SOAP: One Clean Analysis of All Age-Based Scheduling Policies

ZIV SCULLY, Carnegie Mellon University
MOR HARCHOL-BALTER, Carnegie Mellon University
ALAN SCHELLER-WOLF, Carnegie Mellon University

We consider an extremely broad class of M/G/1 scheduling policies called SOAP: Schedule Ordered by Age-based Priority. The SOAP policies include almost all scheduling policies in the literature as well as an infinite number of variants which have never been analyzed, or maybe not even conceived. SOAP policies range from classic policies, like first-come, first-serve (FCFS), foreground-background (FB), class-based priority, and shortest remaining processing time (SRPT); to much more complicated scheduling rules, such as the famously complex Gittins index policy and other policies in which a job's priority changes arbitrarily with its age. While the response time of policies in the former category is well understood, policies in the latter category have resisted response time analysis. We present a universal analysis of all SOAP policies, deriving the mean and Laplace-Stieltjes transform of response time.

CCS Concepts: • **General and reference** → **Performance**; • **Mathematics of computing** → **Queueing theory**; • **Software and its engineering** → **Scheduling**; • **Computing methodologies** → *Model development and analysis*; • **Theory of computation** → Scheduling algorithms;

Additional Key Words and Phrases: M/G/1; exact response time analysis; Gittins index; shortest expected remaining processing time (SERPT)

ACM Reference format:
Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. 2018. SOAP: One Clean Analysis of All Age-Based Scheduling Policies. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 1, Article 16 (March 2018), 30 pages.
https://doi.org/10.1145/3179419

---

1 INTRODUCTION

Analyzing the response time of scheduling policies in the M/G/1 setting has been the focus of thousands of papers over the past half century, from classic early works [10, 15–17, 21–23] to more recent works in the SIGMETRICS community [1, 2, 6–9, 18–20, 24–26]. Examples of common scheduling policies include

- *first-come, first-served* (FCFS), which serves jobs nonpreemptively in the order they arrive;
- *class-based priority*, which serves the job of highest priority class, possibly preemptively and possibly nonpreemptively;
- *shortest remaining processing time* (SRPT), which preemptively serves the job with the least remaining time;
- *foreground-background* (FB), which preemptively serves the job that has received the least service so far; and

---

© 2018 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proc. ACM Meas. Anal. Comput. Syst.*, https://doi.org/10.1145/3179419.

- *processor sharing* (PS), which concurrently serves all jobs in the system at the same rate.

In just these few examples we see a variety of features represented: preemptible jobs, nonpreemptible jobs, prioritizing by class, prioritizing by job size, and prioritizing by service received so far, or *age*. Each policy requires a custom response time analysis that takes into account its particular combination of features.

Although there has been much success in analyzing the response time of specific scheduling policies in the M/G/1 setting, such as those listed above, results are ad-hoc and *limited to relatively simple policies*. Analyzing variants of the above simple policies, let alone fundamentally different policies, is an open problem. For instance, none of the following scenarios have been analyzed before.

- Suppose we have exact size information for some “sized” jobs but not other “unsized” jobs. We run SRPT on sized jobs and FB on unsized jobs, meaning that we serve the sized job of minimum remaining time or unsized job of minimum age, whichever measurement is smaller.
- Suppose we have jobs that are neither fully preemptible nor fully nonpreemptible but instead preemptible only at specific “checkpoint” ages. We run a preemptive policy, such as SRPT or FB, but only preempt jobs when they reach checkpoint ages.
- The *Gittins index policy* [3, 11], long known to be optimal for minimizing mean response time in the M/G/1 queue[1], has only been analyzed in certain special cases [13, 20]. In general, the Gittins index policy can have a complex priority scheme [4] which, while known to perform optimally, has not been analyzed before in its general form.

Approaching the above examples with state-of-the-art techniques, if possible at all, would require an ad-hoc analysis for each scenario. We seek *general principles and techniques* for response time analysis that apply to not just the above examples but to as many scheduling policies as possible, even those not yet imagined.

1.1 Contributions

We introduce *SOAP*, a *universal framework* for defining and analyzing M/G/1 scheduling policies. The SOAP framework can analyze *any SOAP scheduling policy*, which includes nearly any policy where a job’s priority depends on its own characteristics: class, size, age, and so on. Specifically, we make the following contributions.

- We *define* the class of SOAP policies (Section 2), a broad class of policies that includes the three unsolved examples above as well as many other policies, from practical scenarios to policies not yet imagined. We *encode* many policies old and new as SOAP policies (Section 3).
- We give a *universal response time analysis* that works for any SOAP policy (Section 5), obtaining closed forms for the mean (Theorem 5.5) and Laplace-Stieltjes transform (Theorem 5.4). In particular, we apply our results to *previously intractable analyses* (Section 6), such as the response time of the Gittins index policy.

In defining and analyzing SOAP policies, there are two major technical challenges. The first major challenge is that to have a single analysis apply to many scheduling policies at once, we need to *express all such policies within a single framework*. The SOAP framework encodes a scheduling policy as a *rank function*, which maps each job to a priority level, or *rank*. All SOAP policies are based on a single rule: always serve the job of *minimal rank*. For example, in a preemptive class-based priority system, a job’s rank is its class (Example 3.3), whereas in SRPT, a job’s rank is its remaining time (Example 3.4). Rank functions can express a huge variety of policies, from virtually all classic

---

[1] While SRPT is optimal when exact job sizes are known, the Gittins index policy, of which SRPT is a special case, is optimal even when only size distributions are known.

policies (Section 3.1) to complex policies which have never been analyzed before (Section 3.2). A notable exception is PS, which does not fit into the SOAP framework.

The second major challenge is to *analyze policies with arbitrary rank functions*. In particular, nearly all previously analyzed scheduling policies, when expressed as SOAP policies, have rank functions that are *monotonic* in age. For example, under SRPT, a job's rank decreases with age, making it less and less likely to be preempted by another job, while under FB, a job's rank increases with age, making it more and more likely to be preempted by another job. Unfortunately, the techniques used in the past to analyze policies with monotonic rank functions *break down for arbitrary nonmonotonic rank functions*, which appear, for instance, when studying the Gittins index policy (Example 3.6) and jobs that are preemptible only at certain checkpoints (Example 3.7). We develop *new analytical tools* that work for arbitrary rank functions (Section 4).

1.2 Related Work

Our work on SOAP policies follows in the tradition of analyses that address an entire class of policies at once. Two such classes are *SMART* [25] and *multilevel processor sharing* (MLPS) [17].

- The SMART class includes all policies that satisfy certain criteria that ensure they prioritize small jobs over large ones, such as SRPT and PSJF (Example 3.4). Some recent work on SMART policies includes analyzing the tail behavior of response time [19] and characterizing the tradeoff between accuracy of size estimates and response time [26].
- The MLPS class consists of policies that divide all jobs in the system into echelons based on age, then serves jobs in the youngest echelon according to FCFS, FB, or PS. Some recent work on MLPS policies includes optimally choosing the age echelon cutoffs [5] and connecting MLPS to the Gittins index policy [4].

While the SMART and MLPS classes have nearly no overlap, the SOAP class includes many policies from *both* classes. Specifically, the SMART* subclass of SMART [25] and MLPS policies which do not use PS are all SOAP policies.

A particularly important SOAP policy is the Gittins index policy [3, 11], which minimizes mean response time in the M/G/1 queue when job sizes are not known. The Gittins index policy has a rather complex definition, but recent work [3, 4] has revealed some of its structural properties. Osipova et al. [20] analyze a specific case of the Gittins index policy for a multiclass M/G/1 queue where each class's job size distribution has the *decreasing hazard rate* (DHR) property. Using the SOAP framework, we can analyze the Gittins index policy for *arbitrary size distributions*. Hyytiä et al. [13] show that for jobs with known sizes, a weighted version of the Gittins index yields the *shortest processing time product* (SPTP) policy and that this policy minimizes mean *slowdown*, which is the ratio of a job's response time to its size. SPTP is a SOAP policy, so the SOAP framework can obtain the Laplace-Stieltjes transform of slowdown for SPTP, extending the previous mean analysis.

2 SYSTEM MODEL AND SOAP POLICIES

We consider work-conserving scheduling policies for the M/G/1 queue. We write λ for the total arrival rate and X for the overall job size distribution. We assume a stable system, meaning $\lambda\mathbf{E}[X] < 1$, and a preempt-resume model, meaning preemption and processor sharing are permitted without penalty or loss of work.

2.1 Descriptors

Scheduling algorithms use information about jobs in the system when deciding which job to serve. We can divide this information into two types: *static* and *dynamic*.

- *Static* information about a job is revealed when it enters the system and never changes. For example, in a system with multiple job classes, a job's class would be static information, and in a system where exact job sizes are known, a job's exact size would be static information. We call a job's static information its *descriptor* and write $\mathcal{D}$ for the set of descriptors. A job's descriptor d determines its size distribution $X_d = (X \mid \text{job has descriptor } d)$.
- *Dynamic* information about a job changes as a job is served. In this paper, the only dynamic information about a job is its *age*, the amount of time it has been served. The set of possible ages is $\mathbb{R}_{\geq 0}$.

Descriptors are often tuples. To distinguish descriptors from ranks, another type of tuple introduced in Section 2.2, we write descriptors in [square brackets] and ranks in ⟨angle brackets⟩.

Example 2.1. Consider a system with a set of job classes $\mathcal{K}$, where X_k is the size distribution of class $k \in \mathcal{K}$. Depending on what information is known to the scheduler, the set of descriptors $\mathcal{D}$ may be one of several options.

- If jobs do not reveal their exact size upon entering the system, then $\mathcal{D} = \mathcal{K}$, because the only static information we have about each job is its class. The size distribution of jobs with descriptor k is simply X_k.
- If jobs reveal their exact size upon entering the system, then $\mathcal{D} = \mathcal{K} \times \mathbb{R}_{\geq 0}$, because we know each job's class and size. The size distribution of jobs with descriptor $[k, x]$ is $X_{[k,x]} = x$, the deterministic distribution with value x.
- If only some jobs reveal their exact size, then

$$\mathcal{D} = \mathcal{K} \times (\mathbb{R}_{\geq 0} \cup \{?\}),$$

because some jobs have known exact size $x \in \mathbb{R}_{\geq 0}$ while others have unknown size, which we denote by ?. The size distributions are $X_{[k,x]} = x$ for $x \in \mathbb{R}_{\geq 0}$ and $X_{[k,?]} = X_k$.

We require that the descriptors of jobs must be chosen i.i.d. according to a fixed distribution. For instance, in Example 2.1, each job's class must be chosen i.i.d., and in the third scenario, having each job independently reveal its size with probability 1/2 is permitted, but having alternating arrivals reveal their sizes is not.

2.2 SOAP Policies and Rank Functions

A *SOAP scheduling policy* is a preemptive priority policy where a job's descriptor and age determine its priority. SOAP is an acronym for *Schedule Ordered by Age-based Priority.* Specifically, a SOAP policy is specified by the following ingredients:

- a set $\mathcal{R}$ of *ranks*,
- a strict total order $\prec$ on $\mathcal{R}$, and
- a *rank function* assigning a rank $r(d, a)$ to each pair of descriptor d and age a,

$$r : \mathcal{D} \times \mathbb{R}_{\geq 0} \to \mathcal{R}.$$

The defining property of SOAP policies is the following.

Every moment in time, a SOAP policy serves the job of *minimum rank.*

Ties between jobs of the same rank are broken using a tiebreaking rule. For simplicity of exposition, we focus on *first-come, first-served* (FCFS) tiebreaking, which, among the jobs of minimal rank, serves the job that arrived to the system first. Our results also apply to SOAP policies that use *last-come, first-served* (LCFS) tiebreaking, which we defer to Appendix A[2].

---

[2] Sometimes ties for minimum rank lead to processor sharing, and these ties do not require tiebreaking. See Example 3.1 and Appendix B for details.

Suppose job J has descriptor d_J and age a_J and job K has descriptor d_K and age a_K. We say J *outranks* K if $r(d_J, a_J) \prec r(d_K, a_K)$ or both $r(d_J, a_J) = r(d_K, a_K)$ and J arrived before K. SOAP policies always serve the job that outranks all other jobs in the system.

