CHALMERS
UNIVERSITY OF TECHNOLOGY

Shared-object system equilibria: Delay and throughput analysis

Downloaded from: https://research.chalmers.se, 2025-12-04 12:15 UTC

Citation for the original published paper (version of record):
Salem, I., Schiller, E., Papatriantafilou, M. et al (2018). Shared-object system equilibria: Delay and throughput analysis. Theoretical Computer Science, 731: 1-27.
http://dx.doi.org/10.1016/j.tcs.2018.03.030

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Theoretical Computer Science 731 (2018) 1–27

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Shared-object system equilibria: Delay and throughput analysis ☆

Iosif Salem *, Elad M. Schiller, Marina Papatriantafilou, Philippas Tsigas

Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96, Göteborg, Sweden

ARTICLE INFO

Article history:
Received 8 April 2016
Received in revised form 14 March 2018
Accepted 29 March 2018
Available online 16 April 2018
Communicated by R. Klasing

Keywords:
Delay and throughput analysis
Resource sharing
Distributed systems

ABSTRACT

We consider shared-object systems that require their threads to fulfill the system jobs by first acquiring sequentially the objects needed for the jobs and then holding on to them until the job completion. Such systems are in the core of a variety of shared-resource allocation and synchronization systems. This work opens a new perspective to study the expected job delay and throughput analytically, given the possible set of jobs that may join the system dynamically. We identify the system dependencies that cause contention among the threads as they try to acquire the job objects. We use these observations to define the shared-object system equilibria. We note that the system is in equilibrium whenever the rate in which jobs arrive at the system matches the job completion rate. These equilibria consider not only the job delay but also the job throughput, as well as the time in which each thread blocks other threads in order to complete its job. We then further study in detail the thread work cycles and, by using a graph representation of the problem, we are able to propose procedures for finding and estimating equilibria, i.e., discovering the job delay and throughput, as well as the blocking time. To the best of our knowledge, this is a new perspective, that can provide better analytical tools for the problem. That is, our methods can be used to estimate performance measures similar to ones that can be acquired through experimentation on working systems and simulations, e.g., as job delay and throughput in (distributed) shared-object systems.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Our problem's domain considers computing entities, which we call threads. Each thread runs a sequential program (a job) that has to acquire reusable resources (objects), often several at the same time, for a bounded time of use. To guarantee deadlock absence, it is important that all threads acquire the objects in an ordered manner. For example, one can deterministically define a total order among the objects, such that the threads acquire them in totally ordered manner. A common way to model such systems is to consider a generalization of the dining philosophers problem, as in [19,22], in which every job includes a fixed set of objects that it may need. This approach provides a worst-case complexity analysis, which is exponential on the system's parameters.

We provide a new perspective that enables an analysis of the evaluation metrics by considering measures both at the system level and at the level of each resource. In particular, we consider performance measures that are associated with

☆ A technical report and an extended abstract of this paper appeared in [25] and [26], respectively.

* Corresponding author.
E-mail addresses: iosif@chalmers.se (I. Salem), elad@chalmers.se (E.M. Schiller), ptrianta@chalmers.se (M. Papatriantafilou), tsigas@chalmers.se (P. Tsigas).

https://doi.org/10.1016/j.tcs.2018.03.030
0304-3975/© 2018 Elsevier B.V. All rights reserved.

each resource, such as the job delay and completion rate, as well as the blocking time of each thread on each object. On the system level, we consider the job arrival and completion rates, as well as the total number of threads and objects.

We study shared-object systems that require their threads to fulfill the system jobs by first acquiring sequentially all of the job objects. The job then holds on to these objects until the job operation is done. We identify the system dependencies that cause contention among the threads as they try to acquire the job objects. We study the (stochastic) processes of job arrival and completion with an emphasis on the cases in which the job arrival rate matches the job completion rate, i.e., the job throughput. In these cases, the system is in a *shared-Object System Equilibrium* (OSE). For a given $\varepsilon > 0$ and an OSE, we say that the system is in an ε-*OSE* when the completion rate of any job differs from the one of an OSE by at most ε. We study the conditions for a given shared-object system to be in an OSE as well as contention-related properties of OSEs, i.e., the *expected* job delay and completion rate, as well as the time in which each thread blocks other threads and by that prevents them from making progress. We propose an analytical procedure for finding (approximate) OSEs, which we call ε-OSEs (for a given error ε). Moreover, we estimate the performance measures of systems that are in ε-OSE.

The existing practice considers job delay and completion rate as the performance measures of working systems. Empirical experiments often study shared-resource systems at their saturation point in which the system is at its peak utilization. Let us describe peak utilization scenarios using two vectors; one for job arrival rates and another for their completion rates. A saturation point is the case in which: (1) the system is in equilibrium, i.e., the arrival rate of any particular job matches the completion rate of this job, as well as (2) the system is at the stage at which a higher arrival rate of any job to the system cannot increase its completion rate. Our study considers the entire range of these equilibria rather than just peak utilization scenarios (Section 2).

1.1. Related work

We consider a generalization of the dining philosophers problem, as in [19,22], in which every job includes a fixed set of objects that it may need. This problem has well-known results studying the worst-case job delays, which may even be exponential on metrics, such as the chromatic number of the resource graph [18,19]. In this graph, the vertices (objects) are connected if there is at least one thread that may request them both at any point in time. In the context of actual systems, the expected time is rather different than the worst case and therefore computer experiments are the common way for evaluating the system performance.

Systems that support atomic synchronization primitives, such as compare-and-swap (CAS) [14], are motivating examples for our model. In particular, while CAS provides mutually exclusive read and write access to one memory location (word), extensions for accessing many memory locations exist in the literature, such as [13,17]. Harris et al. [13] propose a multi-word compare and swap extension of CAS (double-word CAS and multi-word CAS), while Luchangco et al. [17] propose an atomic primitive that allows reading many words but writing only to one. Our work can provide an analytical early stage evaluation of such systems, by adjusting the job size and the job operation times accordingly.

1.2. Our contribution

We study analytical tools that provide the means to estimate performance measures of working distributed systems. In the context of synchronization challenges that are modeled via a generalization of the dynamic dining philosophers problem, our analytical tools are the first, to the best of our knowledge, to consider performance measures similar to the ones that can be acquired via experimentation on working systems and simulations.

For a given number of threads and objects, as well as the jobs and their arrival rates, we provide a way to analyze the delay of jobs and their completion rates as well as the time for which the threads are blocked. Throughout our analysis we use the thread work cycle events (sections 5.2 and 5.4) to verify our modeling approach. In addition to the job completion period (Approximation 3), we analyze a number of key properties, such as the probability of a thread to request a particular resource after the acquisition of another specific resource, the time during which threads that have acquired a particular resource block other threads that ask to access the same resource (Approximation 4), as well as the time between two requests to access such resources (Approximation 5). Our analysis is based on estimating pairwise states (Section 5.5) that capture these blocking periods, the delay and the throughput with respect to every pair of system items (threads or objects). Since these properties have interdependencies due to thread blocking, we show how the concept of thread work cycles can be represented in subsystems that also include such interdependencies but have no thread blocking. This way, we can resolve these interdependencies (Theorem 6) and estimate the performance of the given (distributed) shared resource system. We present a procedure for satisfying approximately the equilibrium conditions and by that find an ε-OSE as well as the performance measures of the studied system (Section 10). In particular, in Algorithm 3 (Section 10) we compute the first three moments of these pairwise states, and hence the first three moments of job delays and throughput, in an ε-OSE (if such exists).

Our contribution can facilitate early-stage evaluations of systems that are similar to the studied one. Moreover, using our proposed methods, one can analytically, rather than via empirical experiments, study trade-offs among OSEs. Such trade-offs can facilitate the design of mechanisms for adjusting the number of threads and job arrival rates according to the performance measures of a dynamic system.

Notation	Meaning	Page
$objects = (o_1, \ldots, o_M)$	object set, where $M = \lvert objects \rvert$	3
$threads = (t_1, \ldots, t_N)$	thread set, where $N = \lvert threads \rvert$	3
$job_i = \langle objs_i, operation_i \rangle$	the ith job, with object sequence $objs_i$ and operation $operation_i$	3
O_i	operation time of job_i	3
$Q_{thead}(t_n)$	queue of t_n's pending jobs	3
$Q_{object}(o_i)$	queue of o_i's pending requests	3
$\lambda_{i,n}$	job_i's arrival rate to t_n (Exponential distribution)	3
$I[i, n]$	inter-arrival time of job_i to t_n	3

Fig. 1. A summary of the basic notation of Section 2.

Paper organization This paper is organized as follows. In Section 2 we present the system settings for this work. We present the necessary background knowledge in Section 3. In Section 4 we present a high-level description of our approaches. In Section 5 we present a model for the studied problem, which captures the system's dependencies and forms a basis for our analysis and proofs. We then present in detail our solution in the remaining sections. Note that once we find an ε-OSE, we can estimate its performance measures, i.e., job delay, completion rate, and blocking time. To this end, we develop a number of analytical tools for OSEs. In Section 6 we give estimations for the probabilities of threads requesting access to objects. Given the job arrival rates, we show how to estimate the probabilities for threads to follow a certain object acquisition sequence (Section 6). We are then able to formulate recursive equations (with interdependencies) for calculating the blocking periods and the completion rates (sections 7, and respectively, 8). We overcome these dependencies and solve these recursive equations by analyzing the thread work cycles (Section 9). In Section 10 we present Algorithm 3, which gives a procedure for finding OSEs and computing delay and throughput if such exist. In Section 11 we conclude this work.

2. System settings and problem definition

We consider a system that includes *(system) items*, which are (totally ordered) *objects*, $(o_1, \ldots, o_M)$, and (totally ordered) *threads*, $(t_1, \ldots, t_N)$. The objects are shared in a mutually exclusive way, i.e., only one thread at a time may gain access to an object. Each thread is to carry out one *job* at a time, where $job_i = \langle objs_i, operation_i \rangle$, J is the number of the system's jobs, $i \in [1, J]$ and $objs_i = (o_{i_1}, \ldots, o_{i_k})$ is an arbitrary, non-empty subsequence of $(o_1, \ldots, o_M)$, and thus $objs_i$ follows the same order.

A thread carries out job_i by gaining mutually exclusive access to the objects in $objs_i$ and in the order implied by $objs_i$, executing the operation O_i, and then releasing access to the objects in $objs_i$. We assume that $objs_i$ is a fixed vector and that different jobs may have different object vectors of different lengths. Moreover, we assume that every object is included in the object set of at least one job. Furthermore, the *(job) operation time*, O_i, is a random variable with a known distribution. Namely, we assume that the time it takes to execute the job operation is known by its first three moments.

Throughout this paper we consider that random variables are known when we know their first three moments. Therefore, we will focus on computing the first three moments of the studied problem's unknown random variables. We denote by $E[X^m]$ the m-th moment of a random variable X. We give a summary of the section's basic notation in Fig. 1.

2.1. Job arrival rates and job-to-thread assignment

We assume that the time between two consecutive arrivals of job_i to t_n is a random variable $I[i, n]$ (inter-arrival period), where $i \in [1, J]$, $n \in [1, N]$. We define the *job arrival rate*, $\lambda_{i,n}$, in which job_i arrives at the system that then places job_i in a (first in, first out) queue, $Q_{thread}(t_n)$, where $\lambda_{i,n}$ is a positive real number. The inter-arrival period, $I[i, n]$, follows an exponential distribution, $Exp(\lambda_{i,n})$. Note that this is a common way to model arrivals, e.g. [7]. As soon as t_n becomes available, the system assigns to t_n the job that is on $Q_{thread}(t_n)$'s top. Moreover, we assume that at least one job is arriving to every thread's queue. We focus on systems that can be in an equilibrium for which the number of pending jobs in $Q_{thread}(t_n)$ is bounded.

Our assumption that the job arrival rates follow exponential distributions allows us to consider the inter-arrival time of jobs to the entire system as an exponential distribution of rate $\Sigma_{i,n}\lambda_{i,n}$ (cf. Poisson process merging and splitting properties [1,5]). Thus, defining the system to have separate queues for each thread does not compromise generality, since a system scheduler assigns every arriving job to a thread.

2.2. Acquisition requests and periods

A thread can acquire a particular object, o_ℓ, by pending its *(acquisition) request* in a (first in, first out) queue $Q_{object}(o_\ell)$ until all (previously) waiting threads in $Q_{object}(o_\ell)$ have acquired and released o_ℓ. The *acquisition period*, A, is a known random variable (i.e., we know its first three moments) that refers to a period that starts when a thread has acquired an object (or just been assigned to a new job) and ends as soon as that thread places a request for the next object.

Once the thread sequentially acquires the entire object set, $o_{i_1}, \ldots, o_{i_k}$ of job_i, it executes the job operation, $operation_i$. When $operation_i$ is completed, the thread releases access to the object set and job_i is completed. We say that a thread is *blocking*, when other threads are queuing for its acquired objects. That happens whenever different jobs have overlapping object vectors. Note however that threads carry out jobs within finite time even in the presence of blocking, because threads acquire their job's objects following the order of the job's object set and these sets follow the ascending object order. The maximum number of pending requests in an object's queue is $N-1$, since every thread can carry out one job at a time.

2.3. Job delay and throughput

We define the *job delay* on t_n, $\tilde{D}_{t_n}$, to be the random variable that gives the time between the arrival of a job to t_n's queue and the completion of that job. Moreover, we denote by $\tilde{\mathcal{T}}_{t_n}$ the random variable that gives the time between consecutive job completions of t_n. Then, we define each thread's *completion rate* (throughput) by $1/E[\tilde{\mathcal{T}}_{t_n}]$, where $E[\tilde{\mathcal{T}}_{t_n}]$ is the expected value of $\tilde{\mathcal{T}}_{t_n}$.

2.4. Problem definition: computing shared-object system equilibria

Shared-object system equilibria For a given system, $\psi = \{\tilde{D}_{t_n}, \tilde{\mathcal{T}}_{t_n}\}_{n\in[1,N]}$ is the *system state*. Suppose that a system is in a state in which, for every thread the job arrival rate is equal to the job completion rate, i.e., $\forall n \in [1, N]$, $\Sigma_{i=1}^{J} \lambda_{i,n} = 1/E[\tilde{\mathcal{T}}_{t_n}]$. Note that we refer to $\Sigma_{i=1}^{J} \lambda_{i,n}$ as t_n's (aggregate) arrival rate (cf. Section 2.1). We say that $\psi^* = \{\tilde{D}^*_{t_n}, \tilde{\mathcal{T}}^*_{t_n}\}_{n\in[1,N]}$ is the *shared-Object System Equilibrium* (OSE). For a given $\varepsilon > 0$ and an OSE ψ^*, we say that the system state ψ is an *ε-OSE* when $\forall n \in [1, N]$, $|\tilde{D}^*_{t_n} - \tilde{D}_{t_n}| < \varepsilon \wedge |\tilde{\mathcal{T}}^*_{t_n} - \tilde{\mathcal{T}}_{t_n}| < \varepsilon$ holds. Namely, the corresponding values of each item in ψ and ψ^* differ by less than ε.

