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

Ziv Scully
Carnegie Mellon University
Computer Science Department
zscully@cs.cmu.edu

Mor Harchol-Balter
Carnegie Mellon University
Computer Science Department
harchol@cs.cmu.edu

Alan Scheller-Wolf
Carnegie Mellon University
Tepper School of Business
awolf@andrew.cmu.edu

ABSTRACT
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.

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 early works [10, 15–17, 21–23] that are now classic to more recent works [1, 2, 6–9, 18–20, 24–26] in the SIGMETRICS community. 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 jobs either preemptively or nonpreemptively by selecting a job from the class of highest priority possible;
- **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
- **processor sharing** (PS), which concurrently serves all jobs in the system at the same reduced 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 mentioned above, results are ad-hoc and limited to relatively simple policies. Analyzing variants of the above simple policies, let alone policies that are fundamentally different from those above, 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 priority 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 new and old 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 5.4) and Laplace-Stieltjes transform (Theorem 5.3). 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 accomplishes this by encoding an entire scheduling policy as a single rank function, which maps each job to a priority level, or rank. Specifically, 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 5.3),

\(^1\)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.
whereas in SRPT, a job’s rank is its remaining time (Example 3.4). As shown in Section 3.2, rank functions are extremely expressive.

The second major challenge is to analyze policies with arbitrary rank functions. In particular, previously analyzed scheduling policies, when expressed as SOAP policies, nearly all 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 the SMART [25] and multilevel processor sharing (MLPS) [17] classes.

- 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. Ospina 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 is 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 scheduling policies for the M/G/1 queue. We write $\lambda$ for the total arrival rate and $X$ for the overall job size distribution. We assume a stable system, meaning $\lambda E[X] < 1$, 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 $D$ 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 states and ranks, which are other types of tuples introduced below, we use different delimiters for each: descriptors in [square brackets], states in (parentheses), and ranks in (angle brackets).

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

- If jobs do not reveal their exact size upon entering the system, then $D = K$, 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 $D = K \times \mathbb{R}_{\geq 0}$, 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 $D = K \times (\mathbb{R}_{\geq 0} \cup \{?\})$, 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 last scenario, having each job independently reveal its size with some probability $p$ is permitted, but having alternating arrivals reveal their sizes is not.

The pair $(d, a)$ of a job’s descriptor $d \in D$ and age $a \geq 0$ is its state. The state of a job contains all the information known about it. In particular, a job’s state determines its remaining size distribution $X_d(a) = (X_d - a \mid X_d > a)$.

2.2 SOAP Policies and Rank Functions

A SOAP scheduling policy is a preemptive priority policy where a job’s priority depends on its state. SOAP is an acronym for Sched-ule 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

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

assigning a rank to each state.

The defining property of SOAP policies is the following.

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

Ties between jobs of the same rank are broken using a *first-come, first-served* (FCFS) tiebreaking rule, namely by serving the job that arrived to the system first.

We say that a job $J$ in state $(d, a)$ outranks a job $K$ in state $(d', a')$ if $r(d, a) < r(d', a')$ or both $r(d, a) = r(d', a')$ and $J$ arrived to the system before $K$. SOAP policies always serve the job that outranks all other jobs in the system.

For simplicity of exposition, we usually restrict our attention to $\mathcal{R} = \mathbb{R}$ and $\mathcal{R} = \mathbb{R}^2$. In the former case, $\prec$ is the usual ordering on $\mathcal{R}$. In the latter case, $(r_1, r_2) < (r' _1, r' _2)$ if $r_1 < r'_1$ or both $r_1 = r'_1$ and $r_2 < r'_2$. Put another way, when $\mathcal{R} = \mathbb{R}^2$, each state $(d, a)$ has a primary rank $r_1(d, a) \in \mathcal{R}$ and a secondary rank $r_2(d, a) \in \mathcal{R}$, with

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

Jobs are scheduled by their primary rank with ties broken by their secondary rank. Only when both the primary and secondary ranks are tied does the FCFS tiebreaking rule come into play.

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

When we call a rank function "monotonic" or similar, we mean that it is so with respect to age rather than descriptor.

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

### 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 $\mathcal{D} = \{\emptyset\}$, where $\emptyset$ is a "placeholder" descriptor assigned to every job. FB always serves the job of least age, so it has rank function $r(\emptyset, a) = a$. 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 multiple rank functions that encode the same SOAP policy. For instance, any rank function monotonically increasing in age, such as $r(\emptyset, a) = \langle a^2, -6a \rangle$, also describes FB.

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

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

**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 $r(x, a) = (-a, x)$: it is a nonpreemptive priority policy with size as priority.
- PSJF has rank function $r(x, a) = (x, -a)$: it is a preemptive priority policy with size as priority.
- SRPT has rank function $r(x, a) = x - a$: a job’s rank updates with age as its remaining time decreases.

#### 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 $r(d, a) = (-a, E[X_d])$: it is a nonpreemptive priority policy with expected size as priority.
- PSEPT has rank function $r(d, a) = (E[X_d], -a)$: it is a preemptive priority policy with expected size as priority.
- SERPT has rank function $r(d, a) = E[X_d]$: a job’s rank changes with age as we update the estimate of its 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 $\emptyset$ 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 is in contrast with nearly every policy described in Section 3.1, which all have monotonic rank functions. The potential nonmonotonicity of SERPT’s rank function, which occurs in examples like this one, makes previous techniques unable to analyze 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 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.