A great many SOAP policies can be expressed using $\mathcal{R} = \mathbb{R}$, in which case $\prec$ is the usual ordering $<$ on $\mathbb{R}$. However, it is often convenient to have a nested rank structure in which jobs are first prioritized by a *primary rank*, then by a *secondary rank*, and only then by the tiebreaking rule. We express such nested ranks using $\mathcal{R} = \mathbb{R}^2$. Each rank is a pair $\langle r_1, r_2 \rangle$ of primary rank r_1 and secondary rank r_2, and $\prec$ is the lexicographic ordering: $\langle r_1, r_2 \rangle \prec \langle r_1', r_2' \rangle$ if $r_1 < r_1'$ or both $r_1 = r_1'$ and $r_2 < r_2'$. We write primary and secondary ranks of a state (d, a) as $r_1(d, a)$ and $r_2(d, a)$, respectively, so

$$r(d, a) = \langle r_1(d, a), r_2(d, a) \rangle.$$

When specifying a SOAP policy, we usually leave the choice of $\mathcal{R}$ unstated, as it is implied from the formula for the rank function. Our results apply to $\mathcal{R} = \mathbb{R}^n$ ordered lexicographically for any $n \geq 1$, and they easily generalize to other choices of $\mathcal{R}$.

We will devote much time to discussing how jobs' ranks change with age. Thus, when we call a rank function "monotonic" or similar, we mean that it is so with respect to age, not descriptor.

For a SOAP policy to be well-defined, its rank function must satisfy some technical conditions, which are given in Appendix B.

3 SOAP POLICIES ARE EVERYWHERE

3.1 Previously Analyzed SOAP Policies

Example 3.1. The *foreground-background* (FB) policy is a SOAP policy. It uses no static information, so $\mathcal{D} = \{\varnothing\}$, where $\varnothing$ is a "placeholder" descriptor assigned to every job. FB always serves the job of least age, so it has rank function $r(\varnothing, a) = a$. It is likely that many jobs are tied for minimum rank under FB, but whichever job is served immediately loses minimum status, resulting in a processor-sharing effect.

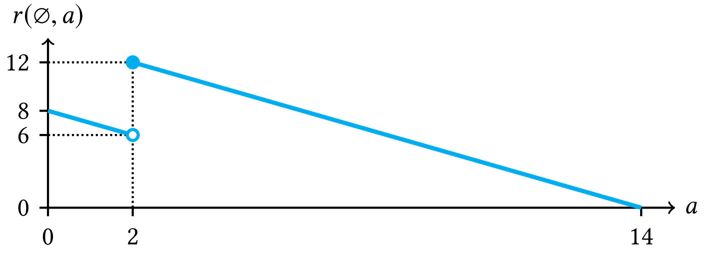
There are always many rank functions that encode the same SOAP policy. For instance, any rank function monotonically increasing in age, such as $r(\varnothing, a) = a^2$, also describes FB.

Example 3.2. The *first-come, first-served* (FCFS) policy is a SOAP policy. It uses no static information, so $\mathcal{D} = \{\varnothing\}$. FCFS is nonpreemptive, which is equivalent to always serving the job of maximal age, so it has rank function $r(\varnothing, a) = -a$. FCFS tiebreaking plays a crucial role by breaking ties between jobs of age 0.

Once again, there are multiple rank functions that describe FCFS. In particular, a constant rank function yields FCFS due to the tiebreaking rule, but we prefer the given encoding because it makes it clear that FCFS is a *nonpreemptive* policy. As the following examples demonstrate, using primary rank $-a$ is a general way to indicate nonpreemptiveness in a rank function.

Example 3.3. Consider a system with classes $\mathcal{K} = \{1, \ldots, n\}$ where jobs within each class are served in FCFS order but class 1 has highest priority, class 2 has next-highest priority, and so on. The *nonpreemptive priority* and *preemptive priority* policies are SOAP policies. Both policies use job class as static information, so $\mathcal{D} = \mathcal{K}$.

- Nonpreemptive priority has rank function $r(k, a) = \langle -a, k \rangle$: the primary rank prevents preemption, and the secondary rank prioritizes the classes when starting a new job.
- Preemptive priority has rank function $r(k, a) = \langle k, -a \rangle$: because k is the primary rank, jobs from high-priority classes preempt those in low priority classes.



The rank function for SERPT using the distribution described in Example 3.5: jobs have size either 2 or 14, each with probability 1/2. The rank is the expected remaining size of a job given it has reached its age a. In this case, the initial expected size is 8, but if the job does not finish at age 2, then we know it must be size 14, so its expected remaining size jumps up to 12.

Fig. 3.1. Rank Function for SERPT (Example 3.5)

Example 3.4. The *shortest job first* (SJF), *preemptive shortest job first* (PSJF), and *shortest remaining processing time* (SRPT) policies are SOAP policies. All three policies assume exact size information is known and use it when scheduling, so all use $\mathcal{D} = \mathbb{R}_{\geq 0}$.

- SJF has rank function $r(x, a) = \langle -a, x \rangle$: it is a *nonpreemptive* priority policy with size as priority.
- PSJF has rank function $r(x, a) = \langle x, -a \rangle$: it is a *preemptive* priority policy with size as priority.
- SRPT has rank function $r(x, a) = x - a$: a job's rank is its remaining size.

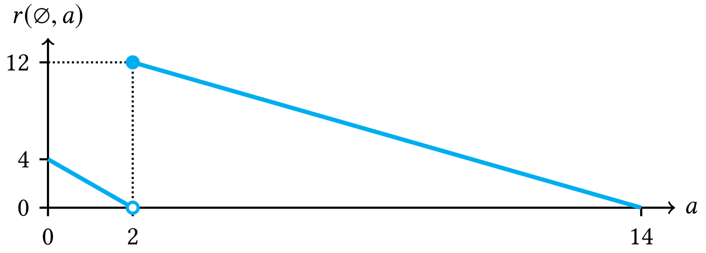
3.2 Newly Analyzed SOAP Policies

Example 3.5. The *shortest expected processing time* (SEPT), *preemptive shortest expected processing time* (PSEPT) and *shortest expected remaining processing time* (SERPT) policies are SOAP policies. The policies are respective analogues of SJF, PSJF, and SRPT, but they do not have access to exact size information.

- SEPT has rank function $r(d, a) = \langle -a, \mathbf{E}[X_d] \rangle$: it is a *nonpreemptive* priority policy with expected size as priority.
- PSEPT has rank function $r(d, a) = \langle \mathbf{E}[X_d], -a \rangle$: it is a *preemptive* priority policy with expected size as priority.
- SERPT has rank function $r(d, a) = \mathbf{E}[X_d - a \mid X_d > a]$: a job's rank is its expected remaining size.

While SEPT and PSEPT have analyses similar to those of SJF and PSJF, respectively, SERPT has never been analyzed before in full generality. We have left the set of descriptors $\mathcal{D}$ unspecified because the definitions above work for any set of descriptors.

For concreteness, consider a system where all jobs have the same descriptor $\varnothing$ and size either 2 or 14, each with probability 1/2. The resulting rank function for SERPT, shown in Figure 3.1, is *nonmonotonic* with respect to age. This differs from *every policy described in Section 3.1*, all of which have monotonic rank functions. The potential nonmonotonicity of SERPT's rank function has prevented previous techniques from analyzing SERPT in full generality. We give the first response time analysis of SERPT using our general analysis of all SOAP policies (Section 6.4).



The rank function for the Gittins Index Policy using the same distribution as in Fig. 3.1: jobs have size either 2 or 14, each with probability 1/2. Compared to SERPT, the Gittins index policy gives more priority to jobs before they reach age 2. For instance, while SERPT ranks a job with age 1.99 on par with a hypothetical job that deterministically has remaining size 6.01, the Gittins index policy ranks such a job on par with a hypothetical job that deterministically has remaining size 0.02. This reflects the fact that it is almost free to run such a job to age 2, just in case it is about to finish. We show in Section 6.4 that the Gittins index policy achieves lower mean response time than SERPT due to its prioritizing potentially short jobs.

Fig. 3.2. Rank Function for Gittins Index (Example 3.6)

Example 3.6. The *Gittins index* of a job with descriptor d and age a is [3, 11]

$$G(d, a) = \sup_{\Delta > 0} \frac{\mathbf{P}\{X_d - a \le \Delta \mid X_d > a\}}{\mathbf{E}[\min\{X_d - a, \Delta\} \mid X_d > a]} = \sup_{\Delta > 0} \frac{\int_a^{a+\Delta} f_d(t)\, dt}{\int_a^{a+\Delta} \bar{F}_d(t)\, dt},$$

where f_d and $\bar{F}_d$ are the density and tail functions of X_d, respectively. The *Gittins index policy* is the scheduling policy that always serves the job of maximal Gittins index, and it is known to minimize mean response time in the M/G/1 queue [11]. Although optimality of the Gittins index policy has long been known, only a few special cases have been analyzed in the past [13, 20]. The Gittins index policy is a SOAP policy with rank function

$$r(d, a) = \frac{1}{G(d, a)}.$$

Like the policies in Example 3.5, the Gittins index policy can be defined with any set of descriptors.

The Gittins index policy is in general not the same as SERPT, as shown in Figure 3.2, but, like SERPT, the Gittins index policy often uses a nonmonotonic rank function, making it impossible to analyze in general using previous techniques. We give the first response time analysis of the Gittins index policy using our general analysis of all SOAP policies (Section 6.4).

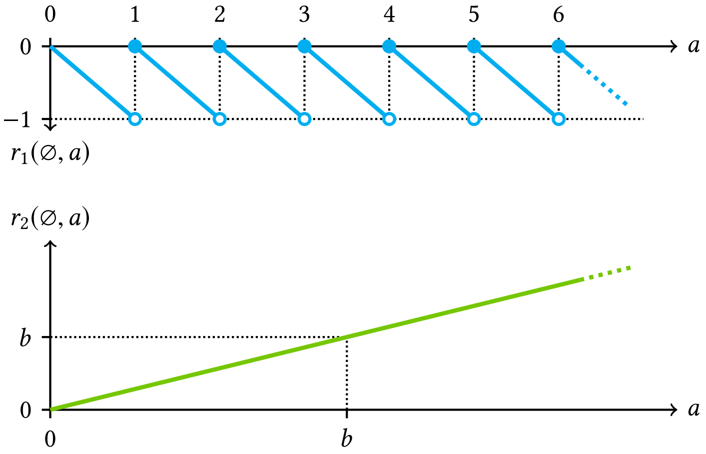
Example 3.7. Consider a system in which jobs, rather than being completely nonpreemptible or preemptible, are *preemptible at specific checkpoints*, say every 1 time unit. The *discretized FB* policy is a variant of FB for jobs with checkpoints: when possible, it serves the job of minimal age, but it does not preempt jobs between checkpoints[3]. Discretized FB is a SOAP policy. It uses no static information, so $\mathcal{D} = \{\varnothing\}$, and it has rank function

$$r(\varnothing, a) = \langle \lfloor a \rfloor - a, a \rangle,$$

This rank function is illustrated in Figure 3.3. Roughly speaking, the primary rank encodes the "discretized" aspect, preempting a job only at integer ages a when $\lfloor a \rfloor - a = 0$, and the secondary rank encodes the "FB" aspect.

---

[3] We note that it is possible to model discretized FB as an MLPS policy with infinitely many thresholds.



The rank function for the discretized FB policy described in Example 3.7. The primary rank (top) ensures that jobs are only preempted at integer ages. This is because new jobs enter the system with primary rank 0, and jobs only have primary rank 0 at integer ages. The secondary rank (bottom) prioritizes the job of lowest age, as in traditional FB.

Fig. 3.3. Rank Function for Discretized FB (Example 3.7)

Discretized FB is just one example of a policy for jobs preemptible only at specific checkpoints, but we can "discretize" any other SOAP policy by using primary rank $\lfloor a \rfloor - a$. For instance, *discretized SRPT* has rank function $r(x, a) = \langle \lfloor a \rfloor - a, x - a \rangle$.

We have seen a variety of features that SOAP policies can model:

- jobs that are nonpreemptible, preemptible, or preemptible at checkpoints;
- jobs with known or unknown exact size;
- priority based on a job's exact size or expected size;
- class-based priority in multiclass systems; and
- priority that changes nonmonotonically as a job ages.

As the following examples show, SOAP policies go even further: they can *combine many such features* as part of a single policy.

Example 3.8. Consider a system with two customer classes, humans (H) and robots (R).

- Humans, unpredictable and easily offended, have unknown service time, are nonpreemptible, and are served according to FCFS relative to other humans.
- Robots, precise and ruthlessly efficient, have known service time, are preemptible, and are served according to SRPT relative to other robots.