We remark that a system cannot always reach a state that satisfies the OSE conditions, and thus neither the ones of an ε-OSE. We refer to the random variable $blocking(n)$, which gives the time between the assignment of a job to t_n until the completion of that job, as t_n's *blocking period*. Equilibria are *unreachable* when there is a thread with blocking period that is longer than the inter-arrival time of jobs to that thread. Note that in that case, the thread's queue is increasing continuously.

The studied problem In this work we study the problem of computing an ε-OSE $\psi^* = \{\tilde{D}^*_{t_n}, \tilde{\mathcal{T}}^*_{t_n}\}_{n\in[1,N]}$, basing on the known system parameters and a given error ε. Specifically, we consider the problem of computing the first three moments of the job delay, $\tilde{D}^*_{t_n}$, and of the time between consecutive job completions, $\tilde{\mathcal{T}}^*_{t_n}$, of the system's threads, when the system is in ε-OSE (for a given ε). We consider as known parameters, the number of threads N and objects M, the set of jobs $\{job_i\}_{i\in[1,J]}$ (which includes the object sets and job operation times), as well as the set $\{\lambda_{i,n}\}_{i\in[1,J],\, n\in[1,N]}$ of job-to-thread arrival rates.

Computing the job delay and throughput is non-trivial due to the fact that they both depend on blocking, i.e., the cases of threads waiting for other threads to release objects that they want to acquire. That is, job delay and throughput depend on the waiting time of a job in a thread's queue and the waiting times of the thread's object acquisition requests in the object queues of the job's object set. For example, there can be an instantiation of the system, in which t_x has access to o_k and waits for access to o_{k+1}, while t_y's request is pending in o_k's queue. Thus, the delay of the job that t_y is carrying out, directly depends on the time for which t_x is holding access to o_k, which includes the time for which t_x is waiting for accessing o_{k+1} (and possibly other objects). Therefore, the key challenge that we address is to effectively model and analyze these dependencies. Section 4 gives a complete overview of our approach.

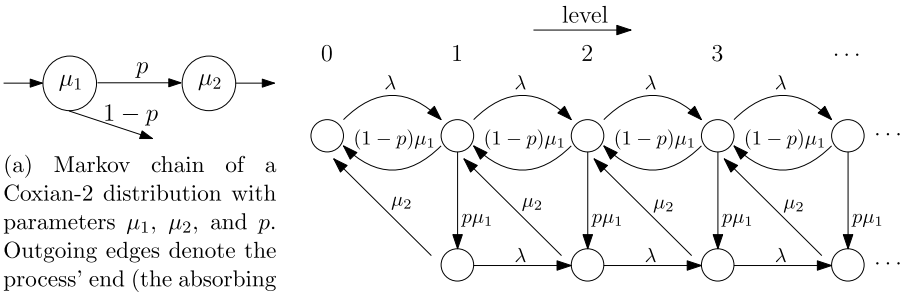
3. Background

In this section we explain how existing tools from queuing theory [1] can be deployed to compute the delay $\tilde{D}_{t_n}$ and job completion period $\tilde{T}_{t_n}$ of the thread queues, given the distribution of the job blocking periods (Section 3.1). Then, we present more complex tools from queuing networks [5], which we will use for analyzing shared-object systems (Section 3.2).

3.1. Computing job delay and throughput when the job blocking period is known

We can compute $\tilde{D}_{t_n}$ and $\tilde{T}_{t_n}$ using the distribution of job arrivals to t_n, $I(n)$, as well as the distribution of the blocking period, $blocking(n)$ (Section 2.4), i.e., the time between job assignment and job completion. Since the job-to-thread arrivals follow exponential distributions, $Exp(\lambda_{i,n})$, $I(n)$ follows a known exponential distribution with parameter $\Sigma_{i=1}^{J} \lambda_{i,n}$ (Section 2.1). However, the distribution of $blocking(n)$ is unknown, since it depends on thread blocking.

In our analysis, we provide an algorithm for computing $blocking(n)$'s first three moments (cf. Algorithm 3 and Section 8), and below we explain how we can use these moments to compute $\tilde{D}_{t_n}$ and $\tilde{T}_{t_n}$. Queuing theory provides a queue's performance parameters in its steady-state, i.e., the case where the arrival and completion rates match, which implies that $\tilde{T}_{t_n}$ follows the same distribution with $I(n)$. Specifically, it gives the steady-state probabilities for the queue length from which we can compute the distributions of the queue waiting time and the job delay.



(a) Markov chain of a Coxian-2 distribution with parameters μ_1, μ_2, and p. Outgoing edges denote the process' end (the absorbing state is reached).

(b) QBD process of a thread's M/Coxian-2/1 queue.

Fig. 2. Fig. 2b depicts the Markov chain (QBD process) that represents the M/Coxian-2/1 queue of a thread, t_n. The arrival rate is $\lambda = \Sigma_i \lambda_{i,n}$, and μ_1, μ_2, as well as p are the parameters of the Coxian-2 distribution (Fig. 2a) that matches *blocking*(n)'s moments (blocking period). The QBD process' states are organized in levels, which denote the number of jobs in t_n's queue (including the one that t_n is possibly carrying out). The single state in level 0 denotes that the thread is idle. In level i, $i > 0$, there are two states due to *blocking*(n)'s Coxian-2 distribution. The upper state ends with rate $(1-p)\mu_1$, the transition rate from the upper to the lower state is $p\mu_1$, and the lower state ends with rate μ_2.

We compute the steady-state probabilities of t_n's queue, $Q_{thread}(t_n)$, by analyzing it as an M/Coxian-2/1 queue. An M/Coxian-2/1 queue is a queue in which the arrivals follow an exponential distribution (denoted by M) and the blocking period follows a Coxian-2 distribution [1] (Fig. 2a), while 1 denotes that only t_n carries out the pending jobs of its queue (cf. Kendall notation of queues [1]). By our assumptions $I(n)$ follows indeed an exponential distribution. A Coxian-2 distribution is a special case of a phase-type distribution [1], which represents the time until a Markov process with two states reaches an absorption (ending) state, and is commonly used for fitting unknown distributions [1,2]. Hence, after computing the first three moments of *blocking*(n) using the algorithm that we propose in this paper (Algorithm 3 and Section 8), we can use the moments of *blocking*(n) to match it with a Coxian-2 distribution (moment matching method, [2,23] and [1, Section 2.5]).

To analyze the M/Coxian-2/1 queue that models $Q_{thread}(t_n)$'s arrival and blocking distributions, we use the standard approach of the Matrix Geometric Method (MGM) [20]. We define a Markov chain (Fig. 2b) that has the structure of a quasi-birth-death (QBD) process [8] using $I(n)$ and *blocking*(n) (as in [21, Section 3.2] and [12]). The QBD process considers the growth and decrease in the number of jobs in $Q_{thread}(t_n)$ (including the one that t_n is carrying out). Each level corresponds to the queue's length (i.e., $0, 1, 2, \ldots$) and the states of each level correspond to the two states of the Coxian-2 distribution of *blocking*(n), except for level zero that has only one state (t_n is idle). Transitions from level i to $i+1$ are determined by $\Sigma_{i=1}^{J} \lambda_{i,n}$ ($Q_{thread}(t_n)$'s arrival rate), and transitions from i to $i-1$ are determined by the parameters of the Coxian-2 distribution that matches the moments of *blocking*(n).

The MGM [20] provides the steady-state probabilities for each state of t_n, due to the Markov chain's QBD structure. Thus, the (steady-state) probability of the QBD process' zero-level state, $u_{n,0}$, gives the probability of t_n to be idle. The MGM is an iterative method and its running time is in $O(I_{MGM} \cdot m^3)$, where m is the maximum number of states in each QBD level and I_{MGM} is the number of iterations until the method converges. As we described above, $m = 2$, since the Coxian-2 distribution is a Markov chain with two states (Fig. 2). Latouche and Ramaswami in [16, Remark 4.3], explain that in order to have a number of iterations, I_{MGM}, that exceeds 40, it must be (practically) possible for the queue length to grow up to 10^{12}. Since this is considered impractical in the setting of shared-object systems (and in generally practical settings), we can assume that I_{MGM} is a small number, and thus MGM's running time is (practically) constant.

3.2. Relevant tools from queuing networks

Our solution uses tools from queuing networks [5]. Although queuing theory celebrated results provide closed forms for single queues, e.g., M/M/c, M/G/1 [1], and queuing networks, e.g., Jackson [15,5], BCMP [3], and Gordon–Newell networks [9], closed form results are far from been the common case. Specifically, there are no relevant closed-form results that can be used for systems like ours in which a thread can block other threads for a non-exponentially distributed period.

We use tools from queuing networks, where non-exact solutions are often provided [5]. In particular, we consider the work of Ramesh and Perros [23] who study a message passing system of multi-tier server networks in which processes communicate iteratively via what is known in the computer systems community as synchronous I/O (also called blocking I/O). Namely, at any point in time, each process handles at most one (outgoing) connection, which gets blocked when sending a message. Ramesh–Perros [23] use a framework proposed by Baynat and Dallery [4] for estimating the system state. The authors of [23,4] demonstrate the convergence of their iterative methods via numerical experiments, while their algorithms run in polynomial time on the system's size.

4. Overview of the proposed solution

In this section we provide an overview of our approach for computing job delay and throughput, when a shared-object system is in equilibrium (Section 2.4). In Section 3.1 we showed how to compute the delay and throughput of a thread's

queue, say $Q_{thread}(t_n)$ of t_n, when the job blocking period $blocking(n)$, i.e., the time between a job assignment and the job completion, is known. By the definition of shared-object systems, $blocking(n)$, for every $n \in [1, N]$, depends on thread blocking, i.e., the time during which a thread waits for other threads to release the objects that it needs to acquire. Thus, we focus on computing $blocking(n)$, when the system is in equilibrium (ε-OSE), i.e., when the job arrival and completion rates match (Section 2.4).

To the end of computing $blocking(n)$, we provide a comprehensive model of the system's events, which captures the dependencies among threads. Basing on these events, we define the system's performance parameters, i.e., the delay, blocking period, and throughput of thread requests for accessing objects, the throughput of every system item, as well as the probability for each acquisition request to occur (Section 5).

Through our model, we are able to connect shared-object systems with relevant analytical tools from Queuing Networks (Section 3.2). Specifically, we identified two methods that are relevant for computing job delay and throughput in ε-OSEs. The method of network approximations considers systems with continuous arrivals that follow Exponential distributions and deterministic acquisition of system items, however it does not consider thread blocking [8, Section 8.2]. Instead, the approximation methods of Ramesh and Perros [23] consider systems that can analyze thread blocking (Section 3.2).

In Section 6, we use methods from network approximations [8, Section 8.2] to approximate the probabilities of demand requests to occur. Then, in sections 7–8, we follow the approach of [23] to approximate the system's performance parameters. In Section 9 we define closed subsystems (where jobs are circulating instead of arriving externally and leaving the system) that we use to complete the computation of the system's performance parameters. As in [23], these subsystems preserve the system's performance parameters, but are not subject to thread blocking, and hence can be analyzed through the framework of Baynat and Dallery [4] for closed systems (Section 9.2).

We continue Section 9 by proving Theorem 6, which states that the decomposition of shared-object systems to closed subsystems can be mapped to the decomposition that Ramesh and Perros use for their queuing networks [23]. This proof provides the justification of our approximations. We base our proofs on the model that we provided in Section 5. Then, we present Algorithm 3 (Section 10), which consolidates our approaches for computing the system's performance parameters (including job delay and throughput) in ε-OSE (if such exists).

In the remainder of this section we give an overview for each component of our analysis. We outline our modeling approach and the approximation of the system's performance parameters in Section 4.1. In Section 4.2 we summarize the decomposition to closed subsystems that are not subject to thread blocking and the proof of the mapping to the queuing networks of [23]. In Section 4.3 we present an overview of the algorithm for computing job delay and throughput in ε-OSEs.

4.1. Modeling dependencies and approximating the system's performance parameters

We base our modeling approach (Section 5) on defining all the events that occur in a shared-object system. We define events for job assignment, object acquisition requests, supply of access to objects, and object release. By that, we define the *thread work cycles*, which depict the sequence of events that occur when a thread is carrying out a job and the fact that threads alternate between idle periods (of possibly zero length) and periods in which they carry out jobs.

We depict dependencies through an *acquisition graph*, which we construct by combining the job object sets and arrival rates to threads. The graph's vertices are the system's items (threads and objects) and each edge depicts a direct dependency. An edge from a thread and to an object shows that this thread carries out at least one job that starts with that object and an edge from one object to another one shows that there is at least one job object set that includes these two objects consecutively.

To the end of capturing dependencies, we define the system's performance parameters based on the edges of the acquisition graph. That is, each edge depicts a sequence of (two) acquisition requests that occur in the system. We first define blocking period B and delays D for each edge. Then, we define the inter-demand period T for each edge, as well as, the item inter-demand period $\mathcal{T}_s$ for each system item s, which relates to the time between consecutive acquisition requests. We show that the inter-demand periods follow the same distribution with the time between release events (e.g. job completions), which we use to compute throughput.

In Section 7 we approximate the request blocking period through the delay for acquiring the job's remaining objects, as in [23]. By that, we identify forward dependencies, i.e., the blocking period of a request to an object o_d depends on the delay of requests to objects $o_{d'}$, $d' \in (d, M]$. In Section 8 we approximate the inter-demand period of an object o_d (by which we compute throughput), by the inter-demand periods of the threads that (possibly) start their jobs by demanding access to o_d as well as the inter-demand periods of each object o_j, $j \in [1, d)$. Moreover, in Section 8 we also approximate the thread inter-demand periods, which depend on the job arrival rates and on $blocking(n)$. Since not all of these requests occur with the same rate, we use the request probabilities (Section 5) as weights in the approximations of blocking and inter-demand periods. In Section 6 we provide an approximation of these probabilities that bases only on the job arrival rates and the job object sets, which follows the approach of network approximations in [8, Section 8.2].

