Mean Field Behaviour of Job Redundancy Queueing Models

by

Aaron Janeiro Stone

Abstract

Technological advancement in cloud computing has resulted in the viability of a new class of routing algorithm, the so-called redundancy models which are able to replicate jobs for processing on different servers. An asymptotic amount of queue-endowed servers could in this way employ their plentiful, otherwise idle servers towards processing. Research on the behaviour of such systems, however, has only conjectured the asymptotic independence of queues (Gardner, Harchol-Balter, & Scheller-Wolf, 2016; Hellemans, Bodas, & Van Houdt, 2019). To this end, by modelling the process as a time-indexed family of hypergraphs wherein hyperedges represent cloned dependents, along with the notion of exchangeable random groups derived by Austin (2008), we are able to demonstrate the conjectured behaviour in a simulation setting.

Acknowledgements

Thank you, Dr. Steve Drekic and Dr. Tim Hellemans, for your guidance as I delved into an area once foreign to me.

Dedication

Thank you, Fatima and my family, for always being there for me in such a fast-moving world.

The whole entire world is a very narrow bridge; the main thing is to have no fear at all.

- *Nachman*

Table of Contents

List of Figures

Chapter 1

Introduction

Enamoured by physicists for its ability to turn probabilistic behaviour into matters of determinism, Mean Field Theory (MFT) also has a place in the study of queueing systems as the number of queues become asymptotic.

Definition 1 (Mean Field)**.**
Over a time-filtered probability space $(\Omega, \mathcal{F}_t, \mathcal{F}, P)$*, for* $N \in \mathbb{N}$*, a mean field describes the behaviour of any set of stochastic random variables*

$$\mathbf{X}^{(N)}(t) := \{X_i(t)\}_{i \leq N}$$

that turns deterministic in law as $N \to \infty$*, irrespective of if the finite-*N *or finite-*t *cases resulted in this set of bodies being dependent [1].*

Next, it is important to define exchangeability, a condition which provides useful properties for MFT.

Definition 2 (Exchangeability)**.** *A collection of random variables* $\mathbf{X}^{(N)}$ *is exchangeable if for any* N*-permutation* $\gamma_N \in \Gamma_N$*,*

$$\mathit{Law}(\gamma_N \mathbf{X}^{(N)}) = \mathit{Law}(\mathbf{X}^{(N)}).$$

de Finetti's Theorem states that exchangeability implies the collection to be conditionionally independent (in a Markovian sense) and identically distributed [2]. Moreover, for partition $N = \bigcup_{i \leq k} N_i$*, exchangeability over partition* $\{N_i\}_{i \leq k}$ *such that*

$$\mathit{Law}(\mathbf{X}^{(\gamma\{N_i\}_{i \leq k})}) = \mathit{Law}(\mathbf{X}^{(\{N_i\}_{i \leq k})})$$

is known as (N, Γ) *exchangeability (with* $\gamma \in \Gamma$*) [3].*

With multiprocessing being employed at its current scale in server farms, society is indeed approaching a time wherein the asymptotic behaviour of parallel queueing systems can be considered realistically. Lately, interest has been given to queueing systems which employ job redundancy in order to lower total processing time [4]. That is, routing policies which take advantage of scenarios wherein a surplus of queues and/or servers are available, replicating each job such that it might be completed faster should it happen to make its way through a less-busy queue than the original.

Definition 3 (Job Redundancy).
A scheduler, $\mathcal{D}$, follows a job redundancy policy if it systematically clones arriving jobs and removes all clones upon (or after a delay following) completion of any one clone.

Prior studies have pointed towards certain redundancy policies as being inefficient or even unrealistic due to over-relying on cloning. Take for example *Redundancy(d)*, wherein d servers are chosen per arrival; each is given a clone and upon completion of any one clone, all others are removed immediately without any cost. As such, implementing a threshold on when to clone a job becomes useful for budgeting cancellation costs. In particular, *Threshold(R,d)* has risen to prominence as a means to balance workload in queueing systems.

Definition 4 (Workload and System Load).
Workload *refers to the total amount of work remaining (in time) for a queue. In trivial cases not involving enqueued bodies potentially leaving, this would merely be the sum of individual jobs' service times. With $w_j^{(i)}(t)$ denoting the (random) service time of the jth job in queue X_i at time t, the workload of a queue would be:*

$$W_i(t) = \sum_{j \leq \# X_i(t)} w_j^{(i)}(t) \tag{1.1}$$

for counting measure $\#$ which counts the jobs waiting in a queue at some particular time.
System Load *refers to the amount of work remaining in the entire system, namely*

$$W(t) = \sum_{i \in \psi} W_i(t)$$

where $\psi \subseteq \mathbb{N}$, given that we will be considering the case of systems operating in finite time as the number of queues grows indefinitely. As such, ψ will henceforth refer to this more general case.

$$\begin{tikzcd}
\mathbf{X}^{(N)}(t) \arrow[r, "N\to\infty"] \arrow[d, "t\to\infty"] & \mathbf{X}(t) \arrow[d, "t\to\infty"] \\
\mathbf{X}^{(N)}(\infty) \arrow[r, "N\to\infty"] & \pi
\end{tikzcd}$$

Figure 1.1: Commutativity of Limits

As an example, a service time in a $G/M/c$ system will be drawn from an exponential distribution. In this simple case, the mth arriving job can be given the "marks" $(T_m, S_m) \equiv (T, S)_m$ where $S_m \overset{IID}{\sim} \text{EXP}(\lambda)$ and T_m is the time of arrival. Conditioning on the process $(T, S)_m$, $W_i(t)$ turns into a matter of merely adding up the enqueued service times and that remaining of the currently serviced job, which is conveniently memoryless.