**Figure 3.1: Rank Function for Example 3.5 (SERPT)**

Example 3.6. The Gittins index of a job in state $(d,a)$ is $[3,11]$

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

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 $r(d,a) = 1/G(d,a)$. Like the policies in Example 3.5, the Gittins index policy can be defined with any set of descriptors.

The Gittins index policy is 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 $\mathcal{D} = \{\emptyset\}$, and it has rank function

$$r(\emptyset,a) = (\lfloor a \rfloor - a, a).$$

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.

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

\(^3\)It is possible to model discretized FB as an MLPS policy with infinitely many thresholds. However, discretization in SOAP is much more flexible: given any preemptive SOAP policy, its discretized variant is also SOAP.

The rank function for the Gittins Index Policy using the same distribution as in Figure 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.

**Figure 3.2: Rank Function for Example 3.6 (Gittins Index)**

- **Figure 3.3: Rank Function for Example 3.7 (Checkpoints)**
  - 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 demonstrate, SOAP policies go even further than this: they allow combining 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 a wide variety of SOAP policies. However, there are some policies which use primary rank to encode preemptibility and secondary rank to encode priority. We analyze this system in Section 6.2.

As the previous examples have demonstrated, there is an extremely powerful information of a job is its class and, if known, its size, so that 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 that jobs of size less than $x$ are nonpreemptible, have known size, and are served according to SRPT; and jobs of size at most $x$ are served according to SJF. For example, the last-come, first-serve (LCFS) policy is not SOAP for a job that arrives 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. 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.

3.3 Non-SOAP Policies

As the previous 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 preemptive last-come, first serve (PLCFS) policy could be a SOAP policy if each job’s descriptor were its arrival time, but jobs’ arrival times are not i.i.d. The earliest deadline first (EDF) policy is not SOAP for a similar reason.

Second, some policies require a tiebreaking rule other than FCFS. For example, the last-come, first-serve (LCFS) and random order of service (ROS) policies could almost be SOAP policies using the rank function in Example 3.2, but they need to break ties between jobs in state $(\emptyset, 0)$ differently. That said, a future generalization of the SOAP class could well allow for LCFS or ROS tiebreaking rules, because, like FCFS, both break ties nonpreemptively. In contrast, the processor sharing (PS) policy, which is also not a SOAP policy, requires a fundamentally different tiebreaking rule, making it difficult to handle even in a modified SOAP framework.

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 only looks at one job at a time, so there is no way to capture such context.

4 HOW TO HANDLE ANY RANK FUNCTION

We have seen how by careful choice of rank function we can express a vast space of policies, both old and new, as SOAP policies. 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.

In more general terms: because jobs’ ranks change with age, the tagged job’s priority 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 Nonmonotonicity Difficulties

There are two major obstacles to analyzing policies with arbitrary nonmonotonic rank functions. We illustrate these obstacles below.

---

1 Due to the FCFS tiebreaking rule, the nonconstant secondary rank in Example 3.4 could be made constant while still encoding PSJF.
Two difficulties arise when attempting to apply the "transformer glasses" approach to a SOAP policy with a nonmonotonic rank function. First (top), suppose a tagged job \( J \) has rank \( r_J \) upon entering the system, and suppose that \( J \) is already in the system in state \((d, a_K)\). The cyan curve shows \( K \)'s rank as a function of its age. Consider how \( J \) views \( K \) if, as pictured, \( r_J < r(d, a_K) \). Then \( J \) outranks \( K \), so \( K \) is initially hidden. However, \( K \) may be only temporarily hidden, because \( J \)'s rank may later increase to \( r'_J < r(d, a) \). Second (bottom), consider the same jobs \( J \) and \( K \) and suppose \( J \) has not entered the system yet. Whether or not \( J \)'s rank will "peak" at \( r(d, a) \) depends on \( K \)'s age when \( J \) arrives. For instance, as \( K \) advances in age from \( a_K \) to \( a'_K \) and later \( a''_K \), it switches back and forth between being visible to and hidden from \( J \).

**Figure 4.1: Difficulties with Nonmonotonic Rank Functions**

and in Figure 4.1 by comparison with SRPT. The first obstacle concerns the nonmonotonicity of the tagged job’s rank.

- In SRPT, we permanently hide other jobs based on the tagged job’s current rank.
- In general, other jobs might be only temporarily hidden. If the tagged job’s rank increases enough, a job that was initially hidden might be made visible later.

The second obstacle concerns the nonmonotonicity of the ranks of "old" jobs, namely those that are already in the system when the tagged job arrives.

- In SRPT, any old job permanently outranks the tagged job if served for long enough before the tagged job arrives.
- In general, an old job might only temporarily outrank the tagged job. Furthermore, if the old job’s rank oscillates above and below the tagged job’s initial rank, whether the old job gets hidden depends on when the tagged job arrives.

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 4.2) and Vacation Transformation (Section 4.3).

### 4.2 The Pessimism Principle

Consider a tagged job \( J \) with descriptor \( d \) and size \( x \) in a system using an arbitrary SOAP policy. Throughout, we call jobs old if they arrived before \( J \) and new if they arrive after \( J \). Given another job \( K \), the amount of time \( K \) is served before \( J \) completes is the delay of \( J \) due to \( K \). The response time of \( J \) is its size \( x \) plus its delays due to all the other jobs that are in the system with \( J \) at some point.

Suppose that another job \( K \), new or old, is in the system when \( J \) has age \( a \). To analyze \( J \)'s response time, we have to know its delay due to \( K \). As we saw in Section 4.1, deciding whether or not to hide \( K \) based only on \( J \)'s current rank \( r(d, a) \) will not work. Instead, we need to examine \( J \)'s current and future ranks.