4.2. Resolving dependencies

We resolve dependencies by decomposing the shared object system to closed subsystems, for each object demand request (Section 9). We refer to these closed subsystems with the term *contention subsystems* and define them such that they are

Algorithm 1: Finding an ε-OSE (procedure sketch).

```
1  Input: M, N, {job_i}_{i∈[1,J]}, {λ_{i,n}}_{n∈[1,N],i∈[1,J]}, demand request probabilities, ε;
2  Output: job delay and throughput; request blocking, delay, inter-demand periods;
3  Start by assuming that all queues are empty;
4  repeat
5      let prevSet ← inter-demand periods of all items;
6      for d = M to 1 (∗ backward iteration ∗) do
7          foreach demand request to o_d (equiv. edge in the acquisition graph) do
8              Approximate the demand request's delays and inter-demand periods through the respective contention subsystem (Section 9);
9              Approximate the demand request's blocking period (Section 7);
10     foreach n ∈ [1, N] do
11         Approximate t_n's inter-demand period (Section 8) and upon OSE condition violation call return('no OSE')
12     for d = 1 to M − 1 (∗ forward iteration ∗) do
13         Approximate the inter-demand periods of requests to o_d through the respective contention subsystem (Section 9);
14         Approximate the inter-demand period of o_d (Section 8);
15 until the system has reached equilibrium (test for ε-OSE using the inter-demand period of every item and the values in prevSet);
16 return job delay and throughput; request blocking, delay, inter-demand periods;
```

not subject to blocking, and yet they preserve the system's performance parameters. We define contention subsystems as closed systems that include subsets of the system's items. The key steps in this definition are that (i) blocking periods in contention subsystems include the entire time that a thread accesses an object, but the thread does not maintain access to that object when it tries to acquire the next object of its job, and (ii) the order in which threads access objects in the shared-object system is preserved in the contention subsystems. This decomposition follows the approach of [23]. Then, we compute the delay and throughput of demand requests through the framework of Baynat and Dallery for closed systems, which we adapt to the context of shared-object systems (Section 9.2).

We then prove that the decomposition of shared-object systems to contention subsystems can be mapped to the decomposition that Ramesh and Perros present in [23] to analyze queuing networks (Section 9). We present the proof for the case of a system with only three objects, $M = 3$, and then present the proof for the general case of systems with M objects.

4.3. Finding approximate equilibria

We compute an approximate equilibrium, ε-OSE, when such is reachable. We propose a procedure that always halts (Algorithm 1 presents the solution sketch and we detail the entire procedure in Section 10). It returns the system in an ε-OSE state whenever the job arrival and completion rates differ by at most ε, or indicates that the system cannot be in a state of an OSE.

The procedure starts with a system state that represents the case in which all queues are empty (line 3). It then estimates the state of a system in which threads can block one another, and the delay grows as more requests are pending in the queues. The procedure works in iterations and decides when to stop updating the item inter-demand periods, i.e., it stops whenever there is no item for which the change in its inter-demand period is greater than ε since the previous iteration (lines 5 to 15).

The procedure repeatedly improves an ε-OSE estimation until the system state satisfies the conditions of an approximate equilibrium. It deals with interdependencies using alternating backward and forward iterations (lines 6, and respectively, 12). Namely, we resolve the forward dependencies in which a demand's blocking period depends on delays for objects with higher index by *iterating backwards*. This backward iteration starts from $d = M$ and counts downwards. Similarly, we use *forward iterations* for resolving backward dependencies with respect to o_d's item inter-demand period, because all of o_d's backward dependencies are resolved. This loop also updates the thread inter-demand periods (line 11). Together with the estimation of the thread inter-demand periods, the procedure checks whether the OSE condition is violated (Section 2.4). In case the OSE condition is violated, the loop breaks and the procedure returns. We show that each iteration takes $O(M^2 \cdot N^4 + M^3)$ time (Section 10.5).

5. Modeling dependencies in shared-object systems

Overview In this section we give a comprehensive model for shared-object systems, on which we will base our analysis. We start by defining all the events that occur in the system, through the concepts of job acquisition paths (Section 5.1) and (thread) work cycles (Section 5.2). Then, we model the dependencies between events by defining the acquisition graph (Section 5.3), conditional events that occur within a work cycle, as well as consecutive events that occur in consecutive work cycles (Section 5.4). We also define the probability of conditional events to occur (Section 5.5), i.e., the probability of a thread, t_n, to demand access to an object, o_d, immediately after (i) t_n has been assigned a job, or (ii) t_n has acquired access to an object, o_k. We refer to items (i) and (ii) as supply events.

We build up on these definitions to model the system's performance parameters, in a manner that captures the dependencies between the system's items. Given the occurrence of a supply event of a thread (see items (i) and (ii) above), we

Notation	Meaning	Page
$path_{i,n}$	acquisition path $(t_n, o_{i_1}, \ldots, o_{i_k})$	8
$cycle(t_n, job_i)$	work cycle of t_n when carrying out job_i	8
$\sigma_i(t_n)$	event of job_i's assignment to t_n	8
$\sigma_i(t_n, o_d)$	event of t_n gaining access to o_d, while carrying out job_i	8
$\delta_i(t_n, o_d)$	event of t_n demanding access to o_d, while carrying out job_i	8
$\phi_i(t_n, o_d)$	event of t_n releasing o_d, while carrying out job_i	8
$\Phi_i(t_n)$	event of t_n releasing all objects of job_i	8
$\bullet$	an arbitrary sequence (according to context)	9
$\mathcal{G} = (V, E)$	acquisition graph	9
$\delta_{i,n}(d \mid s) \equiv \langle \delta_i(t_n, s) \mid \sigma_i(s, t_n) \rangle$	conditional demand event, for job_i and t_n	9
$\phi_{i,n}(d \mid s) \equiv \langle \phi_i(t_n, d) \mid \sigma_i(t_n, s) \rangle$	conditional release event, for job_i and t_n	8
$c[s, o_d]$	pairwise state for items s and d	10
$s.T[d]$	inter-demand period for items s and d	10
$s.D[d]$	delay for items s and d	10
$s.B[d]$	blocking period for items s and d	10
$R(s, o_d)$	pairwise request probability for items s and d	11
R	request probability matrix	11
$\mathcal{T}_s$	item s's inter-demand period	11
$\psi(\mathscr{G}) = \{c[s, o_d]\}_{(s,o_d) \in E}$	state of system $\mathcal{G} = (V, E)$	12
$\tau(\mathscr{G}) = \{\mathcal{T}_{item}\}_{item \in V \setminus \{o_M\}}$	system inter-demand periods for system state $\psi(\mathcal{G})$	12
$\psi^*(\mathcal{G})$	system state in OSE	12
f_{s,o_d}	job completion period	14

Fig. 3. A summary of the basic notation of Section 5.

define random variables for: (a) the time between consecutive (access) demand events for an object (inter-demand period), (b) the time between the consecutive event in which the thread sends a demand request for an object until it releases that object (blocking period), and (c) the time between the consecutive event in which the gains access to an object until the thread releases that object (delay). We refer to these three random variables as the pairwise state, as they give the system's performance parameters with respect to pairs of system items (Section 5.5). To the end of defining the job throughput, as well as the rate in which objects are released, we define the item inter-demand period in Section 5.6. We end the section by using the pairwise states to refine the definition of a shared-object system equilibrium (Section 5.7), which we gave in the system settings section. We give a summary of the section's basic notation in Fig. 3, for quick reference.

5.1. Acquisition paths

Suppose that the system assigns job_i to $t_n : 1 \leq n \leq N$. In this case, job_i's *(acquisition) path* is the vector, $path_{i,n} = (t_n, o_{i_1}, \ldots, o_{i_k})$, which denotes that t_n sequentially acquires $o_{i_1}, \ldots, o_{i_k}$ and then carries out job_i's operation, i.e., $operation_i$.

5.2. Work cycles: demand, supply, and release

The thread work cycle, $cycle(t_n, job_i)$, refers to the events that occur during the period that starts when the system assigns job_i to t_n and ends immediately before the next assignment of any job to t_n. It starts with the event $\sigma_i(t_n)$ in which the system assigns job_i to t_n. For each object o_ℓ of job_i's object vector, $(o_{i_1}, \ldots, o_{i_k})$, the work cycle includes the events in which t_n *demands* (requests) access to o_ℓ, denoted by $\delta_i(t_n, o_\ell)$ and the event in which the system *supplies* (provides) access to o_ℓ, denoted by $\sigma_i(t_n, o_\ell)$. Upon the event of job_i's completion t_n releases each o_ℓ, $\ell \in \{i_1, \ldots, i_k\}$, which we denote by the *release* event $\phi_i(t_n, o_\ell)$ of the work cycle. For simplicity, we refer to the sequence of these release events $\Phi_i(t_n) = (\phi_i(t_n, o_{i_1}), \ldots, \phi_i(t_n, o_{i_k}))$ as a single event and assume that t_n releases all its acquired objects instantaneously and immediately after the operation time, O_i, which is a known random variable. Immediately after the event $\Phi_i(t_n)$, the thread work cycle starts a (possibly zero length) *idle period*, before the system assigns the next (and possibly different than the previous) job to t_n so that the next work cycle begins.

We illustrate the thread work cycle in Fig. 4. We assume that events are instantaneous and mark them as points on a thread's work cycle. Note however, that between a supply event and the next demand event (as well as between the last supply event and the release event), there is a random length period which refers to scheduling uncertainties, i.e. the (random) acquisition period A, which follows a known distribution (and the operation time O_i, respectively). Hence, we denote t_n's work cycle for job_i as in Equation (1):

$$cycle(t_n, job_i) \equiv (\sigma_i(t_n), \delta_i(t_n, o_{i_1}), \sigma_i(t_n, o_{i_1}), \ldots, \delta_i(t_n, o_{i_k}), \sigma_i(t_n, o_{i_k}), \Phi_i(t_n)) \tag{1}$$

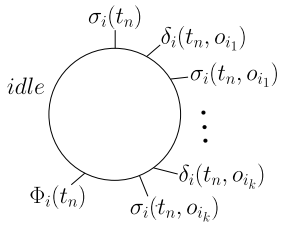


Fig. 4. The work cycle of t_n for job_i, with object set $objs_i = (o_{i_1}, \ldots, o_{i_k})$. After an idle period job_i is assigned to t_n ($\sigma_i(t_n)$), which acquires job_i's objects ($\delta_i(t_n, o_{i_1})$ to $\sigma_i(t_n, o_{i_k})$), executes job_i's operation ($\sigma_i(t_n, o_{i_k})$ to $\Phi_i(t_n)$), and then releases job_i's objects ($\Phi_i(t_n)$).

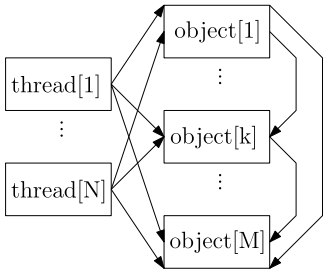


Fig. 5. An acquisition graph, $\mathscr{G}$.

5.3. Subpaths and acquisition graph

Let s (source) and d (destination) be (possibly consecutive) items on a path. We define the set $\epsilon(s, o_d) = \{(s, \bullet, o_d) | \exists job_i : path_i = (\bullet, s, \bullet, o_d, \bullet)\}$ of all *subpaths* between s and o_d, where $\bullet$ denotes a finite, possibly empty, item sequence. The *acquisition graph*, $\mathscr{G} = (V, E)$, is a simple directed graph, where $V = \{t_1, \ldots, t_N, o_1, \ldots, o_M\}$ are the system items (Fig. 5). The edges $E = \{(s, o_d) \mid \exists job_i : path_i = (\bullet, s, o_d, \bullet)\}$ are two consecutive items on a path. That is, $\mathscr{G}$ includes an edge for two system items if and only if there exists a job, such that these two items are consecutive in its path, e.g., a thread and the first job object or two consecutive objects of a job.

5.4. Conditional and consecutive events

In this section, we consider an event that occurs at item d (destination) in condition to an event occurrence at item s (source), where $(\bullet, s, o_d, \bullet)$ is a subpath of job_i and both events belong to the same work cycle of job_i that t_n carries out. We define conditional events, which capture the dependencies within a work cycle, as well as consecutive events, which capture the dependencies between consecutive work cycles. These definitions are the basis for modeling the system's state in Section 5.5.

5.4.1. Conditional demand and supply events

Consider a subsequence $(\sigma_i(t_n, s), \delta_i(t_n, o_d))$ of $cycle(t_n, job_i)$'s events. We denote by $\delta_{i,n}(o_d \mid s) \equiv \langle \delta_i(t_n, o_d) \mid \sigma_i(t_n, s) \rangle$ the *conditional (demand) event*, $\delta_i(t_n, o_d)$, in which t_n requests access to o_d immediately after the supply event, $\sigma_i(t_n, s)$, in a timing order (i.e. in the actual sequence in which these events occur). Note that the event $\sigma_i(t_n, s)$ may refer to: (1) access to object s, or (2) job_i's assignment to $s = t_n$ (Fig. 4), i.e., $\sigma_i(t_n) \equiv \sigma_i(t_n, t_n)$. E.g., $\delta_{i,n}(o_j \mid o_k)$ denotes the conditional demand event $\langle \delta_i(t_n, o_j) \mid \sigma_i(t_n, o_k) \rangle$ in which t_n requests access to o_j immediately after gaining access to o_k, where $k \in [1, M-1]$, $j \in (k, M]$, and $n \in [1, N]$ (i.e., o_k has lower index than o_j). Another example is the case $\delta_{i,n}(o_j \mid t_n)$, where the conditional demand event $\langle \delta_i(t_n, o_j) \mid \sigma_i(t_n, t_n) \rangle$ refers to t_n's request to access job_i's first object, o_j, immediately after the assignment of job_i to t_n. In a similar manner, denote by $\phi_{i,n}(o_d|s) \equiv \langle \phi_i(t_n, o_d) \mid \sigma_i(t_n, s) \rangle$ the *conditional (release) event*, in which t_n releases o_d at event $\phi_i(t_n, o_d)$ that occurs after the supply event, $\sigma_i(t_n, s)$ and at the same work cycle (we will mainly use $\phi_{i,n}(s|s)$).

5.4.2. Events of arbitrary jobs and threads

In some parts of our modeling, we consider an arbitrary job_i that an arbitrary t_n carries out. We then write $\delta(o_d|s)$, $\sigma(s)$ and $\phi(o_d|s)$ instead of $\delta_{\bullet,\bullet}(o_d|s)$, $\sigma_{\bullet,\bullet}(s)$, and respectively, $\phi_{\bullet,\bullet}(o_d|s)$ when referring to events from the sets $\{\delta_{i,n}(o_d|s) : i \in [1, J], n \in [1, N]\}$, $\{\sigma_{i,n}(s) : i \in [1, J], n \in [1, N]\}$, and respectively, $\{\phi_{i,n}(o_d|s) : i \in [1, J], n \in [1, N]\}$, where $i \in [1, J]$ and $n \in [1, N]$. Given subpath $(\bullet, \ell_1, \ldots, \ell_k, \bullet)$, denote by $\delta(\ell_k|\ell_1, \ldots, \ell_{k-1})$ the occurrence of $\delta(\ell_k|\ell_{k-1})$, which happens immediately after $\delta(\ell_{k-1}|\ell_{k-2}), \ldots, \delta(\ell_2|\ell_1)$.