Altogether, Figure 1.1 describes the behaviour which would be expected in an ideal system wherein both a mean field and asymptotic independence can be achieved [1]. In particular, P describes a fixed "equilibrium" point of the system, a state of the system (in terms of queue-counts) which is consistently held once reached, giving the collection a distribution of δ_P. In terms of the predictability and stability of a system, needless to say, this would be a "gold-standard"; one would be interested in their ability to achieve such a system in practice.

For the sake of brevity, the notation of $[n] := \{i \in \mathbb{N} | i \leq n\}$ will be used, along with the understanding of $\mathbf{X}^{[n]} \equiv \mathbf{X}^{(n)}$ for maximal element n. Moreover, accepting this set-index notation, $\mathbf{X}^{\psi}$ will refer to the general case of $[n]$, given we will also consider $[n] \xrightarrow{n\to\infty} \mathbb{N}$.

Definition 5 (Threshold(R, d))**.**
Threshold(R,d)*, denoted by $\mathcal{D}_{\mathit{Thresh}(R,d,Z)}$, selects d queues upon a job arrival. Next:*

1. *For $i \leq d$ queues which have workload less than or equal to R, place copies in these i queues.*

2. *If $i = 0$, place the original arrival in a queue from the d chosen at random.*

Z refers to any imposed job cancellation cost (e.g., an added temporary workload). In this paper we will concern ourselves to the cancellation cost-free case, denoted $\mathcal{D}_{\mathit{Thresh}(R,d)}$.

One important question, however, is yet to be answered. It is unknown whether or not there exists sufficient arrival rate or service rate parameters such that a mean field will be observed for particular values of R or d in the threshold model. This leads us to the following conjecture for which this paper aims to demonstrate.

Conjecture 1.
As $N \longrightarrow \infty$, the system $\mathcal{D}_{Thresh(R,d)}$ becomes (ψ, Γ)-exchangeable. Moreover, As $t \longrightarrow \infty$, the system becomes deterministic.

Chapter 2

Model Specification

In order to model such a system, we incorporate the notation most frequently seen in the study of palm calculus [5]. Most importantly, assuming we delegate a probability space $(\Omega, \mathcal{F}_t, \mathcal{F}, P)$ with the measurable flow $\{\theta_t\}$ which is P/θ_t invariant (i.e., Ergodic such that $\theta_t M = M$ for some $M \in \mathcal{F} \Rightarrow P(M) \in \{0, 1\}$), then the arrival process A can be associated with the flow. In a system with Markovian arrivals, the counting measure associated with the flow would necessarily be Poisson; other so-called counting processes wherein inter-arrivals are determined by random or even deterministic periods of time can be used. Inter-arrival times (for arrival n occurring at T_n), regardless of the generating process, shall be denoted

$$\tau_n = T_{n+1} - T_n, n \in \mathbb{N}.$$

Intensities (even if non-Poisson) will be denoted $\lambda = E(A((0, 1])))$ for arrival process A, being interpreted as the intensity of a process moving a state (i.e., counting an additional element) within unit time.

Definition 6 (Marked Process)**.** *For each job which enters the system, they can be "marked" by a series of random variables defining their behaviour within the system [5]. In general, for the systems we shall consider, we consider the marked process σ_n as being the required service of arrival n and the marks T_n as its time of arrival. The tuple*

$$(T, \sigma)_n, n \in \mathbb{N}$$

will therefore be used to mark the nth job. We shall extend this notation, however, to include sets; for a set of jobs η, wherein each element has their own arrival time,

$$(T, \sigma)_\eta = \{(T, \sigma)_n\}_{n \in \eta}.$$

In a simple $G/M/c$ queue, as discussed before, the double $(T, \sigma)_n$ would be sufficient for reducing the problem into a deterministic one. In our case, given that jobs can find their status in the system tied to the behaviour of their replicas, we must append a marking to track this phenomenon. As a matter of fact, we instead move to marking the queues as was done by [6] to prove the conjecture in the case of Join-The-Shortest-Queue(d) systems, although with an additional job dependency term.

