill, "Optimal control of a maintenance system with variable service rates," *Operations Research*, vol. 22, no. 4, pp. 736–745, 1974.

[76] S. Stidham, "Optimal control of admission to a queueing system," *IEEE Transactions on Automatic Control*, vol. 30, no. 8, pp. 705–713, 1985.

[77] M. Hofri and K. W. Ross, "On the Optimal Control of Two Queues with Server Setup Times and Its Analysis," *SIAM Journal on Computing*, vol. 16, no. 2, pp. 399–420, 1987. eprint: https://doi.org/10.1137/0216029.

[78] N. Lee and V. G. Kulkarni, "Optimal arrival rate and service rate control of multi-server queues," *Queueing Systems*, vol. 76, pp. 37–50, 2014.

[79] R. R. Weber and S. Stidham, "Optimal control of service rates in networks of queues," *Advances in applied probability*, vol. 19, no. 1, pp. 202–218, 1987.

[80] Z. Rosberg, P. Varaiya, and J. Walrand, "Optimal control of service in tandem queues," *IEEE Transactions on Automatic Control*, vol. 27, no. 3, pp. 600–610, 1982.

[81] L. Xia, D. Miller, Z. Zhou, and N. Bambos, "Service rate control of tandem queues with power constraints," *IEEE Transactions on Automatic Control*, vol. 62, no. 10, pp. 5111–5123, 2017.

[82] 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.

[83] 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.

[84] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981.

[85] 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, Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 1403–1411.

[86] 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, San Francisco, CA, USA: Association for Computing Machinery, 2015, pp. 1–10, ISBN: 9781450332057.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] L.-t. implementation notes - the linux kernel documentation, *LC-trie implementation notes*.

[92] *Userspace RCU*, [Online]. Available from: https://liburcu.org/, 2021.

[93] *KnotDNS*, [Online]. Available from: https://www.knot-dns.cz/, 2021.

[94] *netsniff-ng*, [Online]. Available from: http://netsniff-ng.org/, 2021.

[95] *Sheepdog Project*, [Online]. Available from: https://sheepdog.github.io/sheepdog, 2015.

[96] C. Olston *et al.*, "TensorFlow-Serving: Flexible, High-Performance ML Serving," *CoRR*, vol. abs/1712.06139, 2017. arXiv: 1712.06139.

[97] M. Arbel and A. Morrison, "Predicate RCU: An RCU for Scalable Concurrent Updates," *SIGPLAN Not.*, vol. 50, no. 8, pp. 21–30, Jan. 2015.

[98] 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, Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 27–37, ISBN: 9781450362252.

[99] 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.

[100] 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.

[101] 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, Paris, France: Association for Computing Machinery, 2014, pp. 196–205, ISBN: 9781450329446.

[102] 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, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 779–792, ISBN: 9781450362405.

[103] 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.

[104] M. Kokologiannakis and K. Sagonas, "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, Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 172–181, ISBN: 9781450350778.

[105] 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.

[106] 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.

[107] 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, Thira, Santorini, Greece: Association for Computing Machinery, 2010, pp. 284–293, ISBN: 9781450300797.

[108] 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.

[109] 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, ISBN: 978-1-931971-10-2.

[110] 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.

[111] Y.-P. Hsu, "Age of Information: Whittle Index for Scheduling Stochastic Arrivals," in *2018 IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 2634–2638.

[112] Y.-P. Hsu, E. Modiano, and L. Duan, "Scheduling Algorithms for Minimizing Age of Information in Wireless Broadcast Networks with Random Arrivals," *IEEE Transactions on Mobile Computing*, vol. 19, no. 12, pp. 2903–2915, 2020.

[113] I. Kadota, A. Sinha, E. Uysal-Biyikoglu, R. Singh, and E. Modiano, "Scheduling policies for minimizing age of information in broadcast wireless networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2637–2650, 2018.

[114] A. Maatouk, S. Kriouile, M. Assad, and A. Ephremides, "On the Optimality of the Whittle's Index Policy for Minimizing the Age of Information," *IEEE Transactions on Wireless Communications*, vol. 20, no. 2, pp. 1263–1277, 2021.

[115] J. Sun, Z. Jiang, B. Krishnamachari, S. Zhou, and Z. Niu, "Closed-Form Whittle's Index-Enabled Random Access for Timely Status Update," *IEEE Transactions on Communications*, vol. 68, no. 3, pp. 1538–1551, 2020.

[116] I. Kadota, A. Sinha, and E. Modiano, "Scheduling Algorithms for Optimizing Age of Information in Wireless Networks With Throughput Constraints," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1359–1372, 2019.

[117] V. Tripathi and E. Modiano, "A whittle index approach to minimizing functions of age of information," in *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, IEEE, 2019, pp. 1160–1167.

[118] A. Maatouk, S. Kriouile, M. Assaad, and A. Ephremides, "Asymptotically optimal scheduling policy for minimizing the age of information," in *2020 IEEE International Symposium on Information Theory (ISIT)*, IEEE, 2020, pp. 1747–1752.