We can model the system using $\mathcal{D} = \{[H, ?]\} \cup \{[R, x] \mid x \in \mathbb{R}_{\geq 0}\}$, where ? denotes unknown size. A reasonable policy might have humans outrank most robots but let short robots, say those with remaining size less than some threshold x_H, outrank humans that have not yet started service. This results in rank function

$$r([H, ?], a) = \langle -a, x_H \rangle$$
$$r([R, x], a) = \langle 0, x - a \rangle,$$

which uses primary rank to encode preemptibility and secondary rank to encode priority. We analyze this system in Section 6.2.

Example 3.9. Consider a system as in Example 3.3 with $n = 3$ classes. Suppose that, in addition to class-based priority,

- jobs in class 1 are preemptible, have known size, and are served according to SRPT;
- jobs in class 2 are nonpreemptible, have known size, and are served according to SJF; and
- jobs in class 3 are preemptible at specific checkpoints, have unknown size, and are served according to discretized FB, as in Example 3.7.

The static information of a job is its class and, if known, its size, so $\mathcal{D} = \{1, 2\} \times \mathbb{R}_{\geq 0} \cup \{[3, ?]\}$, where ? denotes unknown size. The rank function, which uses $\mathcal{R} = \mathbb{R}^3$ as the set of ranks, is

$$\begin{aligned} r([1, x], a) &= \langle 0, 1, x - a \rangle \\ r([2, x], a) &= \langle -a, 2, x \rangle \\ r([3, ?], a) &= \langle \lfloor a \rfloor - a, 3, a \rangle. \end{aligned}$$

The components of the rank respectively encode preemptibility, class-based priority, and the policy used within each class.

3.3 SOAP Policies with LCFS Tiebreaking

There are two common SOAP policies that use LCFS tiebreaking instead of FCFS. The *last-come, first-serve* (LCFS) policy, which has the same rank function as FCFS in Example 3.2 but uses LCFS tiebreaking. The rank function still ensures nonpreemption, but now ties between jobs of age 0 are broken by LCFS. Similarly, the *preemptive last-come, first-serve* (PLCFS) policy is a SOAP policy with constant rank function and LCFS tiebreaking. SOAP policies that use LCFS tiebreaking admit essentially the same analysis as those that use FCFS tiebreaking (Appendix A).

3.4 Non-SOAP Policies

As our examples have demonstrated, there is an extremely wide variety of SOAP policies. However, there are some policies which are not SOAP policies, many of which fit into three broad categories.

First, some policies *cannot be expressed using descriptors that are distributed i.i.d. for each arriving job.* For example, the *earliest deadline first* (EDF) policy could be a SOAP policy if each job's descriptor were its deadline, but deadlines cannot be i.i.d. because later arrivals need later deadlines.

Second, some policies *require a tiebreaking rule other than FCFS or LCFS.* For example, the *random order of service* (ROS) policy could be a SOAP policy using the rank function in Example 3.2 if it could break ties between jobs of age 0 differently. A future generalization of the SOAP class might allow for ROS tiebreaking because, like FCFS and LCFS, it serves one job at a time. In contrast, the *processor sharing* (PS) policy, which is also not a SOAP policy, requires a fundamentally different tiebreaking rule.

Third, some policies have *job priorities that are context-dependent.* For example, a nonpreemptive policy in a multiclass system that tries to alternate between serving jobs of class 1 and class 2 is not a SOAP policy, because the priority of a job depends on external context, namely the class of the previously served job. The rank function approach used by SOAP policies inherently considers each job individually, so there is no way to capture such context.

4 HOW TO HANDLE ANY RANK FUNCTION

We have seen how to express a vast space of policies in the SOAP framework by careful choice of rank function. However, as demonstrated by Section 3.2, the rank function that encodes a SOAP policy can be very complicated. This leaves us with a difficult technical challenge: how do we analyze SOAP policies with arbitrary rank functions?

Before tackling arbitrary rank functions, let us recall how classic response time analyses work. Though there are of course many approaches, all of the policies in Section 3.1 can be analyzed with the "tagged job" approach, which follows a particular job through the system to analyze its response time. For instance, consider tagging a job of size x in a system using PSJF (Example 3.4). There are two types of jobs that outrank the tagged job:

- jobs of size at most x that are present in the system when the tagged job arrives and
- jobs of size less than x that arrive at the system after the tagged job arrives but before it completes.

One way to think about PSJF is to view the tagged job as seeing the system through "transformer glasses" [12] which transform the system by *hiding jobs that the tagged job outranks*. For PSJF, this transformation is simple because each job's rank is essentially constant[4]. A similar approach still works for policies with increasing or decreasing rank functions, but the hiding transformation becomes more complicated. For instance, under SRPT (Example 3.4), which has a decreasing rank function, a tagged job of size x sees a system transformed as follows.

- Jobs that arrive after the tagged job are hidden if their size is at least $x - a$, where a is the tagged job's age when the other job arrives.
- Jobs that arrive before the tagged job are hidden if their *remaining* size is greater than x. It may be that a job of *initial* size greater than x remains visible.

In more general terms: because jobs' ranks change with age, the tagged job's hiding criterion changes with its age, and whether or not other jobs satisfy that criterion changes with their ages. Handling these changes in rank is already tricky for SRPT, where a job's rank only decreases with age. The situation becomes even more complex when working with *nonmonotonic* rank functions.

4.1 Conventions

For the remainder of this section, we examine the journey of a tagged job J through the system. The tagged job J has descriptor d_J and size x_J. Note that we may use J's size as part of our analysis even if the scheduler does not have access to exact job sizes.

Throughout, we call jobs other than J *old* if they arrive before J and *new* if they arrive after J. As a mnemonic, we name old jobs I and name new jobs K, using subscripts when there are multiple such jobs. In examples, old jobs have descriptor d_I and new jobs have descriptor d_K, though in a real system it may of course happen that different old or new jobs have different descriptors. When it is unspecified whether a job is new or old, we name the job L.

4.2 Nonmonotonicity Difficulties

There are two major obstacles to analyzing policies with arbitrary nonmonotonic rank functions that do not occur when analyzing SRPT, which has a monotonic rank function. We illustrate these obstacles below and in Figure 4.1. The first obstacle concerns the nonmonotonicity of J's rank.

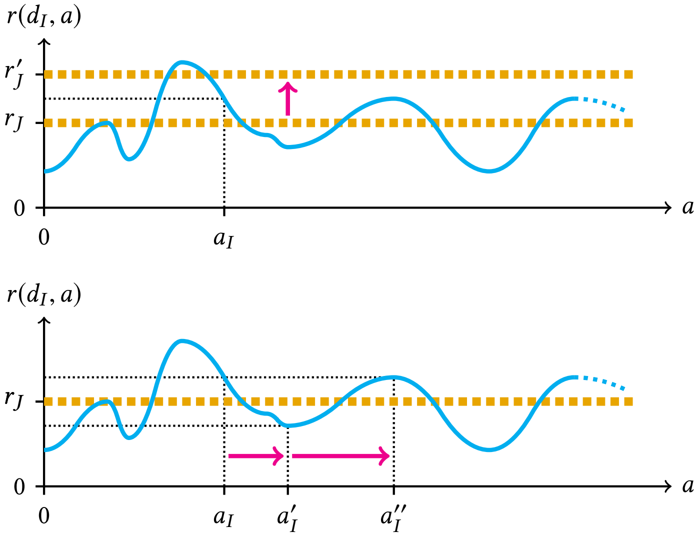
- In SRPT, we *permanently* hide some other jobs based on J's current rank.
- In general, another job L might be only *temporarily* hidden. If J's rank starts below L's rank but later exceeds it, the initially hidden L becomes visible again.

The second obstacle concerns the nonmonotonicity of the ranks of old jobs.

- In SRPT, an old job *permanently* outranks J if served for long enough before J arrives.
- In general, an old job I might only *temporarily* outrank J. Furthermore, if I's rank oscillates above and below J's initial rank, whether I gets hidden or stays visible depends on *when during I's service J arrives.*

---

[4] Due to FCFS tiebreaking, we could encode PSJF without its secondary rank.



We show two obstacles to the "transformer glasses" approach by examining the interaction between the tagged job J and an old job I of age a_I. The cyan curve shows I's rank as a function of its age. First (top), suppose a tagged job J has rank r_J upon entering the system, and suppose that old job I of age a_I is already in the system. Consider how J views I if, as pictured, $r_J \prec r(d_I, a_I)$. Then J outranks I, so I is initially hidden. However, I may be only *temporarily* hidden, because J's rank may later increase to $r'_J \succ r(d_I, a_I)$. Second (bottom), consider the same jobs J and I and suppose J has not entered the system yet. Whether or not I will be hidden from J depends on *I's age when J arrives*. For instance, as I advances in age from a_I to a'_I and later a''_I, it switches back and forth between being visible to and hidden from J.

Fig. 4.1. Difficulties with Nonmonotonic Rank Functions

Dealing with such arbitrarily varying rank functions appears intractable. We need *two key insights* in order to handle nonmonotonic rank functions: the *Pessimism Principle* (Section 4.3) and the *Vacation Transformation* (Section 4.4).

4.3 The Pessimism Principle

We call the amount of time any job L is served before J completes *J's delay due to L*. The response time of J is its size x_J plus its delays due to all jobs that are in the system with J at some point.

Suppose that a new job K arrives when J has age a_J. To analyze J's response time, we have to know its delay due to K. As we saw in Section 4.2, deciding whether or not to hide K based only on J's current rank $r(d_J, a_J)$ will not work. Instead, we need to examine J's *current and future ranks*.

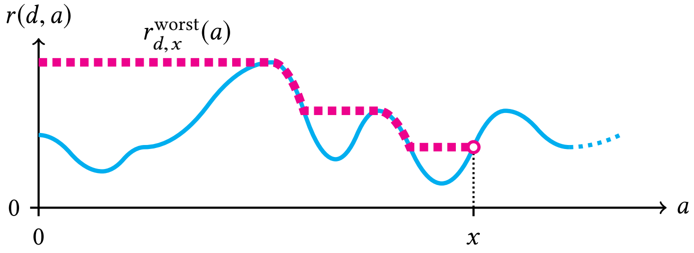
Definition 4.1. The *worst future rank* of a job with descriptor d, size x, and age a is

$$r^{\text{worst}}_{d,x}(a) = \sup_{a \le b < x} r(d, b).$$

Note that the worst future rank only considers ages up to the job's size x. See Figure 4.2 for an illustration.

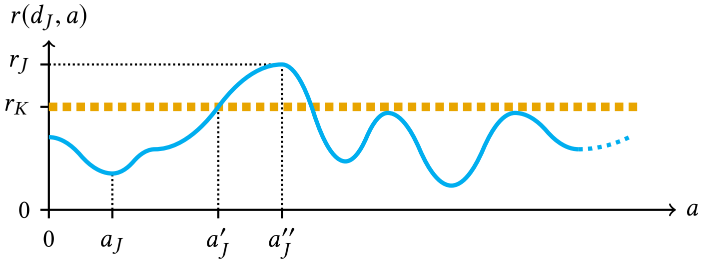
The reason we care about J's worst future rank is that for the purposes of computing J's delay delay due to K, we can essentially pretend that J has its worst future rank

$$r_J = r^{\text{worst}}_{d_J,x_J}(a_J),$$



The relationship between rank $r(d, a)$ (solid cyan), and worst future rank $r^{\text{worst}}_{d,x}(a)$ (dashed magenta) for a job with descriptor d and size x.

Fig. 4.2. Illustration of Worst Future Rank (Definition 4.1)



A new job K arrives when J, whose rank as a function of age is shown in cyan, has age a_J. Suppose for simplicity that K's rank is constant at r_K. An instant after J reaches age a'_J, it will be outranked by K, so K will complete before J reaches age a''_J. In practice, K's rank may change, but K will still complete before J reaches age a''_J unless K surpasses rank r_J, in which case K never outranks J again. Thus, for the purposes of finding J's delay due to K, we can pretend that J's rank is r_J for all of J's ages before a''_J.

Fig. 4.3. A Tagged Job's Delay due to a New Job

as illustrated in Figure 4.3. This is because before J reaches its worst future rank, K must either complete or surpass[5] rank r_J.

We have so far focused on a new job K, but the story is very similar for old jobs. The result is the Pessimism Principle, so named for its pessimistic focus on the worst future rank.