Definition 7 (Job Dependency Graph: Finite Case)**.**
*The **Job Dependency Graph**, $G_t = (V, E)_t$, is held constant between the events of arrivals and job completions, where an edge is drawn between two nodes if and only if they are job-dependent.*

1. *Movement in a queue requires appropriate redrawing of graph.*
2. *G_t is stochastic with law in $Pr(\Omega, \mathcal{F}_t)$.*
3. *Maximal system queue size is bounded, $\sup_{n \in \chi} z_n(t) := \nu(t)$.*

"Appropriate redrawing" in this case means

1. *The graph is redrawn to reflect movement within each queue (moving due to job completions or arrivals).*
2. *G_t can depend only on $G_s, s < t$, and other current values of X_t (denote these other values by $\tilde{X}_t$) and is such that*

$$P(G_t | \tilde{X}_t, \{G_a\}_{a \in S}) = P(G_t | \tilde{X}_t, G_{\max(S)})$$

   for any set S such that $S \subset [0, t)$.

Definition 8 (Job Dependency Matrix: Finite Case)**.**
*The **Job Dependency Matrix** is an adjacency matrix (non-unique but one-to-one) for G_t. Specifically,*

$$\rho(t) \equiv \rho(G_t) = \left[\begin{array}{c|c|c|c} B_{1,1} & B_{1,2} & \dots & B_{1,\nu} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline B_{\nu,1} & B_{\nu,2} & \dots & B_{\nu,\nu} \end{array}\right] \tag{2.1}$$

is a blocked matrix such that sub matrix

$$B_{i,j} = [b_{q,m}]_{q,m \leq N} = \begin{cases} 1, & \substack{\textit{if queue } q \textit{ in row } i \textit{ is connected in } G_t \\ \textit{to queue } m \textit{ in row } j} \\ 0, & \textit{otherwise} \end{cases}$$

where connections between jobs occur if and only if the completion of one job implies the removal of all other connected jobs from the system.

While complicated in the above form, this graph merely draws an edge between jobs of separate queues while tracking their current place in their respective queues. Because at most one event can happen in infinitesimal time (i.e., an arrival or departure), the assumptions merely state that between any two events this graph shall not need to be redrawn. Extending this notion, one can iterate such a procedure indefinitely, viewing the resultant infinite graph as one where jobs are joined by hyperedges if and only if there exists an element of a set of replicas in each of the queues. In other words, one can represent queues which are connected by means of having replicas of the same jobs enqueued within them by viewing each as a node connected with a hyperedge; we will call queues related by such a hyperedge members of a *replica class* . Thus, Definition 7 can be represented more succinctly in a hypergraph form, allowing one to represent the cavity process studied in [7].

Definition 9. *(Hypergraph Representation of the Dependency Matrix) By collapsing edges as jobs and vertices as servers, all connected subgraphs of G_t can be thought of as incident nodes with edges $i_1, \ldots i_j$ for $j < d$, representing connected jobs. This gives us now only as many nodes as there are currently servers. Such a representation for G_t will be denoted by $\mathcal{G}_t$. See Fig 2.1 for a visual representation.*

When one creates the infinite graph iteratively, it is meant that one could view the graph described in Definition 7 as an embedding into a larger graph by merely considering more queues or a larger maximal queue size at any finite time t, corresponding to the adding of an additional column or row of vertices, respectively. Building a metric on an infinite graph space simplifies the matter of quantifying convergence in terms of N significantly because all possible finite embeddings can be expressed in the same space as $N \to \infty$. Thus, let us consider

$$\mathbb{E} := \{\text{locally finite graphs}\}.$$

Now, extending the notation of [6], we can fully describe the marked process for queues with general service times.

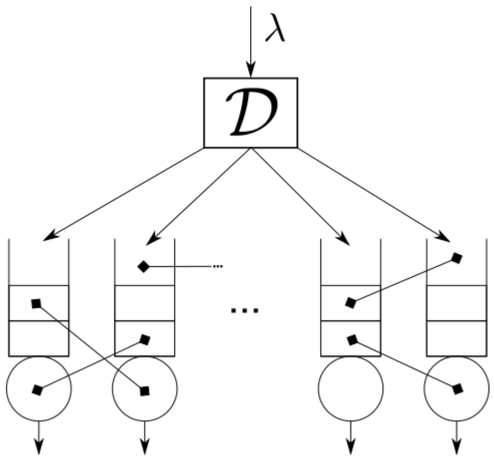

Figure 2.1: A graphical representation of queues with replicated jobs as hyperedges.