[119] J. Sun, Z. Jiang, S. Zhou, and Z. Niu, "Optimizing information freshness in broadcast network with unreliable links and random arrivals: An approximate index policy," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2019, pp. 115–120.

[120] Z. Jiang, B. Krishnamachari, S. Zhou, and Z. Niu, "Can decentralized status update achieve universally near-optimal age-of-information in wireless multiaccess channels?" In *2018 30th International Teletraffic Congress (ITC 30)*, IEEE, vol. 1, 2018, pp. 144–152.

[121] S. Ross, in *Introduction to Stochastic Dynamic Programming*, ser. Probability and Mathematical Statistics: A Series of Monographs and Textbooks, Academic Press, 1983, pp. 89–106.

[122] S. Ross, *Applied Probability Models with Optimization Applications* (Holden-Day series in industrial engineering and management science). Holden-Day, 1970, ISBN: 9780335041510.

[123] L. I. Sennott, "Average Cost Optimal Stationary Policies in Infinite State Markov Decision Processes with Unbounded Costs," *Operations Research*, vol. 37, no. 4, pp. 626–633, 1989.

[124] *Kafka 3.4 Documentation*, https://kafka.apache.org/documentation.html#design_pull, Accessed: 2023-05-04.

[125] D. M. Topkis, *Supermodularity and complementarity*. Princeton university press, 1998.

[126] R. G. Gallager, "Discrete stochastic processes," *OpenCourseWare: Massachusetts Institute of Technology*, 2011.

[127] R. G. Gallager, *Stochastic processes: theory for applications*. Cambridge University Press, 2013.

[128] 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.

[129] R. Yates and D. Goodman, *Probability and Stochastic Processes: A Friendly Introduction for Electrical and Computer Engineers* (Probability and Stochastic Processes: A Friendly Introduction for Electrical and Computer Engineers). Wiley, 2014, ISBN: 9781118324561.

[130] 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.

[131] 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.

[132] 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.

[133] D. P. D. Kit, [Online]. Available from: https://www.dpdk.org/, 2021.

[134] M. Costa, M. Codreanu, and A. Ephremides, "On the Age of Information in Status Update Systems With Packet Management," *IEEE Trans. Info. Theory*, vol. 62, no. 4, pp. 1897–1910, Apr. 2016.

[135] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable Low Latency for Data Center Applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, San Jose, California: Association for Computing Machinery, 2012, ISBN: 9781450317610.

[136] H. Shao, X. Wang, Y. Lu, Y. Yu, S. Zheng, and Y. Zhao, "Accessing Cloud with Disaggregated Software-Defined Router," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, Apr. 2021, pp. 1–14, ISBN: 978-1-939133-21-2.

[137] H. Guo and P. Crossley, "Design of a Time Synchronization System Based on GPS and IEEE 1588 for Transmission Substations," *IEEE Transactions on Power Delivery*, vol. 32, no. 4, pp. 2091–2100, 2017.

[138] P. Emmerich, S. Gallenmüller, G. Antichi, A. W. Moore, and G. Carle, "Mind the Gap - A Comparison of Software Packet Generators," in *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2017, pp. 191–203.

[139] R. Mur-Artal and J. D. Tardos, "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, Oct. 2017.

[140] S. Semenova, S. Y. Ko, Y. D. Liu, L. Ziarek, and K. Dantu, "A Quantitative Analysis of System Bottlenecks in Visual SLAM," in *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '22, Tempe, Arizona: Association for Computing Machinery, 2022, pp. 74–80, ISBN: 9781450392181.

[141] Ben Ali, Ali J. and Kouroshli, Marziye and Semenova, Sofiya and Hashemifar, Zakieh Sadat and Ko, Steven Y. and Dantu, Karthik, "Edge-SLAM: Edge-Assisted Visual Simultaneous Localization and Mapping," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 1, Oct. 2022.

[142] S. M. Ross, *Introduction to Probability Models*, Sixth. San Diego, CA, USA: Academic Press, 1997.

[143] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, 1992.

[144] N. Magen, A. Kolodny, U. Weiser, and N. Shamir, "Interconnect-power dissipation in a microprocessor," in *Proceedings of the 2004 International Workshop on System Level Interconnect Prediction*, ser. SLIP '04, Paris, France: Association for Computing Machinery, 2004, pp. 7–13, ISBN: 1581138180.

[145] T. Sakurai and A. Newton, "Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 2, pp. 584–594, 1990.

[146] R. Gonzalez, B. Gordon, and M. Horowitz, "Supply and threshold voltage scaling for low power CMOS," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 8, pp. 1210–1216, 1997.

[147] J. Gong, Q. Kuang, X. Chen, and X. Ma, "Reducing age-of-information for computation-intensive messages via packet replacement," in *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*, IEEE, 2019, pp. 1–6.

[148] R. D. Yates, "The Age of Gossip in Networks," in *2021 IEEE International Symposium on Information Theory (ISIT)*, 2021, pp. 2984–2989.

[149] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 361–378, ISBN: 978-1-931971-49-2.

[150] M. Bastopcu and S. Ulukus, *Who Should Google Scholar Update More Often?* 2020. arXiv: 2001.11500 [cs.IT].