Pessimism Principle. The tagged job J's delay due to any other job L is the amount of time L is served until it either *completes* or *surpasses J's worst future rank.* To be precise, letting

$$r_J(a) = r^{\text{worst}}_{d_J,x_J}(a),$$

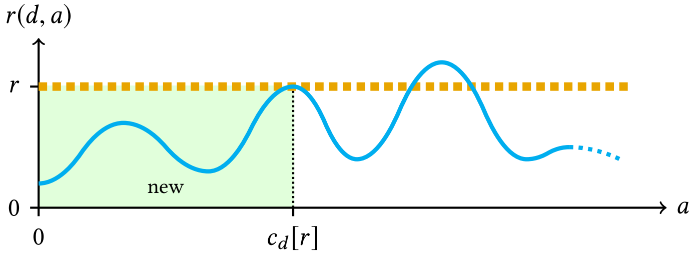
this means the following.

- Each old job I is served until it completes or first has rank $r_I > r_J(0)$. In particular, if $r_I = r_J(0)$, then I outranks J due to FCFS tiebreaking, thus the strict inequality[6].
- Each new job K is served until it completes or first has rank $r_K \geq r_J(a_J)$, where a_J is the age of J when K arrives.

---

[5] Unless otherwise noted, we mean "surpass" in a weak sense. That is, to surpass a rank r means to attain rank at least r.

[6] See Appendix C for discussion of corner cases where we need $\geq$ instead of $>$.



The new r-work for a job with descriptor d, whose rank as a function of age is shown in cyan, is the amount of service the job requires until it either completes or surpasses rank r at the cutoff age $c_d[r]$. Pictorially, the new r-work is the amount of service the job requires while its age is in the green region. The new r-work is at most $c_d[r]$ but may be less if the job completes. Note that the new r-work is not impacted by the job's rank at ages $a > c_d[r]$, even if $r(d, a) \prec r$.

Fig. 4.4. Illustration of New Work (Definition 4.2)

Furthermore, in both cases, the service occurs *before J is served while at its worst future rank.*

To clarify, the discussion above addresses the *total amount of service another job receives* while J is in the system. The service need not be contiguous but might be interleaved with that of J and other jobs.

The Pessimism Principle gives an implicit description of J's delay due to any other job L, but it remains to explicitly find this delay's distribution. We do this now for the case where L is new, treating the case where L is old in Section 4.4.

Definition 4.2. Let r be a rank. The *new r-work* is a random variable, written $X^{\text{new}}[r]$, representing how long a job that just arrived to the system is served until it completes or surpasses rank r. Specifically, we define $X^{\text{new}}[r] = X_D^{\text{new}}[r]$, where D is the random descriptor assigned to a new job and, for any specific descriptor d,

$$c_d[r] = \inf\{a \geq 0 \mid r(d, a) \succeq r\}$$
$$X_d^{\text{new}}[r] = \min\{X_d, c_d[r]\}.$$

That is, $c_d[r]$ is the *cutoff age* at which a new job with descriptor d surpasses rank r. See Figure 4.4 for an illustration.

Together, the Pessimism Principle and Definition 4.2 say that if a new job K has random descriptor and arrives when J has age a_J, then J's delay due to K is $X^{\text{new}}[r_{d_J, x_J}^{\text{worst}}(a_J)]$.

Example 4.3. Consider SRPT as described in Example 3.4, in which a job's descriptor is its size and its rank is its remaining size. Suppose that a new job K of size x_K arrives when J has age a_J. Under SRPT, J's worst future rank is its current rank $r(x_J, a_J) = x_J - a_J$, so the cutoff age for K is

$$c_{x_K}[x_J - a_J] = \begin{cases} \infty & \text{if } x_K < x_J - a_J \\ 0 & \text{if } x_K \geq x_J - a_J. \end{cases}$$

That is, K will always outrank J if $x_K < x_J - a_J$, and K will never outrank J if $x_K \geq x_J - a_J$. This means J's delay due to K is

$$X_{x_K}^{\text{new}}[x_J - a_J] = x_K \mathbb{1}(x_K < x_J - a_J),$$

where $\mathbb{1}$ is the indicator function.

When analyzing the response time of SRPT, we need to know J's delay due to a *random* new job, meaning one with size drawn from the size distribution X. This is the new $(x_J - a_J)$-work,

$$X^{\text{new}}[x_J - a_J] = X_X^{\text{new}}[x_J - a_J] = X\mathbb{1}(X < x_J - a_J).$$

Example 4.4. Consider the SERPT system described in Example 3.5, in which *all* jobs have the same descriptor $\varnothing$ and the same two-point size distribution: jobs are size 2 with probability 1/2 and size 14 otherwise. The rank function is

$$r(\varnothing, a) = \mathbf{E}[X - a \mid X > a] = \begin{cases} 8 - a & \text{if } a < 2 \\ 14 - a & \text{if } a \geq 2, \end{cases}$$

as shown in Figure 3.1.

Suppose that a new job K arrives when J has age $a_J < 2$. The worst future rank of J depends crucially on J's size x_J.

- If $x_J = 2$, then J's worst future rank is its current rank, $8 - a_J$. In this case, the cutoff age for K is $c_\varnothing[8 - a_J] = 0$ because K's initial rank is 8, which is at least J's worst future rank $8 - a_J$. This means J's delay due to K is
$$X^{\text{new}}[8 - a_J] = X_\varnothing^{\text{new}}[8 - a_J] = 0.$$
- If instead $x_J = 14$, then J's worst future rank is 12. In this case, the cutoff age for K is $c_\varnothing[12] = 2$ because K will either complete or jump up to rank 12 when it reaches age 2. This means J's delay due to K is
$$X^{\text{new}}[12] = X_\varnothing^{\text{new}}[12] = 2.$$

4.4 The Vacation Transformation

We have seen how the Pessimism Principle shows us how long each new job delays the tagged job J. The question remains: how long does each old job delay J? This is much harder than the corresponding question for new jobs because *old jobs can have any age*, whereas new jobs always start at age 0.

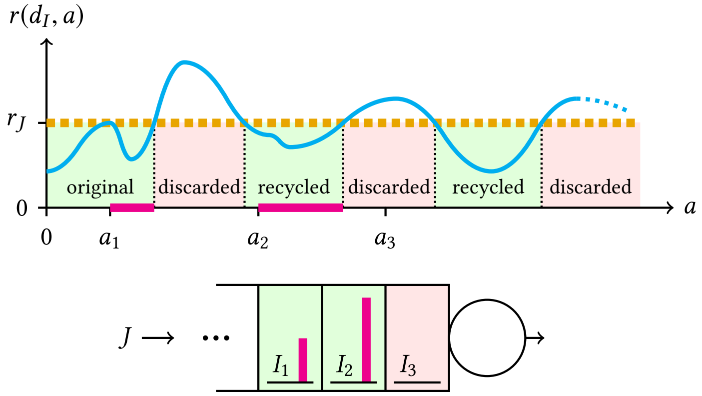
Fortunately, we are actually not directly concerned with the delay due to individual old jobs. What ultimately matters is the delay due to *all old jobs together.* It turns out that we can view this total delay as the *queueing time of a carefully transformed system.* The careful transformation in question is the Vacation Transformation, but we are still a few definitions away from presenting it.

Our goal is to find the J's total delay due to old jobs. We can think of this delay as the amount of "relevant" work in the system at the moment J arrives. Because Poisson arrivals see time averages [27], the distribution of the amount relevant work seen by J upon arrival is the *stationary distribution* of the amount of relevant work. Thus, in this section, we imagine J as a *witness* to the system, watching other jobs enter, receive service, and exit. For this purpose, the most important fact about J is its worst future rank upon arrival,

$$r_J = r_{d_J, x_J}^{\text{worst}}(0).$$

So far, we have considered old jobs as a monolithic category, but it is useful to consider three subcategories. At any moment in time, we can classify old jobs as follows.

- *Discarded* old jobs currently have rank greater than r_J.
- *Original* old jobs currently have rank at most r_J and *have always had* rank at most r_J since arriving themselves.
- *Recycled* old jobs currently have rank at most r_J but had rank greater than r_J at some point in the past.



A system contains jobs I_1, I_2, and I_3 with respective ages a_1, a_2, and a_3. All three jobs have the same descriptor d_I and therefore the same rank as a function of age, drawn in cyan. We view the system from the transformed perspective of witness J, which has worst future rank r_J. By the Pessimism Principle, if J were to arrive at the system now, it would only be delayed by original job I_1 and recycled job I_2, so discarded job I_3 is completely hidden from J. Furthermore, jobs I_1 and I_2 only delay J until they either complete or exceed rank r_J, so there are upper bounds on the delays due to each, shown as magenta bars.

Fig. 4.5. A Witness's Transformed View of the System

More generally, we may call a job discarded, original, or recycled *with respect to rank* r, in which case we replace r_J with r.

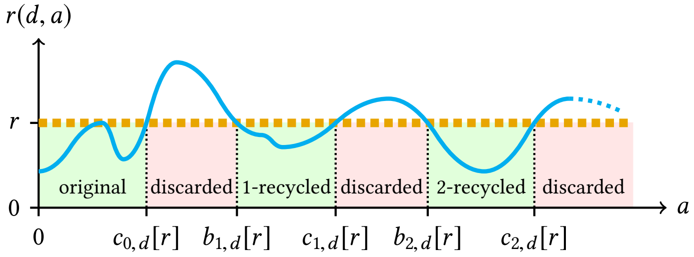
An old job I goes through the following transitions between these categories, as shown in Figures 4.5 and 4.6.

- When I arrives in the system, if its initial rank is at most r_J, it starts out original. Otherwise, it starts out discarded.
- As I ages, it may become discarded if it is not already.
- As I ages further, it may become recycled, then discarded again, then recycled again, and so on until it completes.

Eventually, J will arrive, and each of the old jobs will delay J by some amount of time based on their category at the moment when J arrives. By the Pessimism Principle, J's delay due to discarded jobs is 0, so such jobs do not concern us further. Original jobs are similar to new jobs but, due to the FCFS tiebreaking rule, not quite the same. Recycled jobs are the most difficult type of old job to handle, but fortunately, as we will soon see, J sees *at most one* recycled job in the system when it arrives.

For the purposes of analyzing J's response time, we view the system through "transformer glasses" [12] through which *only original and recycled jobs are visible*, as illustrated in Figure 4.5. In the transformed system, jobs are transformed such that when they become discarded, they complete. Thus, the total work in the transformed system is exactly the "relevant" work in the untransformed system, which would be J's total delay due to old jobs were J to arrive immediately. Therefore, our goal is to find the *stationary distribution of work in the transformed system.*

Both original and recycled jobs arrive in the transformed system. Arrivals of original jobs correspond to arrivals to the untransformed system, but arrivals of recycled jobs occur seemingly arbitrarily, as they are really caused by discarded jobs transitioning to recycled in the untransformed system. A busy period in the transformed system always starts with the arrival of an original or



As a job ages, it can transition repeatedly between being discarded and recycled with respect to a rank r. The i-old r-interval for descriptor d is the interval $[b_{i,d}[r], c_{i,d}[r]]$ during which a job of descriptor d is original ($i = 0$) or recycled for the ith time ($i \geq 1$). We highlight the i-old r-intervals in green. The i-old r-work is the amount of service the job requires while its age is in its i-old r-interval.

Fig. 4.6. Illustration of Old Work (Definition 4.5)

recycled job. Arrivals of original jobs continue during the busy period, but *no more recycled jobs arrive* for the rest of the busy period. This is because for a recycled job to arrive in the transformed system, a discarded job has to become recycled by receiving service in the untransformed system. But discarded jobs never outrank original or recycled jobs, which are present for the entire busy period in the transformed system, so such transitions never occur.

To analyze the amount of work in the transformed system, we need to know how long each old job spends as an original or recycled job. We call the amount of time an old job spends as original with respect to rank r its 0-*old* r-*work*, and we call the amount of time it spends as recycled for the ith time with respect to rank r its i-*old* r-*work*, both of which we now define formally.

Definition 4.5. Let r be a rank and d be a descriptor. The 0-*old* r-*interval* for descriptor d is the interval of ages during which a job of descriptor d is original with respect to rank r. Specifically, the interval is $[b_{0,d}[r], c_{0,d}[r]]$, where[7]

$$
\begin{aligned}
b_{0,d}[r] &= 0 \\
c_{0,d}[r] &= \inf\{a \geq b_{0,d}[r] \mid r(d, a) > r\}.
\end{aligned}
$$

For $i \geq 1$, the i-*old* r-*interval* for descriptor d is the interval of ages during which the job is recycled with respect to rank r for the ith time. Specifically, the interval is $[b_{i,d}[r], c_{i,d}[r]]$, where

$$
\begin{aligned}
b_{i,d}[r] &= \inf\{a > c_{i-1,d}[r] \mid r(d, a) \leq r\} \\
c_{i,d}[r] &= \inf\{a > b_{i,d}[r] \mid r(d, a) > r\}.
\end{aligned}
$$

If $b_{i,d}[r] = c_{i,d}[r] = \infty$, the interval is the empty set. See Figure 4.6 for an illustration.