Definition 10 (State Space With Redundancy).
With ψ as $[n]$ or $\mathbb{N}$, $\rho(\cdot)$ denoting the matrix representation of a graph, and $\mathbf{X}^{\psi}$ denoting a system of ψ parallel queues, let us assign $X_n(t)$ to the nth queue of the ψ-indexed possible queues such that $X_n(t)$ follows the nth marginal distribution of $\mathbf{X}^{\psi}$. Adapting the notation of [6], we assign state spaces to X_n and $\mathbf{X}^{\psi}$ as follows:

$$X_n(t) \in \left(\mathbb{N}, \left(\mathbb{R}^+\right)^3, \rho(\mathbb{E})\right) := \mathcal{E}_n^{(\psi)}$$
$$\mathbf{X}(t) \in \{\mathcal{E}_n^{(\psi)}\}_{n\in\psi} := \mathcal{E}^{(\psi)}$$

Such that $X_n = (z_n(t), w_n(t), l_n(t), v_n(t), A(t))$ where

1. *$z_n(t) \in \mathbb{N}$ denotes queue size*
2. *$w_n(t) \in \mathbb{R}^+$ denotes workload*
3. *$\ell_n(t) \in \mathbb{R}^+$ denotes the amount of service time spent on job currently in server*
4. *$v_n(t) \in \mathbb{R}^+$ denotes the time remaining for job in server*
5. *$A \in \rho(\mathbb{E})$ is a representation of the graph*

Chapter 3

Simulation Study

3.1 ParallelQueue Package

In order to generate and study parallel queueing processes, few trivial options currently exist. Moreover, while there exist some discrete event simulation (DES) frameworks which indeed focus on queueing networks, they currently tend not to permit the simultaneous study of asynchronous, redundancy-based schemes [8]. In order to visualize and analyse the large class of queueing systems within this paradigm [9, 10], I introduced a novel module for Python which is currently available on PyPi: *ParallelQueue* extending the DES package *SimPy*.

The package currently allows for the studying of parallel systems with or without redundancy as well as with the option of allowing thresholds to be implemented in either case. Moreover, the package allows one to specify any inter-arrival and service time distribution as well as their own Monitors, being a class which can gather data from the ongoing simulation to be distributed back to the user upon the completion of a simulation. In particular, the Monitors are currently configured to collect data upon arrival, routing, and job completion as demonstrated by Figure 3.2.

Take Listing **??** and Figure 3.1 for example, which permits one to simulate a Redundancy-2 queueing system with 100 queues in parallel for 1000 units of time while returning the total queue counts over time (which are updated upon a change in queue count).

```
#!/usr/bin/python3
from parallelqueue.base_models import RedundancyQueueSystem
```

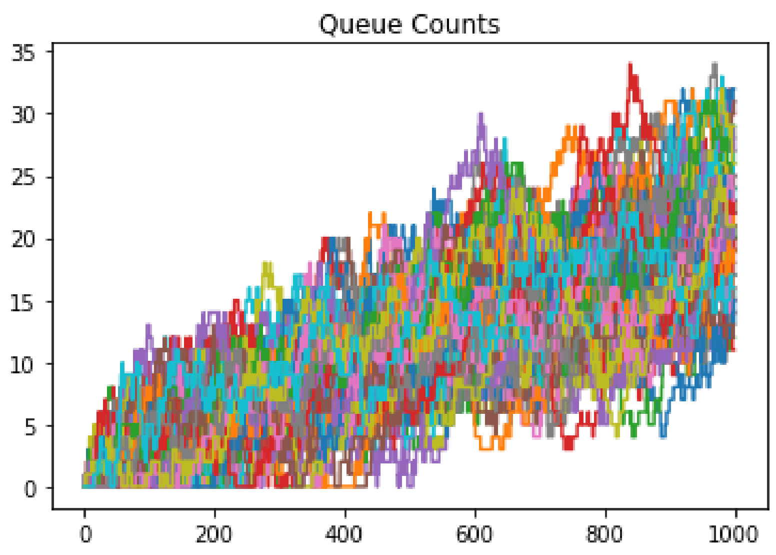

Figure 3.1: Plot using totals of Figure 3.1

```
from parallelqueue.monitors import TimeQueueSize
import random

sim = RedundancyQueueSystem(
                        maxTime=1000.0,parallelism=100, seed=1234,
                        d=2, Arrival=random.expovariate,
                        AArgs=9, Service=random.expovariate,
                        SArgs=0.08, Monitors = [TimeQueueSize])
# Note RedundancyQueueSystem is a ParallelQueueSystem wrapper
sim.RunSim()
totals = sim.MonitorOutput["TimeQueueSize"]

```

Listing 3.1: Python code using *ParallelQueue* to simulate a Redundancy-2 System.

Remarkably, the simulation itself is performed speedily on consumer hardware despite the size of the system as demonstrated in Listing 3.2.

```
CPU times: user 2.02 s, sys: 9.57 ms, total: 2.03 s
Wall time: 2.05 s
Intel i5-8250U (8) @ 3.400GHz