**Definition 4.1.** The worst future rank of a job of size \( x \) in state \((d, a)\) is

\[
r_{d,x}^{\text{worst}}(a) = \sup_{a \leq b < x} r(d, b).
\]

See Figure 4.2 for an illustration.

At some point in the future, \( J \)'s rank will "peak" at \( r_J = r_{d,x}^{\text{worst}}(a) \), and \( J \) will remain at rank \( r_J \) until it is next served. Specifically, let \( r \) be the latest system time, not to be confused with age, at which \( J \) has rank \( r_J \). This means that \( J \) is served at \( t \), seeing as its age must change for its rank to decrease. In particular, this means \( J \) outranks \( K \) at \( t \), so \( K \) either completes or reaches a rank \( r_J \) before \( t \). After \( t \), we can safely hide \( K \) if it is still in the system, because \( J \) will never reach rank \( r_J \) again.

In fact, we can make an even stronger statement. Suppose a new job \( L \) arrives to the system before time \( t \) but after job \( K \), which, recall, arrived when \( J \) had age \( a \). Then \( L \) must also either complete or reach rank \( r_J \) by \( t \). We already know by the previous paragraph that \( L \) must complete or reach rank \( r_{d,x}^{\text{worst}}(b) \), where \( b \geq a \) is the age of \( J \) when \( L \) arrives. However, we know \( J \) has rank \( r_J = r_{d,x}^{\text{worst}}(a) \) at \( t \), and \( J \)'s age at \( t \) is at least \( b \), so \( r_{d,x}^{\text{worst}}(b) = r_J \).

Of particular interest is the case where \( L \) is in the virtual busy period started by \( K \), which is the set of jobs that recursively includes

- job \( K \) itself and
- any new job that arrives while another job from the virtual busy period is in service.

If we arrange the jobs of a busy period in a tree where a job’s children are the jobs that arrived during its service, then \( K \)'s virtual busy period consists of all jobs in the subtree rooted at \( K \).

\( ^5 \)Reaching \( \sup \) rank \( r \) roughly means having rank at least \( r \). See the Pessimism Principle and Appendix B for details.
We now have all we need to state the Pessimism Principle, so named for its pessimistic focus on the worst future rank.

**Pessimism Principle.** Suppose job \( J \) with descriptor \( d \) and size \( x \) has age \( a \) when it is first in the system at the same time as another job \( K \). Then before \( J \) completes, all jobs in \( K \)'s virtual busy period which arrive before \( J \)'s completion are served until they complete or reach rank \( r_J = r_{d,x}^{\text{worst}}(a) \), which specifically means the following.

- If \( K \) is old, which means that \( a = 0 \), then \( K \) is served until it completes or has first rank \( r_K > r_J \). In particular, if \( r_K = r_J \), then \( K \) outranks \( J \) due to the FCFS tiebreaking rule, thus the strict inequality\(^5\).
- If \( K \) is new, then \( K \) is served until it completes or first has rank \( r_K \geq r_J \).
- Other jobs \( L \) in the virtual busy period are all new and are served until they complete or first have rank \( r_L \geq r_J \).

To clarify, we are discussing the total amount of service another job receives while \( J \) is still in the system, but we are not making any claims about when that service occurs. In particular, the service need not be contiguous but might instead be interleaved with that of \( J \) and other jobs.

The Pessimism Principle prompts a natural question: how much time does it take for \( K \) and the new jobs in its virtual busy period to complete or reach rank \( r_{d,x}^{\text{worst}}(a) \)? We can already answer for the new jobs, which include \( K \) in the case where \( K \) arrives after \( J \), so we discuss them now. We consider old jobs in Section 4.3.

**Definition 4.2.** Given rank \( r \in \mathbb{R} \), the new \( r \)-work is a random variable, written \( X_{d,x}^{\text{new}}[r] \), representing how long a job that just arrived to the system is served until it completes or first has rank strictly greater than \( r \). Specifically, we define \( X_{d,x}^{\text{new}}[r] = X_{d,x}^{\text{new}}[r] \), where \( D \) is the random descriptor assigned to a new job and for any specific descriptor \( d \in D \),

\[
X_{d,x}^{\text{new}}[r] = \inf\{a \geq 0 \mid r(d,a) \geq r\}
\]

That is, \( c_d \) is the “cutoff age” at which new jobs reach rank \( r \). See Figure 4.3 for an illustration.

\(^{5}\text{See Appendix B for discussion of corner cases where we need } \geq \text{ instead of } >.\)

The Pessimism Principle implies that the delay of \( J \) due to a new job with random descriptor that arrives when \( J \) has age \( a \) is \( X_{d,x}^{\text{new}}[r_{d,x}^{\text{worst}}(a)] \).

**Example 4.3.** In PSJF, a job’s descriptor is its size, so \( J \)'s descriptor is \( d = x \). Furthermore, \( r_{d,x}^{\text{worst}}(a) = (x,-a) \), and

\[
X_{d,x}^{\text{new}}[(y,-a)] = \begin{cases} 
0 & \text{if } y \leq x \\
(14-a) & \text{if } y > x.
\end{cases}
\]

Thus, the delay due to new jobs under PSJF is their full size if they are shorter than \( J \) and 0 otherwise. This corresponds to our intuition from Section 4.1: new jobs of size at least \( x \) are hidden. The SRPT story is similar, but we compare \( y \) to \( x - a \) instead of \( x \).