For $i \geq 0$, the i-*old* r-*work* is a random variable, written $X_i^{\text{old}}[r]$, representing how long a job will be served while its age is in its i-old r-interval. Specifically, we define $X_i^{\text{old}}[r] = X_{i,D}^{\text{old}}[r]$, where D is the random descriptor assigned to a new job and, for any specific descriptor d,

$$
X_{i,d}^{\text{old}}[r] = \begin{cases} 0 & \text{if } X_d < b_{i,d}[r] \\ X_d - b_{i,d}[r] & \text{if } b_{i,d}[r] \leq X_d < c_{i,d}[r] \\ c_{i,d}[r] - b_{i,d}[r] & \text{if } c_{i,d}[r] \leq X_d. \end{cases}
$$

If $b_{i,d}[r] = c_{i,d}[r] = \infty$, we define $X_{i,d}^{\text{old}}[r] = 0$.

---

[7] See Appendix C for discussion of corner cases where we need $\geq$ instead of $>$.

Suppose old job I has a random descriptor. In the transformed system, I receives service

- for time $X_0^{\text{old}}[r_J]$ as an original job and,
- for all $i \geq 1$, for time $X_i^{\text{old}}[r_J]$ as a job being recycled for the ith time.

Note that $X_0^{\text{old}}[r_J]$ may be 0, representing I starting out discarded, and $X_i^{\text{old}}[r_J]$ for $i \geq 1$ may be 0, representing I completing before being recycled for the ith time.

Example 4.6. Consider SRPT as described in Example 3.4, in which a job's descriptor is its size and its rank is its remaining size. Suppose that J witnesses an old job I of initial size x_I. Under SRPT, every job's rank is strictly decreasing with age, so J's worst future rank is its initial size x_J. The amount of time I spends as an original or recycled job depends on its size relative to J's.

- If $x_I \leq x_J$, then I is original until its completion because its rank never exceeds x_J, so
$$X_{0,x_I}^{\text{old}}[x_J] = x_I.$$
I is never recycled, so $X_i^{\text{old}}[x_J] = 0$ for $i \geq 1$.
- If $x_I > x_J$, then I is starts out discarded but becomes recycled at age $x_I - x_J$, at which point it has remaining size x_J, so
$$X_{0,x_I}^{\text{old}}[x_J] = 0$$
$$X_{1,x_I}^{\text{old}}[x_J] = x_J.$$
I is recycled only once, so $X_{i,x_I}^{\text{old}}[x_J] = 0$ for $i \geq 2$.

When analyzing the response time of SRPT, we need to know the amount of time a *random* old job, meaning one with size drawn from the size distribution X, spends as an original or recycled job. From the above casework, we obtain
$$X_0^{\text{old}}[x_J] = X_{0,X}^{\text{old}}[x_J] = X\mathbb{1}(X \leq x_J)$$
$$X_1^{\text{old}}[x_J] = X_{1,X}^{\text{old}}[x_J] = x_J\mathbb{1}(X > x_J),$$
where $\mathbb{1}$ is the indicator function, and $X_i^{\text{old}}[x_J] = 0$ for $i \geq 2$.

The Vacation Transformation gives a simple specification of the amount of work the witness J sees in the transformed system. It follows from three observations.

- The amount of work in the transformed system is independent of the scheduling policy used on transformed jobs, provided it is work-conserving, so we may assume FCFS among original and recycled jobs[8].
- Because Poisson arrivals see time averages [27], the stationary amount of work in the transformed system is the same as the stationary FCFS queueing time of an original job in the transformed system.
- In the transformed system, the arrival process of recycled jobs is not Poisson, but they only appear at the starts of busy periods, so it is convenient to view them as server vacations.

Vacation Transformation. Consider the tagged job J with descriptor d_J and size x_J arriving to the system, and let
$$r_J = r_{d_J,x_J}^{\text{worst}}(0).$$
J's total delay due to old jobs has the same distribution as *queueing time in a transformed M/G/1/FCFS system with "sparse" server vacations*. We call the transformed system the *Vacation Transformation System*, or simply *VT System*. In the VT System, jobs arrive at rate λ and have size distribution $X_0^{\text{old}}[r_J]$, and several types of vacations occasionally occur, as described below.

---

[8] Recall that an original or recycled job completes the transformed system if it either completes *or becomes discarded* in the untransformed sysytem.

The server in the VT System is always in one of three states:

- *busy*, meaning a job is in service;
- *idle*, meaning the system is empty but the server is ready to start a job immediately should one arrive; or
- *on vacation*, meaning the server will not serve jobs until the vacation finishes, even if there are jobs in the system.

The server only starts vacations when the system is empty. Unlike job arrivals, vacation start times are not a Poisson process. Each vacation has a *type* $i \geq 1$ determining its length. Specifically, type i vacations have i.i.d. lengths drawn from distribution

$$V_i = (X_i^{\text{old}}[r_J] \mid X_i^{\text{old}}[r_J] > 0).$$

The stationary probability that the the server is on a type i vacation is $\lambda \mathbf{E}[X_i^{\text{old}}[r_J]]$. The exact process determining when a type i vacation starts is intractable, but to analyze queueing time in the VT System, as we do in Lemma 5.2, it fortunately suffices to know just this stationary probability.

5 RESPONSE TIME OF SOAP POLICIES

Having spent the previous section understanding the perspective of a tagged job in a system using a SOAP policy, we are now ready to apply our insights to analyze the response time of SOAP policies. Specifically, we analyze $T_{d,x}$, the response time of a tagged job J with descriptor d and size x.

When analyzing traditional policies like SRPT or PSJF (Example 3.4), it often helps to think of J's response time as the sum of two independent random variables [12, 22]:

- *waiting time*, written $T_{d,x}^{\text{wait}}$, the time from when J arrives to when it first enters service; and
- *residence time*, written $T_{d,x}^{\text{res}}$, the time from when J first enters service to when it exits.

Unfortunately, the usual strategy for analyzing waiting and residence time relies on J's rank never increasing, which holds for SRPT and PSJF but does not hold for a great many SOAP policies. To overcome this obstacle, we *replace J's rank with its worst future rank*, which never increases. This rank substitution, fully justified in Appendix D, is made possible by the Pessimism Principle (Section 4.3). We call J with its adjusted rank the *rank-substituted* tagged job.

After replacing J's rank with its worst future rank, it remains to analyze its waiting and residence times. Even though J's worst future rank is monotonic, other jobs' ranks may both increase and decrease with age, making these analyses challenging.

- We can think of *waiting time* as a transformed busy period: the initial transformed work is J's total delay due to old jobs, and each arriving new job's transformed size is the amount it delays J. The main challenge is finding the distribution of initial transformed work, which requires using the Vacation Transformation (Section 4.4). We analyze waiting time in Section 5.1.
- We would also like to think of *residence time* as a transformed busy period. The initial work is simply J's size x, but now the arriving new jobs present a challenge: because J's worst future rank may decrease with time, not all new jobs have the same transformed size distribution. We analyze residence time in Section 5.2.

5.1 Waiting Time

Hereafter, "transform" means Laplace-Stieltjes transform. We write the transform of a random variable V as $\tilde{V}(s)$ and the transform of a parametrized random variable $V[r]$ as $\tilde{V}[r](s)$.

Recall that the waiting time of the rank-substituted tagged job J is the amount of time between J's arrival and when J first enters service. As mentioned previously, we can think of waiting time as a transformed busy period: the initial work is J's total delay due to old jobs, and each arriving new job's size is the amount it delays J. This type of busy period is formalized in the following definition.

Definition 5.1. Let r be a rank. The *new r-work busy period*, written $B^{\text{new}}[r]$, is the length of a busy period in an M/G/1 system with arrival rate λ and job size $X^{\text{new}}[r]$. Its transform satisfies [12]

$$\tilde{B}^{\text{new}}[r](s) = \tilde{X}^{\text{new}}[r](s + \lambda(1 - \tilde{B}^{\text{new}}[r](s))).$$

More generally, the new r-work busy period *started by work W*, written $B^{\text{new}}_W[r]$, is the length of a busy period in the same M/G/1 system with a random initial amount of work W. It has transform [12]

$$\tilde{B}^{\text{new}}_W[r](s) = \tilde{W}(s + \lambda(1 - \tilde{B}^{\text{new}}[r](s))). \tag{5.1}$$

Let $r = r^{\text{worst}}_{d,x}(0)$ be J's worst future rank upon arrival. Because J is not served until its residence time, r remains J's worst future rank for the entirety of J's waiting time. This means J's delay due to each new job that arrives during its waiting time is $X^{\text{new}}[r]$, so J's waiting time is a new r-work busy period started by initial work W, where W is J's total delay due to old jobs.

All that remains is to determine W, for which it is convenient to define two new notations. First, let

$$\rho^{\text{new}}[r] = \lambda \mathbf{E}[X^{\text{new}}[r]]$$
$$\rho^{\text{old}}_i[r] = \lambda \mathbf{E}[X^{\text{old}}_i[r]]$$
$$\rho^{\text{old}}_\Sigma[r] = \sum_{i=0}^{\infty} \rho^{\text{old}}_i[r]$$

be the "loads contributed by" new r-work, i-old r-work, and all old r-work, respectively. Second, let $Y^{\text{old}}_i[r]$ be the *equilibrium distribution* [12], or length-biased sample, of $X^{\text{old}}_i[r]$. It has transform

$$\tilde{Y}^{\text{old}}_i[r](s) = \frac{1 - \tilde{X}^{\text{old}}_i[r](s)}{s\mathbf{E}[X^{\text{old}}_i[r]]}.$$

Note that $Y^{\text{old}}_i[r]$ is also the equilibrium distribution of the length of a type i vacation in the VT System, $V_i = (X^{\text{old}}_i[r] \mid X^{\text{old}}_i[r] > 0)$, because samples of length 0 are never encountered in a length-biased sample.

LEMMA 5.2. *Under any SOAP policy, the Laplace-Stieltjes transform of waiting time for a rank-substituted tagged job with descriptor d and size x is*

$$\tilde{T}^{\text{wait}}_{d,x}(s) = \frac{1 - \rho^{\text{old}}_\Sigma[r] + \sum_{i=1}^{\infty} \rho^{\text{old}}_i[r]\tilde{Y}^{\text{old}}_i[r](\sigma)}{1 - \rho^{\text{old}}_0[r]\tilde{Y}^{\text{old}}_0[r](\sigma)},$$

where $r = r^{\text{worst}}_{d,x}(0)$ *and* $\sigma = s + \lambda(1 - \tilde{B}^{\text{new}}[r](s))$.

PROOF. Call the rank-substituted tagged job J. As previously mentioned, $T^{\text{wait}}_{d,x}$ is a new r-work busy period started by initial work W, where W is J's total delay due to old jobs. This means $\tilde{T}^{\text{wait}}_{d,x}(s) = \tilde{W}(\sigma)$ by (5.1), so it remains only to compute $\tilde{W}(\sigma)$.

The Vacation Transformation states that W has the same distribution as the queueing time in the VT System, which is a particular M/G/1/FCFS system with vacations. A decomposition result of Fuhrmann and Cooper [10, Equation (4)] states that the number of jobs in an M/G/1/FCFS system with vacations, such as the VT System, is distributed as the sum of two independent random variables:

- the number N_Q of jobs in the queue of a vacation-free M/G/1/FCFS system; and

- the number N_V of jobs in the queue observed by an arriving job conditional on observing a non-busy server.

Recalling the Vacation Transformation and a standard result for the M/G/1 queue [12], we immediately obtain the probability generating function for N_Q,

$$\hat{N}_Q(z) = \frac{1 - \rho_0^{\text{old}}[r]}{1 - \rho_0^{\text{old}}[r]\tilde{Y}_0^{\text{old}}[r](\lambda(1-z))}. \tag{5.2}$$

In the VT System, a job arriving to a non-busy server observes N_V as one of the following.

- If the server is idle, which happens with probability $1 - \rho_\Sigma^{\text{old}}[r]$, there are 0 jobs in the queue.
- If the server is in the middle of a type i vacation, which happens with probability $\rho_i^{\text{old}}[r]$, the number of jobs in the queue is the number of Poisson arrivals at rate λ during time $Y_i^{\text{old}}[r]$, which is the amount of time since the start of the type i vacation.