```

Listing 3.2: Runtime Statistics Using the Master ParallelQueue Branch

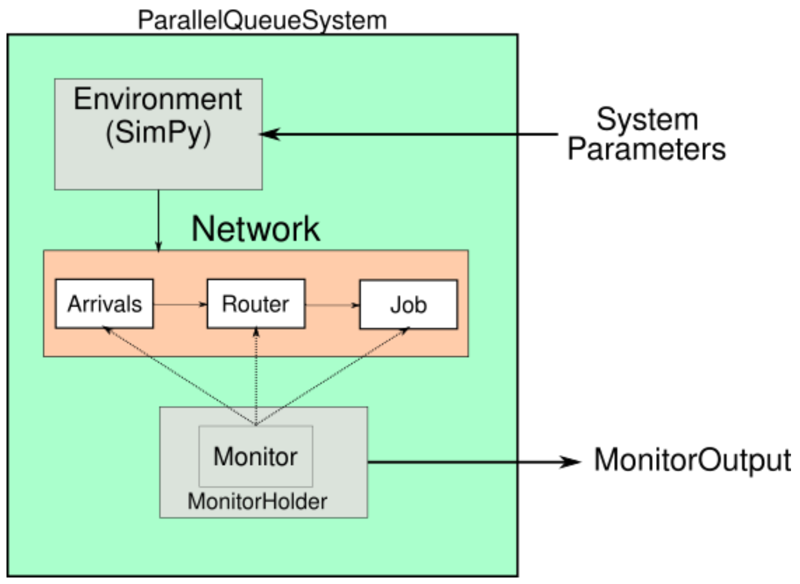

Figure 3.2: Overview of the ParallelQueue API

Altogether, this makes the package easy to parallelize with and thus to compare systems of different sizes and with large running-times. While currently not implemented in any development branch of *ParallelQueue*, the base Python package *multiprocessing* is used in throughout this paper when simulating for the same system across parameters. In general, the main caveat when processing many models is that the storage of the simulation results can quickly begin to consume storage; when processing many models, therefore ensure that they are saved (e.g., using *pickle*) and removed from the local environment when doing analysis.

In terms of development, the models implemented in the *base_models* module use the framework established in 2. That is, modelling redundancy, a hyperedge of sorts is generated whence the dispatcher receives a job to be cloned. This hyperedge then exists for the duration of time for which the replica class is in the system and is defined in such a way that *Monitor* class objects can interact with them in order to acquire data. In Python, such a data structure can be implemented rather easily by employing the *Dict* type which defines a keyed set of values. By keying based on the job arrivals (before cloning), a unique set of marks can be retrieved for the set by simply using the *Dict* object as a reference.

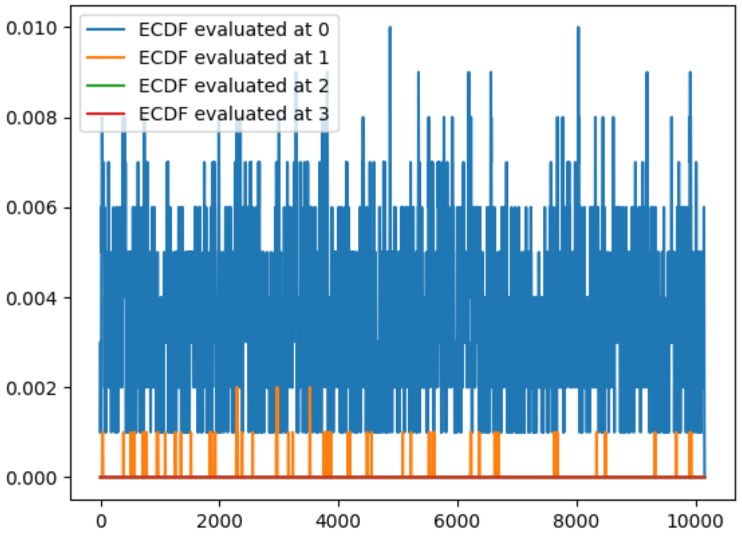

Figure 3.3: Comparisons of Systems: Values are averaged over 30 independent iterations each, running for $t = 1000$)

3.2 Results

3.2.1 Methodology

First, we examine each model in terms of their respective performance in $E(T)$, the expected time each job spends in the system. As Figure 3.3 shows, for a load $\rho \triangleq \frac{\lambda}{\mu} = 0.5$ by taking $\mu = 1, \lambda = 0.5$ (we will assume $\mu \equiv 1$ for the rest of the simulations), Redundancy(2)) and Threshold(2,2) policies are rather alike with low loads as $N \to \infty$. This is to be expected, of course, given that even such a low threshold is unlikely to be exceeded with the processors acting faster than arrivals on average. Ignoring cancellation costs, this clearly demonstrates how utilizing otherwise dormant queues comes to benefit the system's performance. Note that the figure is generated with the *same* seed generation scheme for 30 different seeds per iteration, making the overlap a product of the two accessing random numbers from the system at the same points in time (in their respective simulations).

As [11] show, a Redundancy(d) system is asymptotically stable if and only if $\rho < 1$.

Given that, in premise, Threshold(r, d) models are more or less a superclass of Join-The-Shortest queue and Redundancy(d) (trivially with rising r implying no threshold exists and thus copies should always be made as in the case of Redundancy(d), it is perhaps most interesting to examine if Threshold models are better able to handle high-load environments. Proving the superclass property in terms of Redundancy is relatively easy and is done in Lemma 1. By contrast, after merely setting $r \equiv 0$, we get $\mathcal{D}_{\text{Thresh(0,d)}} \overset{d}{=} \mathcal{D}_{\text{JSQ(d)}}$ by definition. To prove the lemma, we first must introduce the concepts of stochastic domination and coupling.