**Example 4.4.** Consider a system using SERPT (Example 3.5) in which all jobs have the same descriptor \( \phi \). Suppose job sizes have a two-point distribution \( \text{Two}[2,14] \), meaning jobs are size \( 2 \) with probability \( 1/2 \) and size \( 14 \) otherwise. The expected remaining size of a job with age \( a \) is

\[
E[X_{\phi}(a)] = \begin{cases} 
8-a & \text{if } a < 2 \\
14-a & \text{if } a \geq 2.
\end{cases}
\]

The rank function is \( r(\phi, a) = E[X_{\phi}(a)] \), as shown in Figure 3.1, which means

\[
X_{\text{new}}^{\text{new}}[y] = \begin{cases} 
0 & \text{if } y \leq 8 \\
2 & \text{if } 8 < y \leq 12 \\
\text{Two}[2,14] & \text{if } y > 12.
\end{cases}
\]

That is, new jobs are not served at all for low rank cutoffs, new jobs are always served until age \( 2 \) for medium rank cutoffs, and new jobs are served to completion for high rank cutoffs. This third case is not relevant for this particular system since no job ever has rank greater than \( 12 \), but it could be relevant in a system using SERPT with multiple descriptors.

### 4.3 The Vacation Transformation

We continue our running discussion from the previous section, following tagged job \( J \) with descriptor \( d \), size \( x \), and current age \( a \). Our focus in this section is old jobs, for which \( a = 0 \) is the most relevant age, so we write \( r_J = r_{d,x}^{\text{worst}}(0) \).

We have seen how the Pessimism Principle shows us how long each new job delays \( J \), namely \( X_{d,x}^{\text{new}}[r_{d,x}^{\text{worst}}(a)] \). 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.

The Pessimism Principle focuses exclusively on events that occur while \( J \) is in the system. In contrast, the purpose of the Vacation Transformation is to find the delay due to old jobs, which is determined entirely by events that occur before \( J \) enters the system. Thus, throughout this section we often imagine \( J \) as a witness to the system, watching other jobs enter, receive service, and exit, all followed by a new job...
prior to \( J \)'s own arrival. Our goal is to understand the stationary behavior of the system as observed by witness \( J \). Because Poisson arrivals see time averages [27], this stationary behavior helps us understand \( J \)'s delay due to old jobs.

So far, we have been considering old jobs as a monolithic category, but it is useful to consider three subcategories. Imagine our witness job \( J \) watching the system before it arrives. At any point in time, it sees three types of old jobs.

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

More generally, we occasionally talk about a job being discarded, original, or recycled with respect to rank \( r \), in which case we replace \( r_J \) with \( r \) in the above descriptions.

An old job \( K \) goes through the following transitions between these categories, as shown in Figures 4.4 and 4.5.

- When \( K \) arrives in the system, if its initial rank is at most \( r_J \), then it starts out original, and otherwise it starts out discarded.
- As \( K \) ages, it may become discarded if it is not already.
- As \( K \) 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 shortly see, \( J \) sees at most one recycled job in the system when it arrives.

For the purposes of analyzing \( J \)'s response time, we can view the system through "transformer glasses" [12], through which only original and recycled jobs are visible, as illustrated in Figure 4.4. In the transformed system, arrivals of original jobs are caused by arrivals to the original system, but arrivals of recycled jobs are really caused by a discarded job being served in the original system.

A "transformed busy period" is a time interval during which the server is always serving original or recycled jobs. Transformed busy periods always start with the arrival of an original or recycled job in the transformed system. Arrivals of original jobs then continue, but no more recycled jobs arrive for the rest of the transformed busy period. All discarded jobs have rank greater than \( r_J \), which is in particular greater than the rank of all original and recycled jobs. This means no discarded jobs are served during a transformed busy period, so no job can transition from discarded to recycled.

To figure out \( J \)'s total delay due to old jobs, we need to know how long each other job spends as an original or recycled job. Doing this requires identifying the ages during which an old job is original or recycled.

**Definition 4.5.** Let \( r \in \mathcal{R} \) be a rank. Though we will build up to a definition that depends only on \( r \), we need some intermediate definitions that concern a descriptor \( d \in \mathcal{D} \). The 0-old \( r \)-interval for descriptor \( d \) is the interval of ages \([b_{d,0}, c_{d,0}]\) during which it is original with respect to \( r \), as shown in Figure 4.5. Specifically,

\[
b_{d,0} = \inf \{a > b_{d,0} \mid r(d, a) > r\}.
\]

For \( i \geq 1 \), the \( i \)-old \( r \)-interval is the interval of ages \([b_{d,i}, c_{d,i}]\) during which it is recycled with respect to \( r \) for the \( i \)th time, as shown in Figure 4.5. Specifically,

\[
b_{d,i} = \inf \{a > b_{d,i-1} \mid r(d, a) \leq r\},
\]

\[
c_{d,i} = \inf \{a > b_{d,i} \mid r(d, a) > r\}.
\]

If \( b_{d,i} = c_{d,i} = \infty \), we interpret \([b_{d,i}, c_{d,i}]\) as the empty set.

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

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

If \( b_{d,i} = c_{d,i} = \infty \), we define \( X^\text{old}_d[r] = 0 \).

For both \( r \)-intervals and \( r \)-work, we sometimes use original as a synonym for 0-old or i-recycled as a synonym for \( i \)-old for \( i \geq 1 \).