Accounting for the fact that we measure N_V only when a job arrives to a non-busy server, which happens with probability $1 - \rho_0^{\text{old}}[r]$, we obtain the probability generating function of N_V,

$$\hat{N}_V(z) = \frac{1 - \rho_\Sigma^{\text{old}}[r] + \sum_{i=1}^{\infty} \rho_i^{\text{old}}[r]\tilde{Y}_i^{\text{old}}[r](\lambda(1-z))}{1 - \rho_0^{\text{old}}[r]}. \tag{5.3}$$

Multiplying (5.2) and (5.3), applying the distributional version of Little's Law[9] [14], and substituting $\sigma = \lambda(1-z)$ gives the transform of queueing time in the VT System, which matches the desired transform $\tilde{W}(\sigma)$. □

5.2 Residence Time

Lemma 5.3. *Under any SOAP policy, the Laplace-Stieltjes transform of residence time for a rank-substituted tagged job with descriptor d and size x is*

$$\tilde{T}_{d,x}^{\text{res}}(s) = \exp\left(-\lambda \int_0^x (1 - \tilde{B}^{\text{new}}[r_{d,x}^{\text{worst}}(a)](s))\, da\right).$$

Proof. We view residence time as a *sum of many small busy periods*, each started by a small amount of work δ. We then take the $\delta \longrightarrow 0$ limit, which exists thanks to conditions in Appendix B. This is very similar to the argument used to compute the transform of residence time under SRPT [22]. Specifically, we divide $[0, x]$ into chunks of size δ and consider each small busy period started by the work needed to bring the job from age a to age $a + \delta$. In the $\delta \longrightarrow 0$ limit, for the entirety of the small busy period starting at age a, we can assume the job has rank $r_{d,x}^{\text{worst}}(a)$.

By the Pessimism Principle, the amount of work in the small busy period starting at age a is the length of a new $r_{d,x}^{\text{worst}}(a)$-work busy period started by work δ, which by (5.1) has transform

$$\tilde{B}_\delta^{\text{new}}[r_{d,x}^{\text{worst}}(a)](s) = \exp(-\delta\lambda(1 - \tilde{B}^{\text{new}}[r_{d,x}^{\text{worst}}(a)](s))).$$

The lengths of the small busy periods are independent because the arrival process is Poisson, so the residence time, which is the total amount of work in all such busy periods, has transform

$$\prod_{i=0}^{x/\delta} \tilde{B}_\delta^{\text{new}}[r_{d,x}^{\text{worst}}(\delta i)](s) = \exp\left(-\delta\lambda \sum_{i=0}^{x/\delta} (1 - \tilde{B}^{\text{new}}[r_{d,x}^{\text{worst}}(\delta i)](s))\right).$$

Taking the $\delta \longrightarrow 0$ limit yields the desired expression. □

---

[9] Here we apply the distributional version of Little's Law to the VT System, not the original SOAP system, which is crucial because the law applies only to FCFS systems.

5.3 Total Response Time

Theorem 5.4 (SOAP Transform of Response Time). *Under any SOAP policy, the Laplace-Stieltjes transform of response time of jobs with descriptor d and size x is*

$$\tilde{T}_{d,x}(s) = \tilde{T}^{\text{wait}}_{d,x}(s)\tilde{T}^{\text{res}}_{d,x}(s),$$

where $\tilde{T}^{\text{res}}_{d,x}(s)$ and $\tilde{T}^{\text{wait}}_{d,x}(s)$ are as in Lemmas 5.2 and 5.3, respectively.

Proof. Because the arrival process is Poisson, $T^{\text{wait}}_{d,x}$ and $T^{\text{res}}_{d,x}$ are independent, so the result follows from Lemmas 5.2 and 5.3 □

Theorem 5.5 (SOAP Mean Response Time). *Under any SOAP policy, the mean response time of jobs with descriptor d and size x is*

$$\mathbf{E}[T_{d,x}] = \frac{\lambda \sum_{i=0}^{\infty} \mathbf{E}[(X_i^{\text{old}}[r])^2]}{2(1-\rho_0^{\text{old}}[r])(1-\rho^{\text{new}}[r])} + \int_0^x \frac{1}{1-\rho^{\text{new}}[r(a)]}\,da,$$

where $r(a) = r^{\text{worst}}_{d,x}(a)$ and $r = r^{\text{worst}}_{d,x}(0)$.

Proof. This follows from $\mathbf{E}[T_{d,x}] = -\tilde{T}'_{d,x}(0)$ after straightforward computation. □

We can use $T_{d,x}$ to analyze T_d, the response time of jobs with descriptor d and any size, and T, the overall response time of all jobs. Specifically, $\tilde{T}_d(s) = \mathbf{E}[\tilde{T}_{d,X_d}(s)]$, where X_d is the size distribution of jobs with descriptor d, and $\tilde{T}(s) = \mathbf{E}[\tilde{T}_D(s)]$, where D is the random descriptor assigned to a new job. Analyzing response time of jobs with size x and any descriptor is similar, but it requires computing D_x, the distribution of descriptors of jobs with size x.

6 NEW ANALYSES FOR SPECIFIC POLICIES

In this section, we analyze the response time of several policies discussed in Section 3.2. The main challenge to analyzing SOAP policies is determining $X^{\text{new}}[r]$ and $X_i^{\text{old}}[r]$. Throughout, we give expressions for these focusing only on ranks r that are important for the final result. Specifically, for the possible descriptors d and sizes x, we only need to find

- $X^{\text{new}}[r^{\text{worst}}_{d,x}(a)]$ for all ages a and
- $X_i^{\text{old}}[r^{\text{worst}}_{d,x}(0)]$.

For simplicity, we give final formulas only for mean response time.

6.1 Discretized FB

Consider the discretized FB policy (Example 3.7), which can only preempt jobs when their age is at specific checkpoints spaced 1 time unit apart. We represent this using the rank function

$$r(\varnothing, a) = \langle \lfloor a \rfloor - a, a \rangle,$$

shown in Figure 3.3. As in the analysis of traditional FB, it is convenient to work in terms of the *capped size distribution* and its associated load [12, Section 30.3],

$$\begin{aligned} X_{\bar{x}} &= \min\{X, x\} \\ \rho_{\bar{x}} &= \lambda \mathbf{E}[X_{\bar{x}}]. \end{aligned} \tag{6.1}$$

In discretized FB, a job of size x and age a has worst future rank

$$r^{\text{worst}}_{\varnothing,x}(a) = \begin{cases} \langle 0, \lfloor x \rfloor \rangle & \text{if } a \le \lfloor x \rfloor \\ \langle \lfloor a \rfloor - a, a \rangle & \text{if } a > \lfloor x \rfloor. \end{cases}$$

The new r-work and 0-old r-work relevant for obtaining response time are therefore

$$\begin{aligned} X^{\text{new}}[\langle 0, \lfloor x \rfloor \rangle] &= X_{\overline{\lfloor x \rfloor}} \\ X^{\text{new}}[\langle \lfloor a \rfloor - a, a \rangle] &= 0 \\ X_0^{\text{old}}[\langle 0, \lfloor x \rfloor \rangle] &= X_{\overline{\lfloor x \rfloor + 1}}, \end{aligned}$$

where $\lfloor x \rfloor < a < x$ in the second equation. From Theorem 5.5, we conclude the following.

Proposition 6.1. *Under discretized FB (Example 3.7), the mean response time of jobs with size x is*[10]

$$\mathbf{E}[T_x] = \frac{\lambda \mathbf{E}\big[X^2_{\overline{\lceil x \rceil}}\big]}{2\big(1 - \rho_{\overline{\lceil x \rceil}}\big)\big(1 - \rho_{\overline{\lfloor x \rfloor}}\big)} + \frac{\lfloor x \rfloor}{1 - \rho_{\overline{\lfloor x \rfloor}}} + x - \lfloor x \rfloor.$$

6.2 Mixture of Known and Unknown Job Sizes

Consider the "humans and robots" system (Example 3.8). This scenario has two features that were previously difficult to analyze.

- Some jobs have known size, namely robots, while others have unknown size, namely humans.
- Some jobs are preemptible, namely robots, while others are nonpreemptible, namely humans.

Let

- X_H and X_R be the respective size distributions of humans and robots,
- p_H and $p_R = 1 - p_H$ be the respective probabilities that a given arrival is a human or a robot, and
- $\lambda_H = \lambda p_H$ and $\lambda_R = \lambda p_R$ be the respective arrival rates of humans and robots.

Recall that humans all have descriptor $[H, ?]$, indicating that their size is unknown, and robots each have a descriptor of the form $[R, x]$, indicating that their exact size is x. The rank function is

$$\begin{aligned} r([H, ?], a) &= \langle -a, x_H \rangle \\ r([R, x], a) &= \langle 0, x - a \rangle, \end{aligned}$$

where x_H is a constant.

Both humans and robots have maximal primary rank, namely 0, upon entering the system, so the only nonzero new r-work and i-old r-work occur for ranks of the form $r = \langle 0, x \rangle$:

$$X^{\text{new}}[\langle 0, x \rangle] = \begin{cases} X_H \mathbb{1}(x_H < x) & \text{with probability } p_H \\ X_R \mathbb{1}(X_R < x) & \text{with probability } p_R \end{cases}$$

$$X_0^{\text{old}}[\langle 0, x \rangle] = \begin{cases} X_H \mathbb{1}(x_H \le x) & \text{with probability } p_H \\ X_R \mathbb{1}(X_R \le x) & \text{with probability } p_R \end{cases}$$

$$X_1^{\text{old}}[\langle 0, x \rangle] = \begin{cases} X_H \mathbb{1}(x_H > x) & \text{with probability } p_H \\ x \mathbb{1}(X_R > x) & \text{with probability } p_R, \end{cases}$$

where $\mathbb{1}$ is the indicator function. These follow from arguments similar to those given for SRPT in Examples 4.3 and 4.6. Finally, $X_i^{\text{old}}[r] = 0$ for $i \ge 2$, so Theorem 5.5 yields the following.

---

[10] See Appendix C for discussion of why we use $\lceil x \rceil$ instead of $\lfloor x \rfloor + 1$.

Proposition 6.2. *In the humans and robots system (Example 3.8), the mean response time of humans is*

$$\mathbf{E}[T_H] = \frac{\lambda_H \mathbf{E}[X_H^2] + \lambda_R \mathbf{E}[(X_R)^2_{\overline{x_H}}]}{2(1 - \rho_H - \rho_{R \le x_H})(1 - \rho_{R < x_H})} + \mathbf{E}[X_H],$$

and the mean response time of robots with size x is

$$\mathbf{E}[T_{R,x}] = \frac{\lambda_H \mathbf{E}[X_H^2] + \lambda_R \mathbf{E}[(X_R)^2_{\bar{x}}]}{2(1 - \rho_H \mathbb{1}(x_H \le x) - \rho_{R \le x})(1 - \rho_{R < x_H})} + \int_0^x \frac{1}{1 - \rho_H \mathbb{1}(x_H \le t) - \rho_{R < t}}\, dt,$$

where $(X_R)_{\overline{x_H}}$ and $(X_R)_{\bar{x}}$ are capped distributions as defined in (6.1) and

$$\begin{aligned} \rho_H &= \lambda_H \mathbf{E}[X_H] \\ \rho_{R<x} &= \lambda_R \mathbf{E}[X_R \mathbb{1}(X_R < x)] \\ \rho_{R \le x} &= \lambda_R \mathbf{E}[X_R \mathbb{1}(X_R \le x)]. \end{aligned}$$

6.3 The Gittins Index Policy

As discussed in Example 3.6, the Gittins index policy can have a nonmonotonic rank function, and thus only special cases of it have been analyzed in the past [13, 20]. Theorems 5.4 and 5.5 give us exactly the framework we need to analyze the Gittins index policy used with *any* set of descriptors and job size distributions, though $X^{\text{new}}[r]$ and $X_i^{\text{old}}[r]$ do not have a general closed form and thus require details of the system to derive. We start in this section by using the SOAP framework to analyze the model considered by Osipova et al. [20], which is relatively simple thanks to the fact that the rank functions involved are monotonic. In Section 6.4, we move to a more difficult setting in which the Gittins index policy has a nonmonotonic rank function.