Definition 11 (Stochastic Domination)**.**
$\mathbf{X}_1^{(N)}$ *is said to dominate* $\mathbf{X}_2^{(N)}$*, using the notation* $\leq_{st}$*, when* [6]

$$\mathbf{X}_1^{(N)} \leq_{st} \mathbf{X}_2^{(N)} \iff \#X_{1,i} \leq \#X_{2,i} \quad \forall i \in [N] \quad P - a.s.$$

Definition 12 (Coupled Process)**.**
For stochastic processes $X_t \in Pr(\Omega_x, \mathcal{F}_x, \mathcal{F}_{t,x})$ *and* $X_t \in Pr(\Omega_y, \mathcal{F}_y, \mathcal{F}_{t,y})$*.* X_t *and* Y_t *are said to be coupled over the probability space* $Pr(\Omega, \mathcal{F}, \mathcal{F}_t)$ *if there exist* $\hat{X}_t, \hat{Y}_t \in Pr(\Omega, \mathcal{F}, \mathcal{F}_t)$ *such that* $\hat{X}_t \overset{d}{=} X_t$ *and* $\hat{Y}_t \overset{d}{=} Y_t$ *[6].*

Lemma 1. *For* $\mathbf{X}^{(N)}$ *being a system of queues such that* $\rho < 1$*,*

$$\mathcal{D}_{\mathit{Thresh(r,d)}}(X^{(N)}) \overset{r\to\infty}{\rightsquigarrow} \mathcal{D}_{\mathit{Red(d)}}(X^{(N)}).$$

Proof:

To show the sequence of routing algorithms to be convergent, we will construct a coupling in such a manner that $\forall r \in \mathbb{R}^+$, the system is stochastically dominated by another which is made to be finite [5] . To do so, for $\mathbf{X}_1^{(N)}$ under Thresh(r, d), denote the first arrival as T_{ξ_1}, at which time it is the case that a selection set, $\nu \subset [N]$, is prescribed wherein there exists $\hat{X} \in \{X_i\}_{i\in\nu}$ such that $\#(\hat{X}) > r$. Thus, for $t \in [0, T_{\xi_1})$, we have $\mathcal{D}_{\text{Thresh(r,d)}}(X^{(n)}) = \mathcal{D}_{\text{Red(d)}}(X^{(n)})$. For clarity, let us now consider $\mathbf{Y}^{(N)}$ to be a copy of $\mathbf{X}^{(N)}$ such that they are independent, identical in distribution and in terms of the marks of *arrival process and job-size draws* along with the queues parsed (as was similarly done in [6] in Lemma 4.1 using Definition 12). Effectively, the only difference being left between these copies is r changing which queues receive clones and thus, too, the mark of queue-dependency. At time T_{ξ_1} we clearly have $\mathcal{D}_{\text{Red(d)}}(X^{(n)}) = \mathcal{D}_{\text{Thresh(r,d)}}(Y^{(n)})$ due to there being no existing jobs for which the threshold would perclude cloning in the case of Thresh(r, d). As such, we also have $\mathcal{D}_{\text{Thresh}(r,d)}(Y^{(N)}(T_{\xi_1})) \leq_{st} \mathcal{D}_{\text{Red}(d)}(X^{(N)}(T_{\xi_1}))$.

Now, assume $\rho < 1$, giving us $\#Y_i < \infty \quad \forall t \in \mathbb{R}^+$ with probability 1 [11]. As such, we have that $\forall r \in \mathbb{R}^+$ there exists $\xi_1(r)$ such that $P_r(\mathbf{X}^{(N)}(t) = \mathbf{Y}^{(N)}(t) | t \in [0, T_{\xi_1}) = 1$

$P_r(A) := P(\mathcal{D}_{M(r)}(A))$ wherein M denotes the routing algorithm of process A. Note that $\xi_1(r)$ is monotone increasing in r. Letting $r \to \infty \Rightarrow \xi_1 \to \infty$, we then have $\mathbf{X}^{(N)}(t) = \mathbf{Y}^{(N)}(t) \quad \forall t \in \mathbb{R}^+$ in terms of distribution (i.e., as the result holds $\forall t \in \mathbb{R}^+$ as $r \to \infty$), implying the required weak convergence from below for $\mathcal{D}$ in law over system $\mathbf{X}^{(N)}(t)$. □

In order to actually test the hypothesized results with respect to independence, we employ the joint Hilbert-Schmidt Independence Criterion (HSIC) [12] to test the independence of queue counting processes $\{\#X_n(t)\}_{n\in[N]}$ in a Monte Carlo-styled simulation consisting of N_{MC} simulations for the queueing process $\mathbf{X}^{(N)}$.