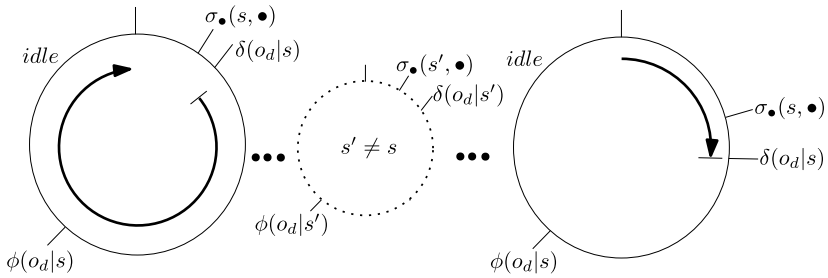


Fig. 6. (s, o_d)'s inter-demand period, $s.T[d]$, i.e., the period between two consecutive $\delta(o_d|s)$ events.

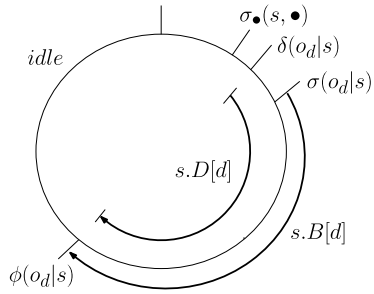


Fig. 7. The delay $s.D[d]$ and blocking $s.B[d]$ start from $\delta(o_d|s)$, and respectively, $\sigma(o_d|s)$, and both end at $\phi(o_d|s)$.

5.4.3. Consecutive events

To the end of defining throughput (inter-demand period) of a system item s in Section 5.5, we define consecutive demand events that follow supply events of s. We say that $\delta_{i,n}(o_k|s)$ *occurs consecutively after* $\delta_{j,n'}(o_\ell|s)$, when all the following conditions hold:

(1) $\delta_{j,n'}(o_\ell \mid s) = \langle \delta_j(t_{n'}, o_\ell) \mid \sigma_j(t_{n'}, s)\rangle$ is part of $t_{n'}$'s work cycle that includes the event $\sigma_j(t_{n'}, s)$ and the event $\delta_j(t_{n'}, o_\ell)$ in which after s's supply, $t_{n'}$ requests access to o_ℓ,
(2) $\phi_j(t_{n'}, s)$ in which $t_{n'}$ releases item s, and
(3) $\delta_{i,n}(o_k \mid s) = \langle \delta_i(t_n, o_k) \mid \sigma_i(t_n, s)\rangle$ is part of t_n's work cycle (which is the successive of $cycle(t_{n'}, job_j)$), which includes the events $\sigma_i(t_n, s)$ and $\delta_i(t_n, o_k)$, in which after the supply of s, t_n requests access to o_k.

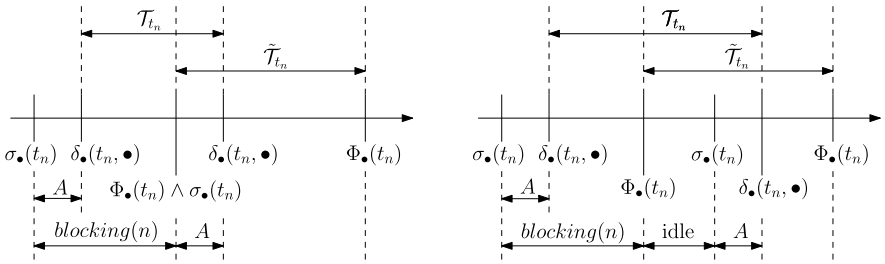
5.5. Pairwise states and request probabilities

The definition of the studied equilibria (at the system level) is based on item-level definitions that consider $\mathscr{G}$'s edges, $(s, o_d) \in E$. We present a definition of the (pairwise) state, $c[s, o_d]$, which considers the delay, blocking and inter-demand periods that are related to the edge (s, o_d) of $\mathscr{G}$ and its conditional events. These periods refer to the time it takes threads to request access to o_d, and release it subsequently (after the acquisition of item s) as well as the time between such (consecutive) requests that are made by (possibly) different threads. Moreover, when estimating the value of the pairwise-state, $c[s, o_d]$, we need to consider the probabilities that are related to the edge (s, o_d) and its conditional events.

5.5.1. Pairwise states

To the end of approximating $\tilde{\mathcal{T}}_{t_n}$, $\tilde{D}_{t_n}$, and $blocking(n)$, for $n \in [1, N]$, we define the pairwise states, which capture the throughput, delay, and blocking between t_n and every system object. To compute the latter, we extend the definition of pairwise states to pairs of objects, since jobs might include more than one object. In sections 7–8 we will show how we can approximate $\tilde{\mathcal{T}}_{t_n}$, $\tilde{D}_{t_n}$, and $blocking(n)$ through the pairwise states.

We refer to (s, d)*'s pairwise state* $c[s, o_d] = s.\langle T[d], D[d], B[d]\rangle$ as the tuple that includes the request completion rate, request delay, and respectively, blocking period with relation to the events $\delta(o_d|s)$, where $d \in [1, M]$ refers to o_d and s can be either a thread or o_ℓ, $\ell \in [1, d)$. The (s, o_d)*'s request inter-demand period*, $s.T[d]$, refers to the period between the consecutive events, $\langle \delta_i(t_n, o_d) \mid \sigma_i(t_n, s)\rangle$ and $\langle \delta_j(t_{n'}, o_d) \mid \sigma_j(t_{n'}, s)\rangle$ (Fig. 6), where $i, j \in [1, J]$ and $n, n' \in [1, N]$. Furthermore, (s, o_d)*'s delay*, $s.D[d]$, refers to a period that starts on the event $\delta(o_d|s)$ in which a thread requests access to o_d (immediately after gaining access to item s) and ends upon the event $\phi(o_d|s)$ in which that thread releases o_d (during the same work cycle). In addition, (s, o_d)'s blocking, $s.B[d]$, is the fraction of (s, o_d)'s delay $s.D[d]$ in which the thread blocks other threads from gaining access to o_d, i.e., the period between the event $\langle \sigma_i(t_n, o_d)|\sigma_i(t_n, s)\rangle$ in which t_n gains access to o_d, and the event $\phi_i(t_n, o_d)$ in which t_n releases o_d (Fig. 7). Note that each of $c[s, o_d]$'s three elements is a random variable (for which maintaining the first three moments provides sufficient accuracy).



(a) No idle period after job completion. (b) Idle period after the job completion.

Fig. 8. Illustration of the fact that $\mathcal{T}_{t_n}$ (inter-demand period) and $\tilde{\mathcal{T}}_{t_n}$ (time between consecutive job completions) follow the same distribution. In both figures, the horizontal line denotes the time horizon, while the vertical lines mark events that occurred, which are denoted below them. $\mathcal{T}_{t_n}$ starts with the first demand event ($\delta_\bullet(t_n,\bullet)$) that follows the job assignment ($\sigma_\bullet(t_n)$) and it continues until the job completion ($\Phi_\bullet(t_n)$). Then there is a possibly zero length idle period (Fig. 8b) until the next job assignment ($\sigma_\bullet(t_n)$), and $\mathcal{T}_{t_n}$ ends at the event $\delta_\bullet(t_n,\bullet)$. Similarly, $\tilde{\mathcal{T}}_{t_n}$ starts at the event of job completion ($\Phi_\bullet(t_n)$), which is followed by a possibly zero length idle period, and ends upon the job completion of the next job ($\sigma_\bullet(t_n)$ to $\Phi_\bullet(t_n)$).

5.5.2. Pairwise request probabilities

An essential component of our analysis is the probability of an event $\delta(o_d|s)$ to occur, i.e., the probability of a demand event to object o_d to occur immediately after the supply event of item s, within the same work cycle. Given that a thread has gained access to item s, i.e., $\sigma_\bullet(t_\bullet, s)$ occurred, then either $\delta_\bullet(t_\bullet, o_d)$ will occur immediately after, or the thread will release s in case the job is completed. We refer to the probability of $\delta(o_d|s)$ to occur as the *(pairwise) request probability*, $R(s, o_d)$.

The exact values of these probabilities can be computed when we are aware of all the events that occurred during a time horizon t. In this case, we denote these probabilities with $R_t(s, o_d)$. When the information about the number of system events during the horizon t is not available or we consider the values of these probabilities independently of t, we use the notation $R(s, o_d)$. In the following we define $R(s, o_d)$ and $R_t(s, o_d)$. In this work, we do not focus on a specific horizon t neither we assume that we have knowledge of the number of events that have occurred during some t. Hence, in Section 6 we propose an approximation of $R(s, o_d)$, based on the system's input parameters.

Definition of $R(s, o_d)$ For a randomly chosen work cycle that includes the event, $\sigma_\bullet(t_\bullet, s)$, of a thread gaining access to item s, we define $\Omega(s) = \{\delta(o_d|s) : (s, o_d) \in E$ is an edge in $\mathcal{G}\} \cup \{\phi(s|s)\}$ as the probability space of the possible events to occur immediately after $\sigma_\bullet(t_\bullet, s)$. Moreover, $R(s, o_d)$ and $R(s, s)$ are the probabilities of $\Omega(s)$'s events $\delta(o_d|s)$, and respectively, $\phi(s|s)$. Namely, $R(s, o_d)$ denotes the probability of a demand event, $\delta_\bullet(t_\bullet, o_d)$ to occur immediately after the supply event, $\sigma_\bullet(t_\bullet, s)$ and in the same (randomly chosen) work cycle. Moreover, $R(s, s)$ denotes the probability of a release event, $\phi_\bullet(t_\bullet, s)$, to occur immediately after its related supply event, $\sigma_\bullet(t_\bullet, s)$ and during the same (randomly chosen) work cycle. Note that $R(s, s)$'s definition requires that s is the last object to which a thread gains access during its work cycle.

We define the *request probability matrix* R to be a $(N+M) \times (N+M)$ row stochastic matrix, such that the (s, o_d) element of R is the probability $R(s, o_d)$. Note that by the definition of the state space, $\Omega(s)$, the following hold: (i) for every two items s and x, such that (s, x) is not an edge of $\mathcal{G}$, $R(s, x) = 0$ holds, and (ii) $R(s, s) + \Sigma_d R(s, o_d) = 1$ holds. The matrix R has a block form (Equation (2)), where $R_{N,N}$ is an $N \times N$ zero matrix, $R_{N,M} = (R(t_n, o_j))_{n\in[1,N], j\in[1,M]}$ is an $(N \times M)$ matrix, $R_{M,N}$ is an $M \times N$ zero matrix, and $R_{M,M} = (R(o_j, o_k))_{j,k\in[1,M]}$ is an $M \times M$ matrix. Note that $R_{M,M}$ is an upper triangular matrix, since the acquisition order of objects (implied by the ascending index order of $objs_i$ in $job_i = \langle objs_i, operation_i\rangle$) restricts each thread to request an object of higher index than the one of the last object that it gained access to.

$$R = \left(\begin{array}{c|c} R_{N,N} & R_{N,M} \\ \hline R_{M,N} & R_{M,M} \end{array}\right) \tag{2}$$

Definition of $R_t(s, o_d)$ In this case, we restrict the (random) choice of the work cycle to the time interval $t = [t_{start}, t_{end}]$ and assume the knowledge of all events that occurred in the system during that period. For the time interval $t = [t_{start}, t_{end}]$, we define $R_t(s, o_d) = \alpha_t(s, o_d)/\eta_t(s)$, to be the number of $\delta(o_d|s)$ occurrences, over the number of $\sigma(s)$ occurrences and $R_t(s, s) = \beta_t(s)/\eta_t(s)$ to be the number of $\phi(s|s)$ occurrences, over the number of $\sigma(s)$ occurrences, where $\#_t X$ denotes the number of event X occurrences in $t = [t_{start}, t_{end}]$. Moreover, we define (i) $\alpha_t(s, o_d) = \Sigma_{i,n}\#_t\delta_{i,n}(o_d|s)$, when $s = o_\ell$, $\ell \in [1, d)$, and (ii) $\alpha_t(s, o_d) = \Sigma_i\#_t\delta_{i,n}(o_d|s)$, when $s = t_n : n \in [1, N]$, the number of $\delta(o_d|s)$ events that occurred during $t = [t_{start}, t_{end}]$. Furthermore, $\beta_t(s) = \Sigma_{i,n}\#_t\phi_{i,n}(s|s)$ and $\eta_t(s) = \Sigma_{i,n}\#_t\sigma_i(t_n, s)$ denote the number of $\phi(s|s)$, and respectively, $\sigma(s)$ events that occurred during $t = [t_{start}, t_{end}]$. Note that profiling tools can be the basis for estimating $R_t(s, o_d)$ and $R_t(s, s)$.

5.6. Item inter-demand period

To the end of defining the job completion rate (throughput), as well as the rate in which an object is released, we define the item inter-demand period. Consider job_i and $job_{i'}$, where $i, i' \in [1, J]$, such that o_{i_1} and $o_{i'_1}$ are the first objects in the

object vectors of job_i, and respectively, $job_{i'}$, and assume that t_n carries out job_i and $job_{i'}$ consecutively. We refer to t_n's *inter-demand period*, $\mathcal{T}_{t_n}$, as the period between $\delta_i(t_n, o_{i_1})$ and the demand event $\delta_{i'}(t_n, o_{i'_1})$.

We remark that the time between (consecutive) job completions of t_n, $\tilde{\mathcal{T}}_{t_n}$ (Section 2.3), follows the same distribution with the thread inter-demand period, $\mathcal{T}_{t_n}$ (essentially, the time between consecutive job assignments). We illustrate this point in Fig. 8, as well as in the following. Recall that $blocking(n)$ denotes the job blocking period for t_n (Section 2.4), i.e., the time between the assignment and completion of a job at t_n. Also, recall that $\tilde{D}_{t_n}$ denotes the job delay for t_n (Section 2.4), i.e., the time between a job arrival to $Q_{thread}(t_n)$ and the completion of that job. When there is an idle period after a job completion (t_n's queue is empty), upon the next job arrival to t_n's queue, the job is immediately assigned to t_n. In the latter case, the job delay and blocking period are equal. Let idl_{t_n} be the random variable that denotes the length of t_n's idle periods ($idl_{t_n} = 0$ when there is no idle period between consecutive jobs).