We consider a system with two job classes A and B, which serve as our descriptors, with respective arrival rates $\lambda_A = \lambda p_A$ and $\lambda_B = \lambda p_B$ and respective Pareto size distributions X_A and X_B. Specifically,

$$\mathbf{P}\{X_A > t\} = \left(1 + \frac{t}{\beta_A}\right)^{-\alpha_A}$$

and symmetrically for B, where $\alpha_A, \alpha_B > 1$ and $\beta_A, \beta_B > 0$ are parameters of the distributions. The rank function is [3, 20]

$$r(A, a) = \frac{1}{G(A, a)} = \frac{\beta_A + a}{\alpha_A}$$

and symmetrically for B. The rank is strictly increasing in a, so $r_{d,x}^{\text{worst}}(a) = r(d, x)$ for all ages $a < x$. Strictly increasing rank also means jobs are never recycled, so $X_i^{\text{old}}[r] = 0$ for all $i \ge 1$.

It remains only to compute $X^{\text{new}}[r]$ and $X_0^{\text{old}}[r]$. As in Definition 4.2, let

$$c_A[r] = \max\{\alpha_A r - \beta_A, 0\}$$

be the age at which a class A job first surpasses rank r, and symmetrically for B. Note that $c_A[r(A, x)] = x$. Because the rank function is strictly increasing,

$$X^{\text{new}}[r] = X_0^{\text{old}}[r] = \begin{cases} (X_A)_{\overline{c_A[r]}} & \text{with probability } p_A \\ (X_B)_{\overline{c_B[r]}} & \text{with probability } p_B, \end{cases}$$

where $(X_A)_{\overline{c_A[r]}}$ and $(X_B)_{\overline{c_B[r]}}$ are capped distributions as defined in (6.1). By Theorem 5.5,

$$\mathbf{E}[T_{A,x}] = \frac{\lambda_A \mathbf{E}[(X_A)^2_{\bar{x}}] + \lambda_B \mathbf{E}[(X_B)^2_{\bar{y}}]}{2(1 - \rho_{\bar{x},\bar{y}})^2} + \frac{x}{1 - \rho_{\bar{x},\bar{y}}},$$

where

$$\begin{aligned} y &= c_B[r(A, x)] \\ \rho_{\bar{x},\bar{y}} &= \lambda_A \mathbf{E}[(X_A)_{\bar{x}}] + \lambda_B \mathbf{E}[(X_B)_{\bar{y}}]. \end{aligned}$$

Of course, $\mathbf{E}[T_{B,x}]$ is symmetrical. It is simple to verify that, modulo notation, this matches the results of Osipova et al. [20].

6.4 Case Study: SERPT vs. the Gittins Index

We now analyze both SERPT and the Gittins index policy on the size distribution introduced in Example 3.5. Both policies have nonmonotonic rank functions in this case, so we need the full power of Theorem 5.5 to compute mean response times. All jobs have descriptor $\varnothing$ and the same two-point size distribution. We write the size distribution as $X = \mathbf{CoinFlip}\{2, 14\}$, meaning jobs have size 2 with probability $1/2$ and size 14 otherwise.

We begin by analyzing SERPT. We have computed $X^{\mathrm{new}}[r]$ for SERPT with this size distribution in Example 4.4,

$$X^{\mathrm{new}}[r] = \begin{cases} 0 & r \le 8 \\ 2 & r > 8 \end{cases}$$

We only need to compute i-old r-work for ranks $r^{\mathrm{worst}}_{\varnothing,2}(0) = 8$ and $r^{\mathrm{worst}}_{\varnothing,14}(0) = 12$. From the rank function plot in Figure 6.1, we see

$$\begin{aligned} X^{\mathrm{old}}_0[8] &= 2 \\ X^{\mathrm{old}}_0[12] &= \mathbf{CoinFlip}\{2, 14\} \\ X^{\mathrm{old}}_1[8] &= \mathbf{CoinFlip}\{0, 8\} \\ X^{\mathrm{old}}_1[12] &= 0. \end{aligned}$$

The most subtle of these is $X^{\mathrm{old}}_1[8]$: the total time an old job I spends as recycled with respect to rank 8 is either 0, if I has size 2, or 8, if I has size 14. Finally, $X^{\mathrm{old}}_i[r] = 0$ for $i \ge 2$. Applying Theorem 5.5 yields size-specific mean response times

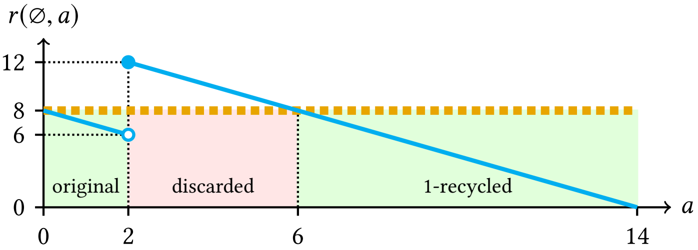
$$\begin{aligned} \mathbf{E}[T^{\mathrm{SERPT}}_2] &= \frac{18\lambda}{1 - 2\lambda} + 2 \\ \mathbf{E}[T^{\mathrm{SERPT}}_{14}] &= \frac{50\lambda}{(1 - 8\lambda)(1 - 2\lambda)} + \frac{6}{1 - 2\lambda} + 8. \end{aligned}$$

The Gittins index policy for the same system has rank function $r(\varnothing, a) = 1/G(\varnothing, a)$, where, by the definition in Example 3.6,

$$\frac{1}{G(\varnothing, a)} = \begin{cases} 4 - 2a & \text{if } a < 2 \\ 14 - a & \text{if } a \ge 2. \end{cases}$$

This rank function is illustrated in Figure 3.2. Broadly speaking, the Gittins index policy places higher priority on jobs of age $a < 2$ than SERPT does. Like that of SERPT, the rank function is piecewise linear with negative slopes, so we omit the very similar analysis and simply state the size-specific mean response times:

$$\begin{aligned} \mathbf{E}[T^{\mathrm{Gittins}}_2] &= \frac{6\lambda}{1 - 2\lambda} + 2 \\ \mathbf{E}[T^{\mathrm{Gittins}}_{14}] &= \frac{50\lambda}{(1 - 8\lambda)(1 - 2\lambda)} + \frac{10}{1 - 2\lambda} + 4. \end{aligned}$$



The rank function for SERPT where jobs have two-point size distribution **CoinFlip**$\{2, 14\}$, meaning size 2 with probability 1/2 and size 14 otherwise. The same rank function appears in Fig. 3.1. The 0-old (original) and 1-old (1-recycled) intervals are highlighted in green. A job is original with respect to rank 8 until age 2, when its rank jumps up if it does not complete. Upon reaching age 6, the job has remaining size 8, so it becomes recycled.

Fig. 6.1. Original and Recycled Work in SERPT

As expected due to its prioritization of jobs of age $a < 2$, the Gittins index policy has shorter mean response time for jobs of size 2 but longer mean response time for jobs of size 14. The Gittins index policy is known to minimize overall mean response time, and it performs as promised:

$$\begin{aligned}\mathbf{E}[T_2^{\text{SERPT}}] - \mathbf{E}[T_2^{\text{Gittins}}] &= \frac{12\lambda}{1-2\lambda}\\ \mathbf{E}[T_{14}^{\text{SERPT}}] - \mathbf{E}[T_{14}^{\text{Gittins}}] &= \frac{-8\lambda}{1-2\lambda}.\end{aligned}$$

Because the two job sizes are equally likely, the Gittins index policy has lower overall mean response time than SERPT.

7 CONCLUSION

We introduce *SOAP policies*, a very broad class of scheduling policies for the M/G/1 queue. The characteristic feature of a SOAP policy is its *rank function*, which maps each possible state a job could be in to a *rank*, meaning priority level. The SOAP class includes many policies old and new. While the mean response times of some relatively simple SOAP policies have been analyzed previously, the vast majority of SOAP policies, in particular those with *nonmonotonic* rank functions, have resisted analysis. Using two key technical insights, the *Pessimism Principle* and the *Vacation Transformation*, we overcome the obstacles presented by nonmonotonic rank functions to present a *universal response time analysis* that applies to any SOAP policy.

Our universal analysis applies to some notable policies. Among these is the *Gittins index policy*, which has long been known to minimize mean response time in settings where exact job sizes are not known. While prior work [13, 20] was restricted to the case of known job sizes or distributions with the decreasing hazard rate property, our analysis can handle the Gittins index policy with *arbitrary size distributions*, which was previously intractable. Our universal analysis also applies to several *practically motivated systems*, such as those in which jobs are only preemptible at certain checkpoints or only some jobs' exact sizes are known. More broadly, we are optimistic that techniques similar to our Pessimism Principle and Vacation Transformation could help analyze the response times of scheduling policies in more complex M/G/1 settings, such as systems with setup times or server vacations.

ACKNOWLEDGMENTS

We thank Peter van de Ven and the anonymous referees for their helpful comments. Ziv Scully was supported by an ARCS Foundation scholarship and the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1745016. Mor Harchol-Balter was supported by NSF-XPS-1629444, NSF-CMMI-1538204, NSF-CMMI-1334194, and a Faculty Award from Google.

REFERENCES

[1] Samuli Aalto and Urtzi Ayesta. 2006. Mean delay analysis of multi level processor sharing disciplines. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings.* IEEE, 1–11.

[2] Samuli Aalto, Urtzi Ayesta, Sem Borst, Vishal Misra, and Rudesindo Núñez-Queija. 2007. Beyond processor sharing. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 34. ACM, 36–43.

[3] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. 2009. On the Gittins index in the M/G/1 queue. *Queueing Systems* 63, 1 (2009), 437–458.

[4] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. 2011. Properties of the Gittins index with application to optimal scheduling. *Probability in the Engineering and Informational Sciences* 25, 03 (2011), 269–288.

[5] Konstantin Avrachenkov, Patrick Brown, and Natalia Osipova. 2009. Optimal choice of threshold in two level processor sharing. *Annals of Operations Research* 170, 1 (2009), 21–39.

[6] Urtzi Ayesta, Onno J. Boxma, and Ina Maria Verloop. 2012. Sojourn times in a processor sharing queue with multiple vacations. *Queueing Systems* 71, 1 (2012), 53–78.

[7] Sem Borst, Rudesindo Núñez-Queija, and Bert Zwart. 2006. Sojourn time asymptotics in processor-sharing queues. *Queueing Systems* 53, 1 (2006), 31–51.

[8] Jacqueline Boyer, Fabrice Guillemin, Philippe Robert, and Bert Zwart. 2002. Heavy tailed M/G/1-PS queues with impatience and admission control in packet networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, Vol. 1. IEEE, 186–195.

[9] Hanhua Feng and Vishal Misra. 2003. Mixed scheduling disciplines for network flows. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31. ACM, 36–39.

[10] Steve W. Fuhrmann and Robert B. Cooper. 1985. Stochastic decompositions in the M/G/1 queue with generalized vacations. *Operations research* 33, 5 (1985), 1117–1129.

[11] John Gittins, Kevin Glazebrook, and Richard Weber. 2011. *Multi-armed Bandit Allocation Indices.* John Wiley & Sons.

[12] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action* (1st ed.). Cambridge University Press, New York, NY, USA.

[13] Esa Hyytiä, Samuli Aalto, and Aleksi Penttinen. 2012. Minimizing slowdown in heterogeneous size-aware dispatching systems. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 29–40.

[14] J. Keilson and L. D. Servi. 1988. A distributional form of Little's law. *Operations Research Letters* 7, 5 (1988), 223–227.

[15] David G Kendall. 1953. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics* (1953), 338–354.

[16] Leonard Kleinrock. 1967. Time-shared systems: A theoretical treatment. *Journal of the ACM (JACM)* 14, 2 (1967), 242–261.

[17] Leonard Kleinrock. 1976. *Queueing Systems, Volume 2: Computer Applications.* Vol. 66. Wiley New York.

[18] Minghong Lin, Adam Wierman, and Bert Zwart. 2010. The average response time in a heavy-traffic SRPT queue. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 38. ACM, 12–14.

[19] Misja Nuyens, Adam Wierman, and Bert Zwart. 2008. Preventing large sojourn times using SMART scheduling. *Operations Research* 56, 1 (2008), 88–101.

[20] Natalia Osipova, Urtzi Ayesta, and Konstantin Avrachenkov. 2009. Optimal policy for multi-class scheduling in a single server queue. In *Teletraffic Congress, 2009. ITC 21 2009. 21st International.* IEEE, 1–8.

[21] Linus E Schrage. 1967. The queue M/G/1 with feedback to lower priority queues. *Management Science* 13, 7 (1967), 466–474.

[22] Linus E Schrage and Louis W Miller. 1966. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research* 14, 4 (1966), 670–684.

[23] Lajos Takács. 1963. Delay distributions for one line with Poisson input, general holding times, and various orders of service. *Bell Labs Technical Journal* 42, 2 (1963), 487–503.