Definition 13 (HSIC Test for Two Processes)**.** *Following [13], for processes X_t and Y_t with common time domain $t \in T$, denote by κ_x, κ_y the kernels which map each process to a seperable reproducing kernel Hilbert space ($\mathcal{H}_x, \mathcal{H}_y$, respectively). Define HSIC to be the $\mathcal{H}_x \otimes \mathcal{H}_y$ norm of form:*

$$HSIC(\mathcal{H}_x, \mathcal{H}_y) = \|\mu_{XY} - \mu_X \otimes \mu_Y\|^2_{\mathcal{H}_x\otimes\mathcal{H}_y},$$

where μ_Z denotes a mean embedding of process Z onto space $\mathcal{H}_Z$.

In order to test the hypotheses of

$$H_0 : P_{XY} = P_X P_Y, H_a : P_{XY} \neq P_X P_Y,$$

where P_Z denotes the marginal distribution of Z, [13] suggest a resampling procedure. Specifically, for any estimator of HSIC(XY), $H_{HSIC}(XY)$, sample a common set of N_{time} times, $\{\tau_\ell\}_{ell\leq N_{time}} \subset T$, and calculate

$$K := \{H_{HSIC}(X_{(\{\tau_{N_{time}}\})}Y_{\gamma(\tau_{N_{time}})}))\}_{\gamma\in\Gamma}$$

where $X_{(\{\tau_{N_{time}}\})}$ denotes X_t restricted to the sampled times $\{\tau_{N_{time}}\}$ and Γ is a set of permutations on these N_{time} samples ordered such that $K_i \leq K_{i+1}$. After taking $c_\alpha := K_{(1-\alpha)N_{time}}$, the null hypothesis is rejected if $H_{HSIC}(XY) > c_\alpha$.

Unlike the procedure utilized by [13], however, we adopt a modification described in [12] to test for joint independence across N processes. While retaining the time sampling procedure of [13], we are instead concerned with testing upon the so-called dHSIC.

Definition 14 (dHSIC)**.** *For a set of d stochastic processes $\mathcal{X} = \{X_i\}_{i\leq d}$, define dHSIC as:*

$$dHSIC(\mathcal{X}) := \|\mu_{\bigotimes_{i\leq d}(P_{X_i})} - \mu_{\{P_{X_i}\}_{i\leq d}}\|_{\bigotimes_{i\leq d}\mathcal{H}_{X_i}}$$

Table 3.1: dHSIC for Varying N

ρ	N	Thresh	dHSIC
0.8	2	1.00e+00	1.00e+00 *
	5	7.62e-04	6.43e-01
	25	3.08e-13	3.43e-01
	50	4.63e-16	9.06e-01
	100	4.91e-16	8.52e-01
0.9	2	1.00e+00	1.00e+00 *
	5	4.23e-04	4.85e-01
	25	1.06e-12	4.59e-01
	50	3.73e-16	5.79e-02
	100	4.96e-16	1.64e-01
0.99	2	1.00e+00	5.13e-01
	5	5.78e-04	5.19e-01
	25	6.28e-12	9.92e-01
	50	4.39e-16	9.10e-01
	100	4.21e-16	8.70e-01

Note: Sampled times comprise every 5th value of the set $[100, 200]$. Data are drawn from 12 simulations run for each N and are used to derive the test at $\alpha = 0.05$.

Much like the procedure described in Definition 13, an estimator and critical value can also be calculated in order to test for independence. The particular means of estimation used in this research is outlined in Sections 4.1, 4.2, and B.5 of [12] and is implemented in the codebase used for our simulations as outlined in Appendix A.

3.2.2 Simulation Results

Table 3.2: dHSIC Test for Large t

ρ	N	Thresh	dHSIC
0.8	2	1.00e+00	1.00e+00 *
	5	8.80e-04	2.71e-01
	25	1.36e-12	1.68e-01
	50	4.70e-16	4.07e-01
	100	5.14e-16	8.86e-01
0.9	2	1.00e+00	1.00e+00 *
	5	7.60e-04	7.52e-01
	25	3.88e-13	7.98e-02
	50	2.82e-16	3.39e-01
	100	3.71e-16	4.27e-01
0.99	2	1.00e+00	1.00e+00 *
	5	8.77e-04	2.69e-01
	25	4.54e-11	8.32e-01
	50	4.02e-16	5.03e-01
	100	5.00e-16	3.99e-01

Note: Sampled times comprise every 5th value of the set $[900, 1000]$. Data are drawn from 12 simulations run for each N and are used to derive the test at $\alpha = 0.05$.

References

[1] A. Mukhopadhyay and R. R. Mazumdar, "Analysis of randomized join-the-shortest-queue (JSQ) schemes in large heterogeneous processor sharing systems," *IEEE Transactions on Control of Network Systems*, vol. 3, no. 2, pp. 116–126, 2016.

[2] T. Austin, "Exchangeable random measures," *Annales de l'Institut Henri Poincaré*, vol. 51, no. 3, pp. 842–861, 2015.

[3] T. Austin, "On exchangeable random variables and the statistics of large graphs and hypergraphs," *Probability Surveys*, vol. 5, no. 0, pp. 80–145, 2008.