From witness \( J \)'s transformed perspective, \( X^\text{old}_J[r_J] \) is the total amount of time an old job \( K \) is served as an original job, and \( X^\text{old}_i[r_J] \) for \( i \geq 1 \) is the amount of time \( K \) is served as a job that is
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 \([b_{d,i}, c_{d,i}]\) during which a job of descriptor \( d \) is original \((i = 0)\) or recycled for the \( i \)-th time \((i \geq 1)\). The \( i \)-old \( r \)-intervals are highlighted green. The \( i \)-old \( r \)-work is the amount of service the job requires while its age is in its \( i \)-old \( r \)-interval.

**Figure 4.5: Illustration of Definition 4.5 (Old Work)**

being recycled for the \( i \)-th time. Note that \( X_0^{\text{old}}[r_j] \) may be 0, representing \( K \) starting out discarded, and \( X_i^{\text{old}}[r_j] \) for \( i \geq 1 \) may be 0, representing \( K \) completing before being recycled for the \( i \)-th time.

The Vacation Transformation follows from three observations.

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

**Vacation Transformation.** Consider a job \( J \) with descriptor \( d \) and size \( x \) arriving to the system, and let \( r_{d,x} \) be \( r_{d,x}^{\text{new}}(0) \). The total delay experienced by \( J \) due to old jobs is the queuing time in a transformed M/G/1/FCFS system with server vacations.

- Jobs with size distribution \( X_0^{\text{old}}[r_j] \) have Poisson arrivals at rate \( \lambda \) and are served in FCFS order. These correspond to original jobs.
- The server takes vacations of length distribution \( X_i^{\text{old}}[r_j] \) for every \( i \geq 1 \) at average rate \( \lambda \). These vacations, called \( i \)-recycled vacations, correspond to jobs becoming recycled. The vacations’ starting times may depend arbitrarily on the system’s history and are not a Poisson process. In particular, vacations can start only when the system is neither busy nor already on vacation.

We call the transformed system the **Vacation Transformation System**, or simply VT System.

It may seem unusual that \( i \)-recycled vacations occur with average rate \( \lambda \) in the VT System, seeing as not all jobs will become recycled \( i \) times. This is accounted for by the fact that \( X_i^{\text{old}}[r_j] \) might be 0. In this context, “transform” means Laplace-Stieltjes transform. The transform of a random variable \( V \) is written \( \tilde{V}(s) \), and the transform of a parametrized random variable \( V[r] \) is written \( \tilde{V}(s)[r] \).

We now analyze the response time of SOAP policies. Specifically, we analyze \( \tilde{T}_{d,x} \), the response time of a tagged job with descriptor \( d \) and size \( x \). The tagged job’s response time is the sum of two parts:

- **job-induced work**, written \( J_{d,x} \), the time spent serving jobs in the virtual busy period started by the tagged job; and
- **queue-induced work**, written \( Q_{d,x} \), the time spent serving jobs in the virtual busy period started by jobs already in the system when the tagged job arrives.

Queue-induced and job-induced work are independent because the job arrival process is Poisson, so \( \tilde{T}_{d,x} = \tilde{J}_{d,x}(s) + \tilde{Q}_{d,x}(s) \).

In both analyses, we make use of the **new \( r \)**-work *busy period*, denoted \( \tilde{B}^{\text{new}}[r] \), which is the length of a busy period in an M/G/1 system with arrival rate \( \lambda \) and job size \( X^{\text{new}}[r] \). Its transform satisfies the functional equation [12]

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

More generally, the new \( r \)-work busy period started by work \( W \), written \( \tilde{B}^{\text{new}}[s][r] \), 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]

\[
\tilde{B}^{\text{new}}[s][r](W) = \tilde{W}(1 - \tilde{B}^{\text{new}}[s][r](W)).
\]

**Lemma 5.1.** Under any SOAP policy, the Laplace-Stieltjes transform of job-induced work for a job with descriptor \( d \) and size \( x \) is

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

**Proof.** Our general approach is to view job-induced work as a sum of many small virtual busy periods started by a small amount of work \( \delta \) and take the \( \delta \to 0 \) limit, which exists thanks to conditions in Appendix A. Specifically, we divide \([0, x]\) into chunks of size \( \delta \) and consider each virtual busy period started by the work needed to bring the job from age \( a \) to age \( a + \delta \). In the limit, we can assume the job has rank \( r(d,a) \) throughout this virtual busy period.

By the Pessimism Principle, the amount of work in each small virtual busy period is the length of a new \( r \)-work busy period started by \( \delta \), which by (5.1) has transform

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

Furthermore, the lengths of the small virtual busy periods are independent because the arrival process is Poisson. Taking the product of (5.2) over ages \( a \) that start a chunk and taking the \( \delta \to 0 \) limit yields the desired expression. \(\square\)
To discuss queue-induced work, it is useful to define two new notations. First, let
\[ \rho_{\text{new}}^\text{new} [r] = \lambda E[X_{\text{new}}^\text{new} [r]] \]
\[ \rho_{\text{old}}^\text{old} [r] = \lambda E[X_{\text{old}}^\text{old} [r]] \]
\[ \rho_{\text{new}}^\text{old} [r] = \sum_{i=0}^{\infty} \rho_{\text{old}}^\text{old} [r] \]
be the loads contributed by new r-work, i-old r-work, and all old r-work, respectively. Second, let \( Y_{i}^\text{old} [r] \) be the equilibrium distribution [12], or length-biased sample, of \( X_{i}^\text{old} [r] \). It has transform
\[ Y_{i}^\text{old} [r] (s) = \frac{1 - Y_{i}^\text{old} [r] (s)}{\lambda E[Y_{i}^\text{old} [r]]}. \]