By the definition of the work cycle events (Section 5.2), $\tilde{\mathcal{T}}_{t_n}$ (time between job completions) follows the distribution of $idl_{t_n} + blocking(n)$. That is, after a job completion event, there is an idle period (of possibly zero length), hence idl_{t_n}. In case t_n's queue is empty, then the next job that arrives is immediately assigned to t_n, hence the next job completion after the idle period occurs after a period equal to $blocking(n)$. In case t_n's queue is not empty, then the next job is immediately assigned to t_n, hence the time until the next job completion is again $blocking(n)$. Similarly, $\mathcal{T}_{t_n}$ follows the distribution of $(blocking(n) - A) + idl_{t_n} + A = blocking(n) + idl_{t_n}$. That is, $\mathcal{T}_{t_n}$ starts with a demand event for the first object of a job, hence there is a period of $blocking(n) - A$ until t_n completes that job. Then, there is an idle period of length idl_{t_n}, until the next job assignment. After the next job assignment, there is a period of length A (acquisition period) until the demand event for the first object of that job.

We refer to o_k's *inter-demand period*, $\mathcal{T}_{o_k}$, as the period between two consecutive conditional (demand) events for accessing an object in $\{o_\ell : \ell \in (k, M)\}$ immediately after gaining access to o_k by two, possibly different, threads, where $k \in [1, M-1]$. Namely, $\mathcal{T}_{o_k}$ is the period between $\delta(o_j|o_k)$ and the successive event $\delta(o_{j'}|o_k)$, where $j, j' \in (k, M]$. Similarly to $\mathcal{T}_{t_n}$, $1/E(\mathcal{T}_{o_d})$ estimates (due to using the expected value of $\mathcal{T}_{o_d}$) the number of job completions that include o_d per time unit.

Note that in systems where jobs with paths $(\bullet, o_d, o_{d'}, \bullet)$ do not exist, we assume that $\mathcal{T}_{o_d}$ is the time between two consecutive $\sigma_\bullet(\bullet, o_d)$ events (which definitely occur due to our assumptions in Section 2). Our definition of $\mathcal{T}_s$ for a system item s, will play a crucial role in resolving dependencies in Section 9 (cf. Section 4). Specifically, it will allow us to define subsystems that include less system items (hence the focus on the time between requests), that preserve the shared-object system's performance parameters.

5.7. Shared-object system equilibria

In this section, we refine the definitions of system state and OSEs given in Section 2.4, basing on the pairwise states. For a given system, $\psi(\mathcal{G}) = \{c[s, o_d]\}_{(s,o_d)\in E}$ is the *system state (set)*, where $\mathcal{G} = (V, E)$ is the acquisition graph. For a given $\psi(\mathcal{G})$, the *inter-demand period of the system* is the set $\tau(\mathcal{G}) = \{\mathcal{T}_{item}\}_{item\in V\setminus\{o_M\}}$.

Suppose that the system is in a state in which $\Sigma_{i\in[1,J]}\lambda_{i,n} = 1/E[\mathcal{T}_{t_n}]$, for every $n \in [1, N]$. That is, the job arrival and completion rates match for every thread's queue. We say that $\psi^*(\mathcal{G}) = \{c^*[s, d]\}_{(s,o_d)\in E}$, is the *shared-Object System Equilibrium* (OSE). Given $\psi^*(\mathcal{G})$, the respective inter-demand period of the system is $\tau^*(\mathcal{G}) = \{\mathcal{T}^*_{item}\}_{item\in V\setminus\{o_M\}}$. For a given $\varepsilon > 0$ and an OSE $\psi^*(\mathcal{G})$, we say that the system state $\psi(\mathcal{G})$ is an *ε-OSE* when $\forall item \in V \setminus \{o_M\}$, $\mathcal{T}^*_{item} \in \tau^*(\mathcal{G})$, $\mathcal{T}_{item} \in \tau(\mathcal{G}) : |\mathcal{T}^*_{item} - \mathcal{T}_{item}| < \varepsilon$ holds. Namely, the corresponding values of each item in $\tau(\mathcal{G})$ and $\tau^*(\mathcal{G})$ differ by less than ε.

6. Request probabilities

In this section we provide approximations of the pairwise request probabilities $R(s, o_d)$ (Section 5.5.2). Recall that $R(s, o_d)$ denotes the probability of a thread to demand access to o_d, immediately after the supply event $\sigma_\bullet(\bullet, s)$ (the latter denotes either a job assignment event or the event of a thread gaining access to object s). Moreover, $R(s, s)$ denotes the probability of a job to be completed at object s.

The fact that the pairwise request probabilities depend on the arrival rates of the corresponding jobs is the basis of our estimation of $R(s, o_d)$ (Approximation 1), where (s, o_d) is an edge in the acquisition graph $\mathcal{G}$. That is, a higher arrival rate of jobs that include s and o_d in their acquisition paths, increases the probability of $\delta_{\bullet,\bullet}(s, o_d)$ events to occur (Section 5.4.2). We follow the approach of network approximations [8, Section 8.2] for estimating the probabilities $R(s, o_d)$. In network approximations, queuing systems (Section 3) with arrivals that follow exponential distributions and jobs that have deterministic routing, but exponentially distributed service times, are approximated by Jackson queuing networks (for which an exact analysis is feasible) [15,5]. This method cannot be applied entirely for the analysis of shared-object systems, since it considers exponentially distributed blocking periods and it does not consider thread blocking. However, as mentioned in [8, Section 8.2], combinations of multiple (non-exponentially distributed) arrivals to a system item's queue, tend to be exponentially distributed.

In Lemma 2 we prove that our estimations of $R(s, o_d)$ and $R(s, s)$ define indeed a probability, i.e., for any item s, $R(s, s) + \Sigma_{d\neq s}R(s, o_d) = 1$. This implies that the probability matrix R, which contains the estimates of $R(s, o_d)$ and $R(s, s)$, is a

stochastic matrix. Let $\langle s_1, \ldots, s_\ell \rangle$ be a vector of objects, $objs_i$ be the object vector of job_i and $i \in [1, J]$. We define the following characteristic functions with values in $\{0, 1\}$:

$$starts_i(\langle s_1, \ldots, s_\ell \rangle) \Leftrightarrow objs_i = (s_1, \ldots, s_\ell, \bullet) \tag{3}$$

$$includes_i(\langle s_1, \ldots, s_\ell \rangle) \Leftrightarrow objs_i = (\bullet, s_1, \ldots, s_\ell, \bullet) \tag{4}$$

$$ends_i(\langle s_1, \ldots, s_\ell \rangle) \Leftrightarrow objs_i = (\bullet, s_1, \ldots, s_\ell) \tag{5}$$

Approximation 1. *Equation (6) and Equation (7) approximate $R(s, o_d)$, when $s = t_n : n \in [1, N]$, and respectively, $s = o_j : j \in [1, M-1]$. Moreover, for any object s, Equation (8) approximates $R(s, s)$.*

$$R(t_n, o_d) \approx \frac{\left(\Sigma_i \lambda_{i,n} \cdot starts_i(\langle o_d \rangle)\right)}{\left(\Sigma_i \lambda_{i,n}\right)} \tag{6}$$

$$R(o_j, o_d) \approx \frac{\Sigma_{i,n} \lambda_{i,n} \cdot includes_i(\langle o_j, o_d \rangle)}{\Sigma_{i,n} \lambda_{i,n} \cdot includes_i(\langle o_j \rangle)} \tag{7}$$

$$R(s, s) \approx \frac{\left(\Sigma_{i,n} \lambda_{i,n} \cdot ends_i(\langle s \rangle)\right)}{\left(\Sigma_{i,n} \lambda_{i,n} \cdot includes_i(\langle s \rangle)\right)} \tag{8}$$

Equation (6) estimates $R(t_n, o_d)$ by the sum of arrival rates of jobs that start with o_d at t_n, divided by the sum of the arrival rates of all the jobs that are assigned to t_n. Equation (7) estimates $R(o_j, o_d)$ by the sum of arrival rates of jobs that include the subvector (o_j, o_d) in their object vector (assigned to any thread), divided by the sum of arrival rates of jobs that include the subvector (o_j) in their object vector (to any thread), where $j \in [1, M-1]$ and $d \in (j, M]$. Equation (8) estimates $R(s, s)$ by the sum of arrival rates of jobs with object vectors that end with the subvector (s) (assigned to any thread), divided by the sum of arrival rates of jobs that include the subvector (s) in their object vector (assigned to any thread), where s is a system object. Note that in any other case, we define $R(s, o_d) = 0$, since no conditional demand events $\delta(o_d|s)$ or conditional release events $\phi(s|s)$ occur in these cases.

We prove that the estimation of the request probabilities in Approximation 1 also defines a probability (Lemma 2).

Lemma 2. *For $R(s, o_d)$'s estimation (Approximation 1), it holds that: (1) $\Sigma_d R(t_n, o_d) = 1$, where $n \in [1, N]$ and o_d is an object, (2) $R(s, s) + \Sigma_{d \neq s} R(s, o_d) = 1$, and (3) R is a row stochastic matrix.*

Proof. Equation (9) demonstrates claim (1).

$$\begin{aligned}
\Sigma_d R(t_n, o_d) &= \frac{\Sigma_d \Sigma_i \lambda_{n,i} \cdot starts_i(\langle o_d \rangle)}{\Sigma_i \lambda_{n,i}} \\
&= \frac{\Sigma_i \lambda_{n,i} \Sigma_d \, starts_i(\langle o_d \rangle)}{\Sigma_i \lambda_{n,i}} \\
&= \frac{\Sigma_i \lambda_{n,i} \cdot 1}{\Sigma_i \lambda_{n,i}} \\
&= 1
\end{aligned} \tag{9}$$

For any object $s \neq o_M$, Equation (11) demonstrates claim (2) due to Equation (10), which holds since a job object vector that includes object s, either ends with s or includes more items $o_d \neq s$.

$$ends_i(\langle s \rangle) + \Sigma_{d \neq s} includes_i(\langle s, o_d \rangle) = includes_i(\langle s \rangle) \tag{10}$$

$$\begin{aligned}
R(s, s) + \Sigma_{d \neq s} R(s, o_d) &= \frac{\Sigma_{n,i} \lambda_{n,i} \cdot ends_i(\langle s \rangle)}{\Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s \rangle)} + \frac{\Sigma_{d \neq s} \Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s, o_d \rangle)}{\Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s \rangle)} \\
&= \frac{\Sigma_{n,i} \lambda_{n,i} \cdot ends_i(\langle s \rangle) + \Sigma_{d \neq s} \Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s, o_d \rangle)}{\Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s \rangle)} \\
&= \frac{\Sigma_{n,i} \lambda_{n,i} \left(ends_i(\langle s \rangle) + \Sigma_{d \neq s} includes_i(\langle s, o_d \rangle)\right)}{\Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s \rangle)} \\
&= \frac{\Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s \rangle)}{\Sigma_{n,i} \lambda_{n,i} \cdot includes_i(\langle s \rangle)} \\
&= 1
\end{aligned} \tag{11}$$

As for claim (3), we note that claims (1) and (2) imply that our estimation of the block matrices $R_{N,M}$ and $R_{M,M}$ of the matrix R are row stochastic (Equation (2)). Since the block matrices $R_{N,N}$ and $R_{M,N}$ are zero matrices, our estimation of R forms a row stochastic matrix. □

7. Blocking periods

In this section we estimate the blocking periods of the pairwise states (Section 5.5.1), i.e., the time between the events of supply and release of access to an object. Recall that the supply event of item s, may refer to either the assignment of a job to $s = t_n$ or the event in which a thread acquires access to $s = o_j$. We estimate the blocking period $s.B[d]$, i.e., the time from the event in which a thread acquires access to o_d immediately after the occurrence of a supply event of item s, until that thread releases o_d.

The $s.B[d]$ blocking period is an effect of multiple threads' job paths, i.e., the $(\bullet, s, o_d)$ and the remaining $(\bullet, s, o_d, \bullet)$ paths. The former case corresponds to the job completion period (Approximation 3), whereas the latter also includes the dependency on the delay of acquiring the path's remaining objects (Approximation 4). Our approximations follow the approach of [23]. In Section 9 we show that the computation of the pairwise states can be indeed based on tools from [23].

7.1. Job completion periods

Let f_{s,o_d} denote the fraction of the blocking period that refers to jobs with paths $(\bullet, s, o_d)$. That is, f_{s,o_d} is the time between the supply event $\sigma_{\bullet,\bullet}(\bullet, o_d)$ and the release event $\phi_{\bullet,\bullet}(\bullet, o_d)$ of the same work cycle, for jobs with paths $(\bullet, s, o_d)$ (cf. Section 5.4.2). We estimate f_{s,o_d} by the operation time average of jobs with $(\bullet, s, o_d)$ paths, weighted by the probability for the related events to occur (Approximation 3). Approximation 3 defines $f_{s,o_d} = \Sigma_{i=1}^{J} weight_i(s, o_d) \cdot O_i$ as a weighted average of all the operation times. We define these weights as follows. If s is an object, then Equation (13) defines $weight_i(s, o_d)$ and Equation (12) defines the normalizing constant for all weights.

$$W_{s,o_d} = \Sigma_{i=1}^{J} \Sigma_{n=1}^{N} \lambda_{i,n} \cdot ends_i(\langle s, o_d \rangle) \tag{12}$$

$$weight_i(s, o_d) = (\Sigma_{n=1}^{N} \lambda_{i,n} \cdot ends_i(\langle s, o_d \rangle))/W_{s,o_d} \tag{13}$$

Similarly, if s is a thread, t_n, then Equation (15) defines $weight_i(s, o_d)$ and Equation (14) defines the normalizing constant for all weights.

$$W_{s,o_d} = \Sigma_{i=1}^{J} \lambda_{i,n} \cdot ends_i(\langle o_d \rangle) \tag{14}$$

$$weight_i(s, o_d) = (\lambda_{i,n} \cdot ends_i(\langle o_d \rangle))/W_{s,o_d} \tag{15}$$

Approximation 3 *(as in [23]). The blocking period of jobs with paths $(\bullet, s, o_d)$ is a weighted average, $f_{s,o_d} = \Sigma_{i=1}^{J} weight_i(s, o_d) \cdot O_i$, of the respective job operation times.*

7.2. Acquiring the remaining objects

In the case of $(\bullet, s, o_d, \bullet)$ paths, the blocking period $s.B[d]$ depends on the completion period of jobs with $(\bullet, s, o_d)$ paths, f_{s,o_d}, as well as on the jobs with $(\bullet, s, o_d, o_{d'}, \bullet)$ paths. In the case of jobs with $(\bullet, s, o_d, o_{d'}, \bullet)$ paths, the blocking period of o_d equals the delay for $o_{d'}$. That is, the time between the events of acquiring and releasing access to o_d, equals the time between the events of queuing an acquisition request in the queue of $o_{d'}$ and the release of $o_{d'}$ (as well as the other objects of that job). Since $d' \in (d+1, M]$, Approximation 4 is an average over all possible choices of d', weighted by the pairwise probabilities, as in [23].