[4] U. Ayesta, T. Bodas, and I. Verloop, "On a unifying product form framework for redundancy models," *Performance Evaluation*, vol. 127-128, pp. 93–119, 2018.

[5] F. Baccelli and P. Brémaud, *Elements of Queueing Theory.* Springer Berlin Heidelberg, 2003.

[6] M. Bramson, Y. Lu, and B. Prabhakar, "Asymptotic independence of queues under randomized load balancing," *Queueing Systems*, vol. 71, no. 3, pp. 247–292, 2012.

[7] T. Hellemans, T. Bodas, and B. Van Houdt, "Performance analysis of workload dependent load balancing policies," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 2, 2019.

[8] G. I. Palmer, V. A. Knight, P. R. Harper, and A. L. Hawa, "Ciw: An open-source discrete event simulation library," 2019.

[9] S. Shneer and A. Stolyar, "Large-scale parallel server system with multi-component jobs," *arXiv:2006.11256 [cs, math]*, 2020.

[10] J. Cruise, M. Jonckheere, and S. Shneer, "Stability of JSQ in queues with general server-job class compatibilities," *arXiv:2001.09921 [math]*, 2020.

[11] K. Gardner, M. Harchol-Balter, A. Scheller-Wolf, M. Velednitsky, and S. Zbarsky, "Redundancy-d: The power of d choices for redundancy," *Operations Research*, vol. 65, no. 4, pp. 1078–1094, 2017.

[12] N. Pfister, P. Bühlmann, B. Schölkopf, and J. Peters, "Kernel-based tests for joint independence," *arXiv:1603.00285 [math.ST]*, 2016.

[13] F. Laumann, J. von Kügelgen, and M. Barahona, "Kernel two-sample and independence tests for non-stationary random processes," 2021.

Appendix A

*d*HSIC Resampling Cython Implementation

```
#cython: language_level=3
#cython: infer_types=True
import random
from copy import deepcopy
import numpy as np
cimport numpy as np
from sklearn.metrics import pairwise_distances
from sklearn.metrics import pairwise_kernels

cpdef width(Z):
    dist_mat = pairwise_distances(Z, metric='euclidean')
    return np.median(dist_mat[dist_mat > 0])

cdef center_k(X, width_X, m=None):
    if m is None:
        m = X.shape[0]
    H = np.eye(m) - (1 / m) * (np.ones((m, m)))
    K = pairwise_kernels(X, X, metric='rbf', gamma=0.5 / (width_X ** 2))
    K = H @ K @ H
    return K

cpdef list time_sampler(X, time_samples, max_time = 1000):
    """
    For a list of runs of the same process, returns array of each at
   specified times.
    Samples using binary search algorithm (https://numpy.org/doc/stable/
   reference/generated/numpy.searchsorted.html).

    :param X: list of time-indexed (sorted) data of the same process with
    the same max running time.
    :param time_samples: list of times to sample at.
    :param max_time: maximum allotted time per simulation.
    :return: data at sampled times.
    """
    cdef list ret = []
    for time in time_samples:
        data_slice = []
        for proc in X:
            time_list = list(proc.keys())
            insertion_point = np.searchsorted(time_list, time)  # a[i-1]
   < v <= a[i] via binary search algo
            if time_list[insertion_point] != time:
                insertion_point = time_list[insertion_point - 1]  #
   Cadlag
            data_slice.append(proc[insertion_point])
        ret.append(data_slice)
    return ret

cdef dHSIC_hat(Xs):
    """https://arxiv.org/pdf/1603.00285.pdf -- see algorithm 1"""
    cdef int x_len = Xs[0].shape[0]
    #inits
    t1 = 1
    t2 = 1
    t3 = (2 / x_len)
    for x in Xs:
        K = center_k(x, width(x))
        t1 = np.multiply(t1, K)
        t2 = (1 / x_len ** 2) * t2 * np.sum(K)
        t3 = (1 / x_len) * t3 + np.sum(K, axis=0)
    return (1 / x_len ** 2) * np.sum(t1) + t2 - np.sum(t3)

cpdef dHSIC_resample_test(list Xs, int shuffle=500, float alpha = 0.05):
    """Resampling implementation -- see sec 4.3. of https://arxiv.org/pdf
   /1603.00285.pdf
     Returns stat and threshold (if possible)."""
    init = dHSIC_hat(Xs)
    locX = deepcopy(Xs) # deep copy
    cdef int hits = 0
    cdef list dXs = []
    for i in range(shuffle):
        random.shuffle(locX)  # void shuffles
        permed = dHSIC_hat(locX)
        if permed >= init:
            hits += 1
        dXs.append(permed)
    dXs.sort()
    critIndex = 0
    for i in range(shuffle):
        if dXs[i] == init:
            critIndex += 1
    critIndex += np.ceil((1-alpha) * (shuffle + 1))
    critIndex = int(critIndex)
    if critIndex < shuffle:
        thrsh = dXs[critIndex]
    else:
        thrsh = None
    #stat, threshold
    return (hits + 1) / (shuffle + 1), thrsh
```