**Lemma 5.2.** Under any SOAP policy, the Laplace-Stieltjes transform of queue-induced work for a job with descriptor \( d \) and size \( x \) is
\[ \tilde{Q}_{d,x}(s) = \frac{1 - \rho_{\text{old}}^\text{old} [r] + \sum_{i=0}^{\infty} \rho_{\text{old}}^\text{old} [r] Y_{i}^\text{old} [r] (s)}{1 - \rho_{\text{old}}^\text{old} [r] Y_{0}^\text{old} [r] (s)} \]
where \( r = r_{d,x}^\text{worst} (0) \) and \( \sigma = \lambda (1 - \rho_{\text{new}}^\text{new} [r] (s)). \)

**Proof.** By the Pessimism Principle, the queue-induced work is the length of an M/G/1 busy period started by a random amount of work \( W \), namely the amount the job is delayed by other jobs that arrived before it, with arrivals at rate \( \lambda \) of size \( X_{\text{new}}^\text{new} [r] \). This means \( \tilde{Q}_{d,x}(s) = \tilde{W}(s) \) by (5.1), so all that remains is to find \( \tilde{W}(s) \).

The Vacation Transformation states that \( W \) has the same distribution as the queuing time in the VT System, which is a particular M/G/1/FCFS system with vacations. A decomposition result of Fuhrmann and Cooper [10] 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 \( A_{V} \) of jobs that an arrival to a non-busy server observes before the server becomes busy again.

Recalling the Vacation Transformation and a standard result for the M/G/1 queue [12], we immediately obtain the probability generating function for \( N_{Q} \).
\[ \tilde{N}_{Q}(z) = \frac{1 - \rho_{\text{old}}^\text{old} [r]}{1 - \rho_{\text{old}}^\text{old} [r] Y_{0}^\text{old} [r] (\lambda (1 - z))}. \]

In the VT System, \( A_{V} \) is either 0, if the system is idle, or the number of arrivals at rate \( \lambda \) during time \( Y_{i}^\text{old} [r] \), if the system is in the middle of an i-recycled vacation. The probability of a job arriving to an idle system is \( 1 - \rho_{\text{old}}^\text{old} [r] \), and the probability of a job arriving to an i-recycled vacation is \( \rho_{\text{old}}^\text{old} [r] \). Accounting for the for the fact that we measure \( A_{V} \) only when a job arrives to a non-busy server, which happens with probability \( 1 - \rho_{\text{old}}^\text{old} [r] \), we find that the probability generating function of \( A_{V} \) is
\[ \tilde{A}_{V}(z) = \frac{1 - \rho_{\text{old}}^\text{old} [r] + \sum_{i=0}^{\infty} \rho_{\text{old}}^\text{old} [r] Y_{i}^\text{old} [r]\lambda (1 - z))}{1 - \rho_{\text{old}}^\text{old} [r]}. \]

Multiplying (5.3) and (5.4), applying the distributional version of Little’s Law [14], and substituting \( \sigma = \lambda (1 - z) \) gives the transform of queueing time in the VT System, which matches the desired transform \( \tilde{W}(\sigma) \).

**Theorem 5.3 (SOAP Transform of Response Time).** Under any SOAP policy, the Laplace-Stieltjes transform of response time for a job with descriptor \( d \) and size \( x \) is
\[ \tilde{T}_{d,x}(s) = \tilde{J}_{d,x}(s) \tilde{Q}_{d,x}(s), \]
where \( \tilde{J}_{d,x}(s) \) and \( \tilde{Q}_{d,x}(s) \) are as in Lemmas 5.1 and 5.2, respectively.

**Proof.** This follows immediately from Lemmas 5.1 and 5.2 and the independence of \( J_{d,x} \) and \( Q_{d,x} \).

**Theorem 5.4 (SOAP Mean Response Time).** Under any SOAP policy, the mean response time for a job with descriptor \( d \) and size \( x \) is
\[ E[T_{d,x}] = \int_{0}^{\infty} \frac{1}{1 - \rho_{\text{new}}^\text{new} [r] (a)} da + \frac{\lambda \sum_{i=0}^{\infty} E(X_{i}^\text{old} [r])^{2}}{2(1 - \rho_{\text{old}}^\text{old} [r] (1 - \rho_{\text{new}}^\text{new} [r])), \]
where \( r = r_{d,x}^\text{worst} (a) \) and \( r = r_{d,x}^\text{worst} (0) \).

**Proof.** This follows from \( -E[T_{d,x}] = \tilde{T}_{d,x}(0) + \tilde{Q}_{d,x}(0) \) after straightforward computation.

## 6 NEW ANALYSES FOR SPECIFIC POLICIES

In this section, we analyze the response time of several policies discussed in Section 3.2. Though we focus on obtaining expressions for mean response time, essentially the same work is enough to obtain the Laplace-Stieltjes transform of response time.

The main challenge to analyzing SOAP policies is determining \( X_{\text{new}}^\text{new} [r] \) and \( X_{i}^\text{old} [r] \). Throughout, we give expressions for these focusing only on ranks \( r \) that are important for the final result. In particular, we only need to find i-old r-work with \( r = r_{d,x}^\text{worst} (0) \) for the possible descriptors \( d \) and sizes \( x \).