Approximation 4 *(as in [23]). The blocking period of jobs with $(\bullet, s, o_d, \bullet)$ paths can be estimated by $s.B[d] = R(s, o_d) \cdot R(o_d, o_d) \cdot f_{s,o_d} + \Sigma_{d'=d+1}^{M} R(s, o_d) \cdot R(o_d, o_{d'}) \cdot (A + o_d.D[d'])$.*

Note that Approximation 4's recursive equation details a forward dependency on the delays $o_d.[d']$, $d \in (d+1, M]$. Thus, it is possible to compute this estimation for $d = M$ directly from the input parameters (sections 6 and 7.1), i.e., $s.B[M] = R(s, o_M) \cdot R(o_M, o_M) \cdot f_{s,o_M}$. Then, given the delays $s.D[M]$, it is possible to compute the blocking periods $s.B[M-1]$, and so on until $s.B[1]$. This computation is the basis of the backward iterations (Section 4) to resolve these dependencies (Section 9).

8. Item inter-demand periods

The inter-demand period of an item s, $\mathcal{T}_s$, is the time between two consecutive demand requests that follow the supply of item s (Section 5.6). In this section, we approximate the item inter-demand period, which together with the blocking period, $s.B[d]$ (Section 7), are essential for calculating the delay, $s.D[d]$ (Section 9), where $d \in [1, M]$ and (s, o_d) is an edge in the acquisition graph $\mathcal{G}$. In Section 8.1 we estimate $\mathcal{T}_{t_n}$, for every thread t_n and in Section 8.2 we estimate $\mathcal{T}_{o_d}$, for every object o_d. The estimation of $\mathcal{T}_{t_n}$ depends on the job arrival process and the job blocking period, and the estimation of $\mathcal{T}_{o_d}$ bases on the pairwise inter-demand periods, $s.T[d]$, where s is a system item.

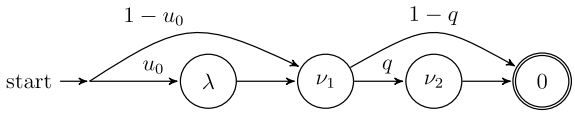


Fig. 9. The Markov process that models $\tilde{\mathcal{T}}_{t_n}$. The process starts after a job completion on t_n. If $Q_{thread}(t_n)$ is empty (with steady-state probability u_0) the next job arrives with rate λ. After a job has arrived or in case $Q_{thread}(t_n)$ is not empty (which occurs with probability $1-u_0$), the blocking period begins. The blocking period follows a Coxian-2 distribution, and thus it consists of two states with rates ν_1 and ν_2. The blocking period ends with rate ν_1 with probability $1-q$, by moving to the absorbing (ending) state, marked with 0. Otherwise, the blocking period continues to the second state (of rate ν_2) with probability q, after which it also moves to the absorbing state.

8.1. Thread inter-demand period

We estimate the inter-demand period of a thread t_n, $\mathcal{T}_{t_n}$, through a Markov process, following the approach of [23]. Recall that $\mathcal{T}_{t_n}$ follows the same distribution with the time between job completions, $\tilde{\mathcal{T}}_{t_n}$ (Fig. 8, Section 5.6). Thus, we focus on computing the first three moments of $\tilde{\mathcal{T}}_{t_n}$. We achieve the latter, by defining a Markov process which describes $\tilde{\mathcal{T}}_{t_n}$, such that $\tilde{\mathcal{T}}_{t_n}$ follows the distribution of the time until the Markov process reaches an absorption (ending) state [23]. By the definition of $\tilde{\mathcal{T}}_{t_n}$, the absorbing state is the event of job completion. In Fig. 9 we define a Markov process that models $\tilde{\mathcal{T}}_{t_n}$, which we explain below.

The Markov process of Fig. 9 is defined as follows. By the definition of $\tilde{\mathcal{T}}_{t_n}$, the process starts upon a job completion event (and ends in the next job completion event). Hence, in the beginning of the process the queue of t_n is empty with steady-state probability u_0 (Section 3.1), or non-empty (with probability $1-u_0$), and in the latter case the blocking period of the next job begins. Thus, if t_n's queue is empty, the next job arrives with rate $\lambda = \Sigma_{i=1}^{J}\lambda_{i,n}$ (first state in Fig. 9), and the blocking period begins immediately after the job arrival.

Since the distribution of the blocking period, $blocking(n)$, is unknown, we use its moments to match it with a Coxian-2 distribution (Section 3.1 and [2]). We denote the parameters of the Coxian-2 distribution that matches the moments of $blocking(n)$ with ν_1, q, and ν_2 (Section 3.1). By the definition of Coxian-2 distributions, the blocking period either ends after the state with rate ν_1 with probability q, or it continues (with probability $1-q$) to the state with rate ν_2, after which it certainly ends. The end of the blocking period (job completion) sets the end of $\tilde{\mathcal{T}}_{t_n}$'s period, which is modeled by the absorbing (ending) state of the Markov process (state 0 in Fig. 9).

Recall that $\tilde{\mathcal{T}}_{t_n}$ follows the distribution of the time until the Markov process of Fig. 9 reaches an absorption state. The latter time follows a phase-type distribution, by the distribution's definition [1,21]. In the following we define the parameters of $\tilde{\mathcal{T}}_{t_n}$'s phase-type distribution, based on the Markov process of Fig. 9, which will then allow us to compute $\tilde{\mathcal{T}}_{t_n}$'s moments.

A phase-type distribution is defined by an initial probability vector c (probability of starting in each state) and a transition rate matrix Q (rates of transitions between states), both determined by the Markov process. By Fig. 9, $c = (u_0, 1-u_0, 0, 0)$, where each of the elements corresponds to the probability of starting in each of the states of the Markov process in the following order: λ, ν_1, ν_2, and 0. Similarly, the matrix Q is a 4×4 matrix with block form $Q = \left[\begin{array}{c|c} S & S^0 \\ \hline \mathbf{0} & 0 \end{array}\right]$, where $Q(x,y)$ is the transition rate of moving from state x to state y, multiplied by the transition's probability, $\mathbf{0}$ is a 1×3 vector of zeros, $\mathbf{1}$ is a 3×1 vector of ones, and $S^0 = -S\mathbf{1}$. Note that the last column and row of Q correspond to the absorbing state and each diagonal element is defined by the row's sum times -1. By Fig. 9, we have the following:

$$Q = \begin{bmatrix} -\lambda & \lambda & 0 & 0 \\ 0 & -\nu_1 & q\nu_1 & (1-q)\nu_1 \\ 0 & 0 & -\nu_2 & \nu_2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \text{ and } S = \begin{bmatrix} -\lambda & \lambda & 0 \\ 0 & -\nu_1 & q\nu_1 \\ 0 & 0 & -\nu_2 \end{bmatrix}.$$

The m-th moment of $\mathcal{T}_{t_n}$, which equals the one of $\tilde{\mathcal{T}}_{t_n}$, is $E[\mathcal{T}_{t_n}^m] = E[\tilde{\mathcal{T}}_{t_n}^m] = (-1)^m m! c S^{-m}\mathbf{1}$ [1].

We define the function $augmntThreadBlock(I(n), blocking(n))$, which estimates $\mathcal{T}_{t_n}$'s moments as we explained above. The function input includes the inter-arrival times $I(n)$ (which follows an $Exp(\Sigma_{i=1}^{J}\lambda_{i,n})$ distribution) and the function $blocking(n)$, which we approximate by $blocking(n) = A + \Sigma_{d=1}^{M} R(t_n, o_d)\cdot t_n.D[d]$, i.e., the average time it takes t_n to complete a job ($t_n.D[d]$ ends on job completion). Moreover, $augmntThreadBlock(I(n), blocking(n))$ outputs the estimation of $\mathcal{T}_{t_n}$ and checks if the OSE condition is violated (line 11 in Algorithm 1 of Section 4.3), i.e., the rate $\Sigma_{i=1}^{J}\lambda_{i,n}$ that defines $I(n)$ is greater or equal than $1/E[blocking(n)]$ for every $n \in [1, N]$. We detail the calculation of the pairwise delays, $s.D[d]$, in Section 9.

8.2. Object inter-demand period

The inter-demand period $\mathcal{T}_{o_d}$ of an object o_d is the time between two consecutive demand events that immediately follow the events of a thread gaining access to o_d (Section 5.6). As we showed in Section 5.6, $\mathcal{T}_{o_d}$ follows the same distribution

with the time between two consecutive release events of o_d, which defines the completion rate of jobs that include o_d (o_d's throughput). By its definition, $\mathcal{T}_{o_d}$ depends on the rate in which threads place acquisition requests in o_d's queue, as well as on the job completion period on o_d, $f_{\bullet,o_d}$ (Section 7). In the following, we detail an approximation of $\mathcal{T}_{o_d}$ (Approximation 5), which follows the approach of [23].

Consider an edge (s, o_d) of the acquisition graph $\mathscr{G}$. Threads that execute jobs that have an acquisition path equal to $(\bullet, s, o_d, \bullet)$ request access to o_d, immediately after the supply of item s. Therefore, Approximation 5 computes $\mathcal{T}_{o_d}$ by a weighted average of $\mathcal{T}_s$ (inter-demand period of item s) and f_{s,o_d} (job completion period), for every item s, such that (s, o_d) is an edge in $\mathscr{G}$. The weight $\omega(s, o_d)$ depends on the probability $R(s, o_d)$ as well as on the pairwise inter-demand period $s.T[d]$. That is, $\omega(s, o_d)$ depends on the probability of a thread to demand access to o_d, immediately after a supply event of item s, as well as on the time between such demands. We denote with $arrivals(o_d)$ the set of all items s, such that (s, o_d) is an edge in $\mathscr{G}$. For example, $arrivals(o_2) = thread \cup \{o_1\}$ and $thread = \{t_1, \ldots, t_N\}$.

Approximation 5 *(as in [23]). Let $\omega(s, o_d) = R(s, o_d) \cdot s.T[d]$ and $arrivals(o_d) = thread \cup \{o_j \mid j \in [1, d-1]\}$. The inter-demand period $\mathcal{T}_{o_d}$ of o_d is:*

$$\mathcal{T}_{o_d} = \frac{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d) \cdot (\mathcal{T}_s + f_{s,o_d})}{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d)}$$

Approximation 5 details $\mathcal{T}_{o_d}$'s backward dependency via its recursive equation that calculates $\mathcal{T}_{o_j}$, $j \in [1, d-1]$, before calculating $\mathcal{T}_{o_d}$. Moreover, the calculation of $\mathcal{T}_{o_d}$ depends on the inter-demand periods, which is $t_n.T[d]$, of (t_n, o_d) and respectively, $s.T[d]$, where $d \in [1, M]$ and $s \in thread \cup \{o_j \mid j \in [1, d-1]\}$. We use forward iterations (Section 4) for resolving such dependencies (Section 10). Furthermore, in Section 9 we calculate the pairwise inter-demand periods $s.T[d]$ for (s, o_d) in $\mathscr{G}$.

9. Resolving dependencies: pairwise delay and throughput computation

We showed how to estimate the blocking period of an object, $s.B[d]$, while depending on the delay for acquiring other objects, $o_d.D[d']$ (Section 7), as well as how to estimate the inter-demand periods $\mathcal{T}_{t_n}$ and $\mathcal{T}_{o_d}$, while depending on the $s.T[d]$ pairwise inter-demand periods, due to $(\bullet, s, o_d, \bullet)$ paths (Section 8). Recall that these variables are inter-dependent due to blocking. Theorem 6 demonstrates that we can resolve these interdependencies by representing the thread work cycles as a *contention subsystem*, in a way that is not subject to blocking and yet preserves these interdependencies.

To the end of proving Theorem 6, we first define contention subsystems in Section 9.1. Contention subsystems are closed systems that we construct basing on the items s and o_d, as well as on distinct copies (relay nodes) of every object o_j, such that $(\bullet, s, o_j, \bullet)$ is a job path, where $j \neq d$. These subsystems effectively represent the system parameters and thread blocking, because by their definition the time during which a thread is accessing an item in the contention subsystem includes the item's entire blocking period in the shared-object system.

Then, we present the Baynat–Dallery framework (Section 9.2), which is an adaptation of Baynat and Dallery's algorithm [4] to the setting of shared-object systems. The Baynat–Dallery framework's input is a contention subsystem for item s and o_d and its output is (the distributions of) $s.T[d]$ and $s.D[d]$. We provide the key steps of Theorem 6's proof by looking into the case of $M = 3$ in Section 9.3, before giving the complete proof in Section 9.4, using background knowledge [4,23] (which we refer to in Section 3).

Theorem 6. *Let $s \in S_{o_d}$, where $S_{o_d} \in \{thread\} \cup \{\{o_j\} \mid j \in [1, d)\}$. The Baynat–Dallery framework can approximate the pairwise delay $s.D[d]$ and pairwise inter-demand period $s.T[d]$ through the contention subsystem $\mathcal{CS}(S_{o_d}, o_d) = (\mathcal{H}(S_{o_d}, o_d), (\mathcal{R}_s)_{s \in S_{o_d}}, (\mathcal{B}_s)_{s \in S_{o_d}})$, where $S_{o_d} \in \{thread\} \cup s_{o_d}$ and $s_{o_d} = \{\{o_i\} \mid i \in [1, d-1]\}$. The running time of each framework iteration is $O(M \cdot N^4)$.*

9.1. Contention subsystems

Given any pair of system items, s and o_d, we calculate the (pairwise) state $c[s, o_d]$ for $\mathscr{G}$'s edge (s, o_d) using a construction that we name *contention subsystem* $\mathcal{CS}(S_{o_d}, o_d)$, where $d \in [1, M]$, $s \in S_{o_d}$ and S_{o_d} is either the set of all threads, $S_{o_d} = thread$, representing $(t_n, d, \bullet)$ job paths, or a set including a single object, $S_{o_d} = \{o_j\}$, $j \in [1, d-1]$, representing $(\bullet, o_j, o_d, \bullet)$ paths. For every item $s \in S_{o_d}$, we use the thread work cycle, $cycle(t_n, job_j)$ of t_n, $n \in [1, N]$, carrying out job_j, (Section 5), to show that the contention subsystem $\mathcal{CS}(S_{o_d}, o_d)$ represents the state of the shared-object system, with respect to the interdependencies among the delay $s.D[d]$ and the pairwise inter-demand period $s.T[d]$ when the blocking time and the item inter-demand periods are known.