[24] Adam Wierman. 2007. Fairness and classifications. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 34. ACM, 4–12.

[25] Adam Wierman, Mor Harchol-Balter, and Takayuki Osogami. 2005. Nearly insensitive bounds on SMART scheduling. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 33. ACM, 205–216.

[26] Adam Wierman and Misja Nuyens. 2008. Scheduling despite inexact job-size information. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 36. ACM, 25–36.
[27] Ronald W. Wolff. 1982. Poisson arrivals see time averages. *Operations Research* 30, 2 (1982), 223–231. https://doi.org/10.1287/opre.30.2.223 arXiv:https://doi.org/10.1287/opre.30.2.223

A EXTENSION TO LCFS TIEBREAKING

SOAP policies that use LCFS tiebreaking admit almost exactly the same analysis as those that use FCFS tiebreaking. As explained in detail below, the entire analysis is unchanged except for *reversing the strictness of rank comparisons* in Definitions 4.2 and 4.5, meaning swapping $\prec$ and $\preceq$ and swapping $\succ$ and $\succeq$.

Throughout Sections 4.3 and 4.4, which follow a tagged job J through the system, we distinguish between *new* jobs, which arrive after J, and *old* jobs, which arrive before J. When there are multiple jobs of minimal rank, FCFS tiebreaking prioritizes old jobs, then J, and then new jobs. This prioritization affects the *strictness of rank comparisons* when defining new r-work and old r-work. For instance, Definition 4.2 defines

$$c_d[r] = \inf\{a \geq 0 \mid r(d,a) \succeq r\},$$

whereas Definition 4.5 defines

$$c_{0,d}[r] = \inf\{a \geq 0 \mid r(d,a) \succ r\},$$

which is the same but with $\succ$ in place of $\succeq$. When J's worst future rank is r, the above values each represent a "cutoff age" before which a job of descriptor d outranks J. Under FCFS tiebreaking, a new job K outranks J until K's rank is *at least* r, whereas an old job I outranks J until I's rank *strictly exceeds* r. Under LCFS tiebreaking, this situation is reversed, which manifests as reversing the strictness of rank comparisons.

B RANK FUNCTION DETAILS

In order to ensure that a SOAP policy is well-defined, its rank function r must satisfy the following conditions.

- With respect to descriptor, r must be *piecewise continuous* to ensure that certain expectations are well-defined.
- With respect to age, r must be *piecewise monotonic* and *piecewise differentiable* to determine when and how to share the processor between multiple jobs. Any compact region of $\mathbb{R}_{\geq 0}$ must contain only finitely many boundary points between pieces. Furthermore, upwards jump discontinuities must be continuous from the right[11].

These conditions allow us to define a SOAP policy as the limit of discrete-time priority policies, with the limit taken as the discretization increment approaches 0. When $\mathcal{R} = \mathbb{R}^2$ ordered lexicographically, the limiting policy is Algorithm B.1, which has a clear generalization to $\mathcal{R} = \mathbb{R}^n$. In Algorithm B.1, we say a job is "in state (d, a)" to mean it has descriptor d and age a.

C WORST FUTURE RANK DETAILS

For simplicity of exposition, throughout Section 4, we assumed that a job's worst future rank $r_{d,x}^{\mathrm{worst}}(a)$ was actually attained by that job in the future. However, there are two cases where the supremum in Definition 4.1 is *not* be attained by some age a: when there is a jump discontinuity or when the maximum is at the open boundary $a = x$. For example, in Section 6.1, if a job has integer size x, then the job never attains rank $\langle 0, x\rangle$.

---

[11] That is, if a job jumps from low rank to high rank at age a, its rank exactly at age a should be the high rank.

Algorithm B.1 SOAP Policy in Continuous Time

Let $\mathcal{J}$ be the set of jobs in states (d, a) of minimal $r_1(d, a)$.

- Within $\mathcal{J}$, consider jobs such that r is strictly decreasing in age. If there are any, schedule the job of minimal r_2, using FCFS tiebreaking if there are multiple such jobs.
- Otherwise, within $\mathcal{J}$, consider jobs such that r_1 is constant and r_2 is strictly increasing in age. If there are any, share the processor between all such jobs, giving a job in state (d, a) share proportional to $1/\partial_a r_2(d, a)$.
- Otherwise, $\mathcal{J}$ must only contain jobs such that r_1 is strictly increasing. Share the processor between jobs in $\mathcal{J}$, giving a job in state (d, a) share proportional to $1/\partial_a r_1(d, a)$.

There are multiple ways to remedy the situation. The most intuitive is to say that $r^{\text{worst}}_{d,x}(a)$ is not a rank but a *rank bound.* The set of rank bounds is $\mathcal{R} \times \{-1, 0\}$ ordered lexicographically. An ordinary rank r corresponds to the pair $(r, 0)$ representing the *closed* upper bound $r' \leq r$ over other ranks r', but the pair $(r, -1)$ is "just below" $(r, 0)$, representing the *open* upper bound $r' < r$.

The corrections to definitions are as follows. In Definition 4.1, we define the worst future rank to be the rank bound

$$r^{\text{worst}}_{d,x}(a) = (\sup_{a \leq b < x} r(d, b), -\mathbb{1}(\text{the supremum is not attained}))$$

instead of just a rank. In Definitions 4.2 and 4.5, instead of defining r-work for a rank r, we define (r, q)-work for rank bounds (r, q). When we compare a rank r' against a rank bound (r, q), we compare rank bound $(r', 0)$ against (r, q). Concretely, in Definition 4.2, we define

$$c_d[(r, q)] = \inf\{a \geq 0 \mid (r(d, a), 0) \geq (r, q)\},$$

and similarly for $b_{i,d}[(r, q)]$ and $c_{i,d}[(r, q)]$ in Definition 4.5.

For example, consider the analysis of discretized FB in Section 6.1. When a job has integer size x, the supremum in $r^{\text{worst}}_{\varnothing,x}(a)$ is attained only in the $b \to x$ limit. Thus, $X_0^{\text{old}}[r^{\text{worst}}_{\varnothing,x}(0)] = X_{\overline{x}}$ when x is an integer, not $X_{\overline{x+1}}$ as would follow from the uncorrected Definition 4.5. To correct for this, Proposition 6.1 uses $\lceil x \rceil$ instead of $\lfloor x \rfloor + 1$.

The above discussion assumes FCFS tiebreaking. As discussed in Appendix A, the strictness of rank comparisons in Definitions 4.2 and 4.5 is reversed under LCFS tiebreaking, but the same changes described above apply without issue.

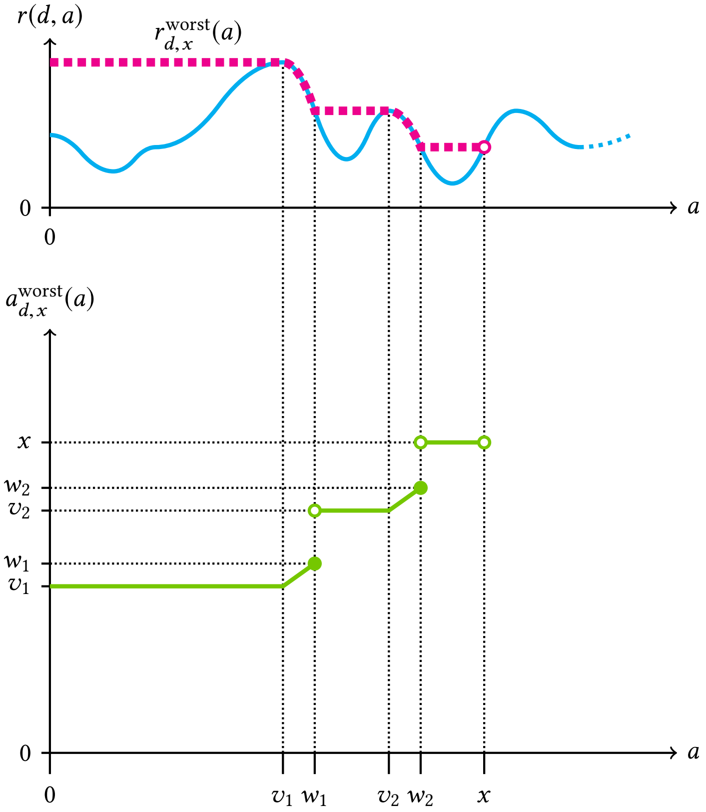
D THE RANK-SUBSTITUTED TAGGED JOB

In this section, we show that for the purposes of analyzing $T_{d,x}$, we can use a *rank-substituted* tagged job.

Worst Future Rank Substitution. Consider an arbitrary arrival sequence that includes the arrival of a tagged job J with descriptor d and size x. The response time of J is unaffected if we *schedule J as if its rank were $r^{\text{worst}}_{d,x}(a)$ instead of* $r(d, a)$ at every age a, without otherwise changing the arrival sequence or scheduling policy. We call this process *rank substitution.*

Worst Future Rank Substitution is a direct consequence of the Pessimism Principle (Section 4.3). To see why it holds, consider two systems experiencing identical job arrivals, including tagged job J.

- *System A* is unmodified, so J is scheduled according to its current rank.
- *System B* uses rank substitution, so J is scheduled according to its worst future rank.



The relationship between rank $r(d, a)$ (solid cyan), worst future rank $r^{\text{worst}}_{d,x}(a)$ (dashed magenta), and worst future age $a^{\text{worst}}_{d,x}(a)$ (solid green) for a job with descriptor d and size x. Age and worst future age coincide for ages in the intervals $[v_1, w_1]$ and $[v_2, w_2]$.

Fig. D.1. Illustration of Worst Future Age

We say the two systems *synchronize* at time t if they contain the same jobs at the same ages at t. The systems clearly synchronize at J's arrival time. We show below that the systems also synchronize at many other points in time, one of which is J's exit time, so J's response time is the same in each system.

The Pessimism Principle states that all of J's delay due to another job L occurs *before J is served while at its worst future rank*. This suggests we should focus on J's *worst future age*, which when J has age a is

$$a^{\text{worst}}_{d,x}(a) = \inf\{b \geq a \mid r(d, b) = r^{\text{worst}}_{d,x}(b)\},$$

namely the earliest age at which J attains its worst future rank[12]. See Figure D.1 for an illustration. Note that $a = a^{\text{worst}}_{d,x}(a)$ if and only if $r(d, a) = r^{\text{worst}}_{d,x}(a)$.

As J ages, its worst future age alternates between being a constant future age and its current age. Let $[v_i, w_i]$ be the ith interval of ages a such that $a = a^{\text{worst}}_{d,x}(a)$. It is convenient to set $w_0 = 0$ and

---

[12] As discussed in Appendix C, a job's worst future rank is sometimes only attained in a limit due to the job's completion or a jump in the rank function. Accounting for this changes only minor details in the following discussion.

$v_{n+1} = x$, where n is the number of $[v_i, w_i]$ intervals. We will show that System A and System B synchronize when, for some i, either

- J has age v_i and is in service or
- J's age is in $(v_i, w_i]$.

It is clear that if the systems synchronize when J is served at age v_i, then the systems remain synchronized until J reaches age w_i, because J's rank in the two systems is identical until J reaches age w_i. Thus, it suffices to show that if the systems synchronize when J has age w_{i-1}, then the systems synchronize when J is served at age v_i.

Consider how the two systems change during the interval between their synchronization when J has age w_{i-1}. Let t_{A} be the moment when J is served at age v_i in System A, and symmetrically for t_{B} in System B.

- In System A, by the Pessimism Principle, each other job L in the system during the interval is served until it either completes or surpasses J's worst future rank $r(d, v_i)$, after which L is never served again. This is because $a^{\mathrm{worst}}_{d,x}(a) = v_i$ for all $a \in (w_{i-1}, v_i]$.
- In System B, by the Pessimism Principle, each other job L is served for the same amount of time as in System A, because rank substitution does not change J's worst future rank.

Thus, if System A experiences the same arrivals before t_{A} that System B experiences before t_{B}, then the systems synchronize at $t_{\mathrm{A}} = t_{\mathrm{B}}$, as desired. Suppose for contradiction that one system, say System A, experiences an extra arrival. Because the arrival sequence is the same for the two systems, this occurs only if $t_{\mathrm{A}} > t_{\mathrm{B}}$. But by work conservation and the observations above, at t_{B}, System A must serve J at age v_i, so $t_{\mathrm{A}} = t_{\mathrm{B}}$ after all, contradicting $t_{\mathrm{A}} > t_{\mathrm{B}}$.