### 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
\[ r(\emptyset, a) = (\lfloor a \rfloor - a, a), \]
shown in Figure 3.3. More generally, we might consider the \( \Delta\)-
discretized FB policy, where now the time between checkpoints is \( \Delta \) instead of 1. Its rank function and analysis are the same as those of 1-discretized FB, except we replace every instance of \( [a] \) with \( [a]_{\Delta} = [a] / \Delta \), and do similarly for \( \lfloor a \rfloor \), so we discuss 1-
discretized FB for simplicity. Just like traditional FB, it is convenient to work in terms of the truncated distribution and its associated load [12],
\[ X_{\Delta} = \min(X, x) \]
\[ \rho_{\Delta} = \lambda E[X_{\Delta}] \]

The new r-work for discretized FB is
\[ X_{\text{new}}^\text{new} ([\lfloor a \rfloor - a, a]) = \begin{cases} X_{\Delta} & \text{if } a \text{ is an integer} \\ 0 & \text{otherwise} \end{cases} \]

The 0-old r-work is similar but with \( X_{\Delta} \) in place of \( X_{\Delta} \) thanks to the FCFS tiebreaking rule. A job’s maximal rank is
\[ r_{d,x}^\text{worst} (0) = (\lfloor 0, \lfloor x \rfloor \rfloor) \].
and $X_i^{\text{old}}((0, a)] = 0$ for $i \geq 1$. Applying Theorem 5.4 yields the following.

**Proposition 6.1.** Under $\Delta$-discretized FB, the mean response time for a job with size $x$ is \( g \)

$$E[T_{x}] = x - [x]_\Delta + \frac{1}{1 - \rho_{x}/\lambda} \left( \frac{\lambda E[X^2_{\text{new}}]}{\lambda} \right).$$

In the $\Delta \to 0$ limit, we recover the well-known mean response time of traditional FB.

### 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: only some jobs’ exact sizes are known, and only some jobs are preemptible. Let

- $X_H$ and $X_R$ be the respective size distributions of humans and robots,
- $\rho_H$ and $\rho_R$ be the respective probabilities that a given arrival is a human or a robot, and
- $\lambda_H = \lambda R_H$ and $\lambda_R = \lambda R_R$ 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 $x$.

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

- $X_i^{\text{new}}((0, x)] = \begin{cases} x_H & \text{with probability } p_H \\ x_R & \text{with probability } p_R \end{cases}$
- $X_i^{\text{old}}((0, x)] = \begin{cases} x_H & \text{with probability } p_H \\ x_R & \text{with probability } p_R \end{cases}$
- $X_i^{\text{old}}((0, x)] = \begin{cases} x_H & \text{with probability } p_H \\ x_R & \text{with probability } p_R \end{cases}$

where $\mathbb{I}$ is the indicator function. Finally, $X_i^{\text{old}}[r] = 0$ for $i \geq 2$, yielding the following by Theorem 5.4.

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

$$E[T_{H}] = X_H + \frac{\lambda_H E[X_H^2] + \lambda_R E[X_R^2]}{2(1 - \rho_{<X_H}) (1 - \rho_H - \rho_{\leq X_H})}$$

and the mean response time of robots with size $x$ is

$$E[T_{R,x}] = \int_0^x \frac{1}{1 - \rho_{<X_H}} \frac{d \rho}{x} - \rho_{<R} - \rho_{R \leq X_H}$$

where

- $\rho_H = \lambda_H E[X_H]$,
- $\rho_{R \leq X_H} = \lambda_R E[X_R] I(X_R \leq x)$,
- $\rho_{R \leq X_H} = \lambda_R E[X_R] I(X_R \leq x)$.

See Appendix B for discussion of why we use $[x]_\Delta$ instead of $[x]_\Delta + \Delta$.

### 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 5.3 and 5.4 give us exactly the framework we need to analyze the Gittins index policy used with any set of descriptors and job size distributions. 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 $\lambda_A = \lambda_R A$ and $\lambda_B = \lambda_R B$ and respective Pareto size distributions $X_A$ and $X_B$. Specifically,

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

and symmetrically for $B$. The rank is strictly increasing in $a$, so $r_{\text{worst}}(a) = r(d, x)$ for all ages $a < x$. Furthermore, once a job becomes discarded with respect to some rank, it is never recycled, so $X_i^{\text{old}}[r] = 0$ for all $i \geq 1$.

It remains only to compute $X_i^{\text{new}}[r]$ and $X_i^{\text{old}}[r]$. Let

$$y(A, r) = \max(\alpha_A r - \beta_A, 0)$$

be the size of class-$A$ job whose worst future rank is $r$, and symmetrically for $B$. Note that $y(A, r, x) = x$. Because the rank function is increasing and Pareto distributions are continuous,

$$X_i^{\text{new}}[r] = X_i^{\text{old}}[r] = \begin{cases} (X_A)_{\beta(A, r)} & \text{with probability } p_A \\ (X_B)_{\beta(B, r)} & \text{with probability } p_B, \end{cases}$$

where $(X_A)_{\beta(A, r)}$ and $(X_B)_{\beta(B, r)}$ are truncated distributions, as defined in Section 6.2. By Theorem 5.4,

$$E[T_{A,x}] = \frac{1}{1 - \rho_x} \left( x + \frac{\lambda_A E[X_A^2] + \lambda_R E[X_R^2]}{2(1 - \rho_x)} \right)$$

where

- $y_B = y(B, r(A, x))$,
- $\rho_x = \lambda_A E[X_A x] + \lambda_R E[X_R x]$.

Of course, $E[T_{B,x}]$ 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 example size distribution introduced in Example 3.5. Both policies have nonmonotonic rank functions in this case, so we need the full power of Theorem 5.4 to complete mean response times.

We consider a system in which all jobs have descriptor $\emptyset$ and the two-point size distribution $Two(2, 14)$, meaning they have size $2$ with probability $1/2$ and size $14$ otherwise. We computed $r(\emptyset, a)$ and $X_i^{\text{new}}[r]$ for SERPT in Example 4.4. We only need to compute
The rank function for SERPT with job size distribution $Two \{2, 14\}$, originally shown in Figure 3.1. The original and 1-recycled 8-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, a job has remaining size 8, so it becomes 1-recycled.