Let $o_d \in \{o_1, \ldots, o_M\}$, $s_{o_d} = \{\{o_1\}, \ldots, \{o_{d-1}\}\}$ (i.e., $s_{o_d} = \emptyset$ when $d = 1$), and $S_{o_d} \in \{thread\} \cup s_{o_d}$, where $thread = \{t_1, \ldots, t_N\}$. Moreover, let $Rel(thread, o_d) = [1, M] \setminus \{d\}$ and $Rel(\{o_i\}, o_d) = [i+1, M] \setminus \{d\}$, where $\{o_i\} \in s_{o_d}$. We partition the $(\bullet, s, \bullet)$ paths to three sets, $\mathcal{P}(s, o_d) = \cup_{\ell \in [1,3]} \mathcal{P}_\ell$, where $\mathcal{P}_1 = \{path \mid path = (\bullet, s, o_d, \bullet)\}$, $\mathcal{P}_2 = \{path \mid path = (\bullet, s, o_i, \bullet, o_d, \bullet) \wedge i \in Rel(S_{o_d}, o_d) \setminus [d+1, M]\}$, $\mathcal{P}_3 = \{path \mid path = (\bullet, s, \bullet) \wedge o_d \notin path\}$.

A *contention subsystem*, is denoted by $\mathcal{CS}(S_{o_d}, o_d) = (\mathcal{H}(S_{o_d}, o_d), (\mathcal{R}_s)_{s \in S_{o_d}}, (\mathcal{B}_s)_{s \in S_{o_d}})$ and defined by (1)–(3).

(1) The *contention graph* $\mathcal{H}(S_{o_d}, o_d) = (\mathcal{V}, \mathcal{E})$ has the set of vertices $\mathcal{V} = \cup_{s \in S_{o_d}} \mathcal{V}_s$, and the set of edges $\mathcal{E} = \cup_{s \in S_{o_d}} \mathcal{E}_s$, such that for every $s \in S_{o_d}$, $\mathcal{V}_s = \{s\} \cup \{d\} \cup \{relay(s, o_j) \mid j \in Rel(S_{o_d}, o_d)\}$ and $\mathcal{E}_s = \mathcal{E}_s^1 \cup \mathcal{E}_s^2 \cup \mathcal{E}_s^3$, where $\mathcal{E}_s^1 = \{(s, o_d), (o_d, s)\}$, $\mathcal{E}_s^2 = \cup_{j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]} \{(s, relay(s, o_j)), (relay(s, o_j), o_d), (o_d, s)\}$, and $\mathcal{E}_s^3 = \cup_{j \in Rel(S_{o_d}, o_d)} \{(s, relay(s, o_j)), (relay(s, o_j), s)\}$, i.e., $\mathcal{E}_s^k$ corresponds to the partition $\mathcal{P}_k$, where $k \in [1, 3]$. Note that $\mathcal{H}(S_{o_d}, o_d)$ is a simple graph, i.e., there are no multiple edges between two vertices. Moreover, $relay(s, o_j)$ is a distinct copy of a relay o_j, $j \in Rel(S_{o_d}, l)$, for $s \in S_{o_d}$. A relay node $relay(s, o_j)$ will allow us to separately analyze the effect of jobs with paths $(\bullet, s, o_j, \bullet)$ on $s.T[d]$ and $s.D[d]$. For example, jobs with paths $(t_n, o_j, \bullet)$ and $(t_{n'}, o_j, \bullet)$ effect differently $s.T[d]$ and $s.D[d]$, for $s \in \{t_n, t_{n'}\}$. This decomposition technique follows the approach of [23].

(2) The *request probability matrices* $\mathcal{R}_s$ for $\mathcal{H}(S_{o_d}, o_d) = (\mathcal{V}, \mathcal{E})$ and $s \in S_{o_d}$. $\mathcal{R}_s(s, o_d)$ depicts the probability of a path $(\bullet, s, o_d, \bullet)$. Moreover, $\mathcal{R}_s(s, relay(s, o_j))$ depicts the probability of a path $r = (\bullet, s, o_j, \bullet)$, while $\mathcal{R}_s(relay(s, o_j), o_d)$ depicts the probability that $r = (\bullet, s, o_j, \bullet, o_d, \bullet)$. Furthermore, $\mathcal{R}_s(v, s)$, $v \in \mathcal{V}_s \setminus \{s\}$, depicts the probability of a thread becoming idle or starting a new job after the completion of a job, which is a certain event and therefore $\mathcal{R}_s(v, s) = 1$, $v \in \mathcal{V}_s \setminus \{s\}$.

(3) The *blocking periods*, $(\mathcal{B}_s)_{s \in S_{o_d}}$, where $\mathcal{B}_s$ is a function over the set of items in $\mathcal{V}_s$, and $s \in S_{o_d}$ refers to the thread blocking periods on each of $\mathcal{V}_s$'s items. Note that $\mathcal{E}_s^i$ forms a directed circle in $\mathcal{H}(S_{o_d}, o_d)$, where $s \in S_{o_d}$ and $i \in [1, 3]$. A demand request to o_d that follows the supply of $s \in S_{o_d}$ and possibly the supply of a relay object, o_j, $j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]$, is blocking o_d for $\mathcal{B}_s(o_d) = s.B[d]$ time. A demand request to node $relay(s, o_j)$ that follows the supply of $s \in S_{o_d}$, blocks that node for a period of $\mathcal{B}_s(relay(s, o_j)) = s.D[j]$, if $j \in [d+1, M]$ and $\mathcal{B}_s(relay(s, o_j)) = W(s, relay(s, o_j), o_d)$, if $j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]$, where $W(s, relay(s, o_j), o_d)$ equals the delay $s.D[j]$ minus the blocking period of a possibly subsequent demand event to o_d. Once a job is completed, another demand event follows the supply of $s \in \mathcal{V}_s$ after a period of $\mathcal{B}_s(s) = \mathcal{T}_s$ (by the definition of $\mathcal{T}_s$). Lemma 7 of Section 9.4 shows that a contention subsystem represents the dependencies among the threads in a shared-object system with respect to its state.

9.2. Baynat–Dallery framework

In this section, as a background knowledge, we discuss a variation on Baynat and Dallery's framework [4] that we adapt to the context of shared-object systems (Algorithm 2). The $BDF()$ function denotes our adapted version of Baynat and Dallery's framework. For every $s \in S_{o_d}$, this function takes a contention subsystem, $\mathcal{CS}(S_{o_d}, o_d)$, which is the tuple $(\mathcal{H}(S_{o_d}, o_d), (\mathcal{R}_s)_{s \in S_{o_d}}, (\mathcal{B}_s)_{s \in S_{o_d}})$, as an input and returns an estimation of the delay $s.D[d]$ and inter-demand period $s.T[d]$. Namely, $(s.T[d], s.D[d])_{s \in S_{o_d}} = BDF(\mathcal{CS}(S_{o_d}, o_d))$. The solution of Baynat and Dallery is based on iterative approximations of the demand arrival rates and request completion rate to o_d with the ones in the subgraph $\mathcal{H}_s(S_{o_d}, o_d) = (\mathcal{V}_s, \mathcal{E}_s)$, and vice versa, until, for every $s \in S_{o_d}$, their absolute difference is below a given threshold. The authors of [4] demonstrate the convergence of their iterative methods via numerical experiments. Baynat and Dallery [4] show that each iteration has polynomial running time, which is $O(M \cdot N^4)$ for the OSE case (Lemma 9).

We complete this section with a detailed explanation of Algorithm 2. The procedure starts by an initialization phase (lines 7–9), which is followed by a repeat-until loop (lines 10–24) and the output calculation (lines 25–26) before returning the output (line 27).

9.2.1. Variables

Let $s \in S_{o_d}$ denote a thread, if $S_{o_d} = thread$, or an object, if $S_{o_d} = \{o_j\}$, where $j \in [1, d-1]$. For item $v \in \mathcal{V}_s$, we define $\mathcal{I}_s(v)$, $\mathcal{B}_s(v)$ and $\mathcal{C}_s(v)$ to be item v's the inter-arrival time $\mathcal{I}_s(v)$, blocking period $\mathcal{B}_s(v)$, and respectively, inter-demand period $\mathcal{C}_s(v)$. With respect to the $BDF()$ function, $\mathcal{I}_s(v)$ is the time between two demand events to item v, $\mathcal{B}_s(v)$ is the blocking period of an arbitrary demand event to v, and respectively, $\mathcal{C}_s(v)$ is the time between two release events on v. Note that when s is a thread ($S_{o_d} = thread$), $\mathcal{I}_s(s)$ is the time between two object release events by s, $\mathcal{B}_s(s)$ is the time from an object release event until the next job completion by s, and respectively, $\mathcal{C}_s(s)$ is the time between two job completions by s. Baynat and Dallery approximate $\mathcal{I}_s(v)$, $\mathcal{B}_s(v)$ and $\mathcal{C}_s(v)$ using exponential distributions with parameters (rates) $\gamma_s(v)$, $\mu_s(v)$, and respectively, $\nu_s(v)$. Note that these random variables depend only on s and v, due to the definition of the contention subsystem (Section 9.1) and Lemma 7 of Section 9.4. The $BDF()$ function estimates $\gamma_s(v)$, $\mu_s(v)$ and $\nu_s(v)$ and uses them to compute $s.D[j]$ and $s.T[j]$ for every $s \in S_{o_d}$.

We define the subgraph $\mathcal{H}_s(S_{o_d}, o_d) = (\mathcal{V}_s, \mathcal{E}_s)$ of $\mathcal{H}(S_{o_d}, o_d)$ (Section 9.1). Note that when $S_{o_d} = thread$, thread $s = t_n$ requests to access only to items in $\mathcal{H}_s(S_{o_d}, o_d)$'s subgraph. Similarly, when $S_{o_d} = \{o_\ell\}$, a thread that has access to $s = o_\ell$ requests to access only to $\mathcal{H}_s(S_{o_d}, o_d)$'s items, where $\ell \in [1, d-1]$. Baynat and Dallery treat these subgraphs as Gordon–Newell networks [9,6]. The $BDF()$'s repeat-until loop (lines 10–24) alternates between computing $\gamma_s(v)$, $\mu_s(v)$ and $\nu_s(v)$ for the Gordon–Newell network defined by the subgraph $\mathcal{H}_s(S_{o_d}, o_d)$, for every $s \in S_{o_d}$ and the same values for every individual item $v \in \mathcal{V}$. The iterations stop when for two consecutive loops there is no $s \in S_{o_d}$ and $v \in \mathcal{V}_s$, such that the values of $\mu_s(v)$ differ more than a given ε (line 24).

9.2.2. Initialization phase

The $BDF()$ function initializes $\mu_s(v)$ with $1/E(\mathcal{B}_s(v))$ (lines 7–8) and computes the steady-state probabilities of a thread to demand or have access to item v, after the supply of item s. It does that through the function $stationary()$, which takes

Algorithm 2: The $BDF()$ function for estimating delay and pairwise inter-demand period through a contention subsystem.

1 **Input:** $\mathcal{CS}(S_{o_d}, o_d) = (\mathcal{H}(S_{o_d}, d), (\mathcal{R}_s)_{s \in S_{o_d}}, (\mathcal{B}_s)_{s \in S_{o_d}})$;
2 **Output:** $(s.T[d], s.D[d])_{s \in S_{o_d}}$;
3 **Macros:**
4 $converged(prev, curr) = (\nexists \mu_s(v) \in prev, \mu'_s(v) \in curr : |\mu_s(v) - \mu'_s(v)| \geq \varepsilon)$;
5 $Z(v) = \{s \in S_{o_d} | v \in \mathcal{V}_s\}$;
6 **begin**
7 **foreach** $s \in S_{o_d}$ **do**
8 **foreach** $v \in \mathcal{V}_s$ **do** $\mu_s(v) \leftarrow 1/E(\mathcal{B}_s(v))$
9 **foreach** $s \in S_{o_d}$ **do** $(steadyStateProbabilities_s(v))_{v \in \mathcal{V}_s} \leftarrow stationary(\mathcal{R}_s)$
10 **repeat**
11 **let** $oldValues = (\mu_s(v))_{s \in S_{o_d}, v \in V}$;
12 **foreach** $s \in S_{o_d}$ **do**
13 $GordonNewellConstant_s \leftarrow \sum_{v \in \Theta(s)} steadyStateProbabilities_s(v)/\mu_s(v)$;
14 **foreach** $v \in \mathcal{V}_s$ **do**
15 $subgraphMarginalProbs_{s,v}(1) \leftarrow \frac{steadyStateProbabilities_s(v)}{\mu_s(v) \cdot GordonNewellConstant_s}$;
16 $subgraphMarginalProbs_{s,v}(0) \leftarrow 1 - subgraphMarginalProbs_{s,v}(1)$;
17 $\gamma_s(v) \leftarrow \mu_s(v) \cdot \frac{subgraphMarginalProbs_{s,v}(1)}{subgraphMarginalProbs_{s,v}(0)}$;
18 **foreach** $v \in V$ **do**
19 **foreach** $s \in Z(v)$ **do**
20 $itemMarginalProbs_{s,v}(0) \leftarrow idleProb(v, s, \{\gamma_s(v)\}_{s \in S_{o_d}}, \{\mu_s(v)\}_{s \in S_{o_d}}, (\mathcal{R}_s)_{s \in S_{o_d}})$;
21 $itemMarginalProbs_{s,v}(1) \leftarrow 1 - itemMarginalProbs_{s,v}(0)$;
22 **foreach** $s \in Z(v)$ **do** $\nu_s(v) \leftarrow \gamma_s(v) \cdot \frac{itemMarginalProbs_{s,v}(0)}{itemMarginalProbs_{s,v}(1)}$
23 **foreach** $s \in S_{o_d}$ **do** (**foreach** $v \in \mathcal{V}_s$ **do** $\mu_s(v) \leftarrow \nu_s(v)$);
24 **until** $converged(oldValues, \{\mu_s(v)\}_{s \in S_{o_d}, v \in \mathcal{V}})$;
25 **foreach** $s \in S_{o_d}$ **do**
26 **let** $(s.D[d], s.T[d]) = (1/(\mu_s(o_d) \cdot subgraphMarginalProbs_{s,o_d}(0)), 1/\gamma_s(o_d))$;
27 **return** $(s.T[d], s.D[d])_{s \in S_{o_d}}$;

the stochastic matrix $\mathcal{R}_s$ as an input. The function $stationary()$ outputs the steady state vector, $steadyStateProbabilities_s$, which has the size of $|\mathcal{V}_s|$. Moreover, $steadyStateProbabilities_s$ satisfies the equations $\pi \cdot \mathcal{R}_s = \pi$ and $\Sigma_{i=1}^{|\mathcal{V}_s|} \pi_i = 1$ (line 9).