**Figure 6.1: Original and Recycled Work in SERPT**

The rank function for SERPT with job size distribution $Two \{2, 14\}$, originally shown in Figure 3.1. The original and 1-recycled 8-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, a job has remaining size 8, so it becomes 1-recycled.

The rank function for SERPT with job size distribution $Two \{2, 14\}$, originally shown in Figure 3.1. The original and 1-recycled 8-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, a job has remaining size 8, so it becomes 1-recycled.

**Figure 6.1: Original and Recycled Work in SERPT**

The rank function for SERPT with job size distribution $Two \{2, 14\}$, originally shown in Figure 3.1. The original and 1-recycled 8-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, a job has remaining size 8, so it becomes 1-recycled.

$\text{(a)}$ The rank function plot in Figure 6.1, where we see that

$X_0 \left( \begin{array}{c} \text{old} \\ \text{new} \end{array} \right) = 2$

$X_0 \left( \begin{array}{c} \text{old} \\ \text{new} \end{array} \right) = 12$

$X_1 \left( \begin{array}{c} \text{old} \\ \text{new} \end{array} \right) = 2$

$X_1 \left( \begin{array}{c} \text{old} \\ \text{new} \end{array} \right) = 0$

Furthermore, $X_i \left( \begin{array}{c} \text{old} \\ \text{new} \end{array} \right) = 0$ for $i \geq 2$ and $y \geq 8$. Applying Theorem 5.4 yields size-specific mean response times

$E[T_{\text{SERPT}}^2] = 2 + \frac{18\lambda}{1 - 2\lambda}$

$E[T_{\text{SERPT}}^4] = 8 + \frac{6\lambda}{1 - 2\lambda} + \frac{50\lambda}{(1 - 2\lambda)(1 - 8\lambda)}.$

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

$G(\emptyset, a) = \begin{cases} 4 - 2a & \text{if } a < 2 \\ 14 - a & \text{if } a \geq 2. \end{cases}$

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 similar analysis of new and 1-old $r$-work and simply state the final result:

$E[T_{\text{Gittins}}^2] = 2 + \frac{6\lambda}{1 - 2\lambda}$

$E[T_{\text{Gittins}}^4] = 4 + \frac{10\lambda}{1 - 2\lambda} + \frac{50\lambda}{(1 - 2\lambda)(1 - 8\lambda)}.$

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:

$E[T_{\text{SERPT}}^2] - E[T_{\text{Gittins}}^2] = \frac{12\lambda}{1 - 2\lambda}$

$E[T_{\text{SERPT}}^4] - E[T_{\text{Gittins}}^4] = \frac{-8\lambda}{1 - 2\lambda}.$

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 job’s state 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 Vacation Transformation, we overcome the obstacles presented by nonmonotonic rank functions to present a universal response time analysis that works for 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.

**REFERENCES**


A 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 \( \mathbb{R}_{\leq 0} \) must contain only finitely many boundary points between pieces.

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 \( R = \mathbb{R}^2 \) ordered lexicographically, the limiting policy is Algorithm A.1, which has a clear generalization to \( R = \mathbb{R}^n \).

Algorithm A.1 SOAP Policy in Continuous Time

Let \( J \) be the set of jobs \((d,a)\) of minimal \( r_1(d,a) \).

- Within \( J \), consider jobs such that \( r \) is strictly decreasing in age. If there are any, schedule the job of minimal \( r_2 \), using the FCFS tiebreaking rule if there are multiple such jobs.

- Otherwise, within \( J \), 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 each job \((d,a)\) share proportional to \( 1/\partial_d r_2(d,a) \).

- Otherwise, \( J \) must only contain jobs such that \( r_1 \) is strictly increasing. Share the processor between jobs in \( J \), giving each job \((d,a)\) share proportional to \( 1/\partial_d r_1(d,a) \).

B WORST FUTURE RANK DETAILS

For simplicity of exposition, throughout Section 4, we assumed that a job’s worst future rank \( \mathcal{r}_{d,x}^{\text{worst}}(a) \) 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 \((0,x)\).

There are multiple ways to remedy the situation. The most intuitive is to say that \( \mathcal{r}_{d,x}^{\text{worst}}(a) \) is not a rank but a rank bound. The set of rank bounds is \( R \times (−1,0) \) 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

\[
\mathcal{r}_{d,x}^{\text{worst}}(a) = (\sup (r(d,b), −I(\text{the supremum is not attained}))_{a \leq b < x}
\]

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 comparing a rank \( r \) against a rank bound, we interpret the rank as the rank bound \((r,0)\). This does not require changing Definition 4.2: a rank \( r' \) satisfies \((r',0) \geq (r,q)\) if and only if \( r' \geq r \). However, Definition 4.5 is affected, specifically for open rank bounds: a rank \( r' \) satisfies \((r',0) > (r,−1)\) if and only if \( r' \geq r \). That is, with an open rank bound, we replace \( > \) with \( \geq \) when defining \( c_{d,1} \) and we replace \( \leq \) with \( < \) when defining \( h_{d,1} \). Returning to our example from Section 6.1 of a job with integer size \( x \), it turns out that \( X_0 \{ r_{d,x}^{\text{worst}}(0) \} \) is actually \( X_x \), not \( X_{x+1} \). We correct for this by using \( \lceil x \rceil \) in place of \( x + 1 \).