9.2.3. The repeat-until loop

The $BDF()$ function's repeat-until loop calculates $\gamma_s(v)$ (lines 12–17), $\nu_s(v)$ (lines 18–22) and $\mu_s(v)$ (line 23). We calculate the Gordon–Newell normalizing constant (line 13) and the marginal probabilities of Gordon–Newell (lines 14–16). Using these marginal probabilities, we calculate $\gamma_s(v)$ for every $s \in S_{o_d}$ and $v \in \mathcal{V}_s$ (line 17).

We find the marginal probability of an item to be idle through the $idleProb()$ function (lines 18–21) and then calculate $\nu_s(v)$ (line 22) for every $s \in Z(v)$, where $Z(v) = \{s \in S_{o_d} \mid v \in \mathcal{V}_s\}$ (line 5). The $idleProb()$ function calculates item v's marginal probability to be idle through the underlying Markov chain of a multi-class queue with exponential arrivals ($\gamma_s(v)$ for every $s \in Z(v)$). It also calculates the blocking periods ($\mu_s(v)$ for every $s \in Z(v)$). Note that the queue length is limited by the maximum number of pending demands (N when $S_{o_d} = thread$ and 1 when $S_{o_d} = \{o_j\}$, $j \in [1, d-1]$) [4,5].

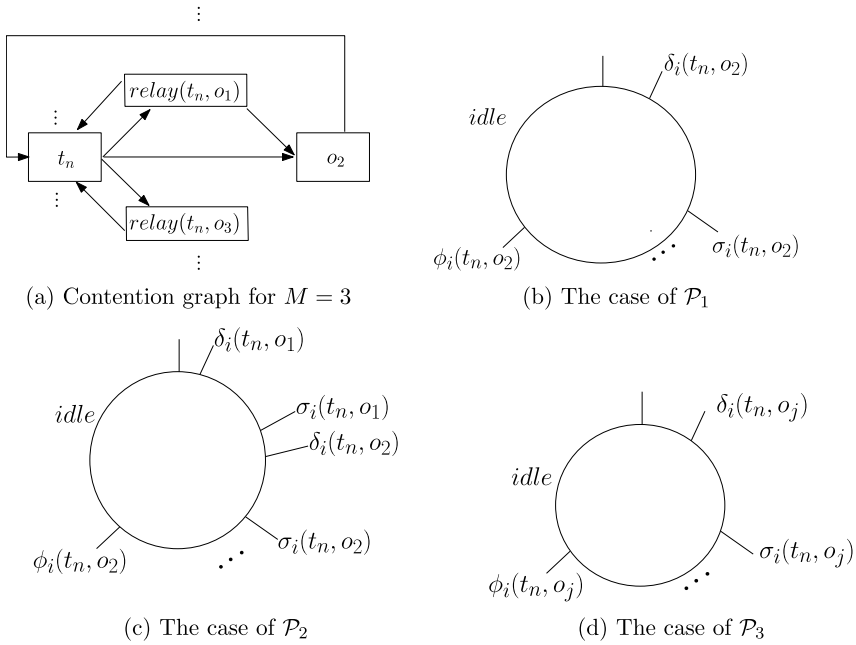
The calculation $\mu_s(v)$'s new estimates (line 23) happens before the next iteration. In order to check the convergence condition, each iteration begins with storing in the $oldValues$ variable the last estimations of $\mu_s(v)$ for every $s \in S_{o_d}$ and $v \in \mathcal{V}_s$ (line 24).

9.2.4. $BDF()$'s output

We estimate $s.D[d]$ through the delay in o_d's queue in the contention subsystem (line 26), i.e. by $Exp(\mu_s(o_d) \cdot subgraphMarginalProbs_{s,o_d}(0))$. Moreover, we obtain an estimation of the inter-demand period $s.T[d]$ through $Exp(\gamma_s(o_d))$ (line 26). The $BDF()$ function returns the output in line 27.

9.3. The case of systems with $M = 3$ objects

We use an illustrative example that shows how the contention subsystem of $S_{o_2} = thread$ and o_2 represents the dependencies among the threads of a system with $M = 3$ objects and N threads, with respect to the delays and pairwise inter-demand periods, when the blocking times and the item inter-demand periods are known. We construct the contention subsystem $\mathcal{CS}(thread, o_2) = (\mathcal{H}(thread, o_2), (\mathcal{R}_s)_{s \in thread}, (\mathcal{B}_s)_{s \in thread})$ based on the work cycles related to the delay $t_n.D[2]$ and inter-demand period $t_n.T[2]$, for every $s = t_n \in thread$. We explain the representation of work cycles by a contention graph, which is illustrated in Fig. 10a, and the adaptation of the request probabilities and blocking times to the ones of the



(a) Contention graph for $M = 3$ (b) The case of $\mathcal{P}_1$

(c) The case of $\mathcal{P}_2$ (d) The case of $\mathcal{P}_3$

Fig. 10. The contention graph for $\mathcal{CS}(thread, 2)$ and the work cycles partitions, $\mathcal{P}(s, o_2) = \cup_{\ell \in [1,3]} \mathcal{P}_\ell$, of $(s, \bullet)$ paths, where $\mathcal{P}_1 = \{\chi | \chi = (s, o_2, \bullet)\}$, $\mathcal{P}_2 = \{\chi | \chi = (s, o_1, o_2, \bullet)\}$, $\mathcal{P}_3 = \{\chi | \chi = (s, \bullet) \wedge o_2 \notin \chi\}$, $s = t_n$.

contention subsystem. The challenge here is to demonstrate that a dynamic system that is based on correlated events with dependencies that are due to blocking and follow non-deterministic schedules can be represented by these subsystems. After demonstrating this part of the proof, the rest of the proof follows by matching between the subsystems presented here to the one by Ramesh–Perros [23], which use a framework proposed by Baynat and Dallery [4] for estimating our system's state.

9.3.1. Contention graph of $\mathcal{CS}(thread, o_2)$

Let $\mathcal{H}(thread, o_2) = (\mathcal{V}, \mathcal{E})$ for $S_{o_d} = thread$ (Fig. 10a). Given an arbitrary thread, $s = t_n$, $n \in [1, N]$, let $\mathcal{P}(s, o_2) = \cup_{\ell \in [1,3]} \mathcal{P}_\ell$ be a partition of $(s, \bullet)$ paths, where $\mathcal{P}_1 = \{path \mid path = (s, o_2, \bullet)\}$, $\mathcal{P}_2 = \{path \mid path = (s, o_1, o_2, \bullet)\}$ and $\mathcal{P}_3 = \{path \mid path = (s, \bullet) \wedge o_2 \notin path\}$. Moreover, let $\mathcal{V} = \cup_{n \in [1,N]} \mathcal{V}_s$ be the union of $\mathcal{V}_s = \{s, relay(s, o_1), o_d, relay(s, o_3)\}$, where $relay(s, o_j)$, $j \in \{1, 3\}$, are s's distinct copies of a relay object, o_j, which allow us to distinguish paths with respect to threads. The contention graph's nodes $s, relay(s, o_j)$ and o_d represent s, o_j and, respectively o_d, in the shared-object system, where $j \in \{1, 3\}$.

The edges $\mathcal{E} = \cup_{n \in [1,N]} \mathcal{E}_s$ follow the path partition cases, $\{\mathcal{E}_\ell\}_{\ell \in [1,3]}$. Let job_i be a job that $s = t_n$ carries out. The edge sets $\mathcal{E}_\ell$, where $\ell \in [1, 3]$, are defined as follows:

- $\mathcal{P}_1$'s case refers to work cycles, for which s demands access to o_2, once it is assigned with job_i. When s gains access to o_d, job_i might require s to demand access to o_3. Upon job_i's completion s releases any acquired object. Thus, the edges in $\mathcal{E}_1 = \{(s, o_2), (o_2, s)\}$ (Fig. 10a) represent the work cycle subvectors $(\delta_i(s, o_2))$, and respectively, $(\sigma_i(s, o_2), \ldots, \phi_i(s, o_2), \ldots)$ (Fig. 10b).
- $\mathcal{P}_2$'s case refers to work cycles, for which s's demand for access to o_1 is followed by s's demand for access to o_2 after o_1's supply, which is then followed by job_i's completion. Thus, the edges in the set $\mathcal{E}_2 = \{(s, relay(s, o_1)), (relay(s, o_1), o_2), (o_2, s)\}$ (Fig. 10a) represent the work cycle subvectors $(\delta_i(s, o_1))$, $(\sigma_i(s, o_1), \delta_i(s, o_2))$, and respectively, $(\sigma_i(s, o_2), \ldots, \phi_i(s, o_2), \ldots)$ (Fig. 10c).
- $\mathcal{P}_3$'s case refers to work cycles, for which s demands access to o_j and then completes job_i, where $j \in \{1, 3\}$. Therefore, the edges in $\mathcal{E}_3 = \{(s, relay(s, o_j)), (relay(s, o_j), s)\}$ (Fig. 10a), represent the subvector $(\delta_i(s, o_j))$, and respectively, $(\sigma_i(s, o_j), \ldots, \phi_i(s, o_j), \ldots)$ of the work cycle (Fig. 10d).

9.3.2. $\mathcal{CS}(thread, o_2)$'s blocking times and request probabilities

We complete the example in which we show how the contention subsystem represents the dependencies among the threads in the shared-object system. We refer to an arbitrary job, say job_i, that $s = t_n$ carries out and explain how the contention subsystem's request probabilities $(\mathcal{R}_s)_{n \in [1,N]}$ and blocking periods $(\mathcal{B}_s)_{n \in [1,N]}$ represent the request probabilities, and respectively, the blocking periods in the shared-object system. We justify this representation using the work cycle of job_i, when its path is in the path partition $\mathcal{P}_j$, for every $j \in [1, 3]$. Note that for each such path partition, the period

between two consecutive work cycles completed by s is represented by $\mathcal{B}_s(s) = \mathcal{T}_s$ (Fig. 10). Namely, if s carries out job_i and consecutively $job_{i'}$, and their first demand requests are $\delta_i(s, o_2)$, and respectively, $\delta_{i'}(s, d')$, the blocking period $\mathcal{B}_s(s)$ represents the period between these two events (in the contention subsystem).

The case of $(s, o_1, \bullet)$ *paths* Consider the case where s carries out job_i with path $r = (s, o_1, \bullet)$, where $r \in \mathcal{P}_2$, if o_2 is included in r and $r \in \mathcal{P}_3$, if o_2 is not included in r. We present the request probabilities among $s, relay(s, o_1)$ and o_2, as well as the blocking times on each of these items in the contention subsystem.

The probability $\mathcal{R}_s(s, relay(s, o_1))$ The probability $\mathcal{R}_s(s, relay(s, o_1)) = R(s, o_1)$ (by the definition of R) denotes the contention subsystem event of s demanding access to $relay(s, o_1)$, which represents s demanding access to o_1 immediately after job_i's assignment in the shared-object system (Figs. 10c and 10d, when $j = 1$).

The blocking period $\mathcal{B}_s(relay(s, o_1))$ Let $W(s, o_1, o_2)$ denote the period during which s blocks o_1, minus the possible blocking period on o_2 (after the supply of access to o_1) in the shared-object system. Moreover, let $X(s, o_1, o_2) = \Pr[(s, o_1, o_2)] \cdot o_1.B[2]$ denote the (possible) blocking period of $s = t_n$ to o_2, after gaining access to o_1 in the shared-object system, where the probability $\Pr[(s, o_1, o_2)] = R(s, o_1) \cdot R(o_1, o_2)$ denotes the event of s demanding access to o_1 and successively to o_2. Namely, $X(s, o_1, o_2)$ is the time between the work cycle events $\delta_i(s, o_2)$ and $\phi_i(o_2, s)$ times the probability of s to demand access to o_2 after gaining access to o_1 (Fig. 10c). Thus, in the shared-object system $W(s, o_1, o_2) = s.D[1] - X(s, o_1, o_2)$. Therefore, $\mathcal{B}_s(relay(s, o_1)) = W(s, o_1, o_2)$ represents the period during which s blocks $relay(s, 1)$ before possibly demanding access to o_2 in the contention subsystem.

The probabilities $\mathcal{R}_s(relay(s, o_1), o_2)$ *and* $\mathcal{R}_s(relay(s, o_1), s)$ Let $\mathcal{F}_{(s,o_1,o_2)}$ and $\mathcal{F}'_{(s,o_1,o_2)}$ denote the events in which s demands access to o_2 after gaining access to o_1 in the shared-object system, and respectively, to $relay(s, o_1)$ in the contention subsystem. The event $\mathcal{F}'_{(s,o_1,o_2)}$ in the contention subsystem represents the event $\mathcal{F}'_{(s,o_1,o_2)}$ in the shared-object system, and therefore, the probability of $\mathcal{F}'_{(s,o_1,o_2)}$ is given by $\mathcal{R}_s(relay(s, o_1), o_2) = \Pr[(s, o_1, o_2)]$ (Fig. 10c). Moreover, when the event $\mathcal{F}'_{(s,o_1,o_2)}$ (and thus the event $\mathcal{F}_{(s,o_1,o_2)}$) does not occur (Fig. 10d), the respective job is completed and $s = t_n$ becomes idle or starts a new job with probability $\mathcal{R}_s(relay(s, o_1), s) = 1 - \mathcal{R}_s(relay(s, o_1), o_2)$.

The case of $(s, o_2, \bullet)$ *paths* Consider the case where job_i's path is $(s, o_2, \bullet) \in \mathcal{P}_1$. In Fig. 10b, s demands access to o_2 immediately after it is assigned with job_i. This is represented in $\mathcal{CS}(thread, o_2)$ by s demanding access to o_2, with probability $\mathcal{R}_s(s, o_2) = R(s, o_2)$. Moreover, s blocks o_2 for a period of $\mathcal{B}_s(o_2) = s.B[2]$ in $\mathcal{CS}(thread, o_2)$, which represents the period between the events $\sigma_i(s, o_2)$ and $\phi_i(s, o_2)$ in Figs. 10b and 10c. After the job completion and the release event of o_2 in $\mathcal{CS}(thread, o_2)$, s enters, with probability $\mathcal{R}_s(o_2, s) = 1$, an idle period (of possibly zero length) until it starts carrying out a new job.

The case of (s, o_3) *paths* Consider the case where s carries out job_i with path $r = (s, o_3) \in \mathcal{P}_3$. In $\mathcal{CS}(thread, o_2)$, s demands access to $relay(s, o_3)$, with probability $\mathcal{R}_s(s, relay(s, o_3)) = R(s, o_3)$, which represents s demanding access to o_3 immediately after job_i's assignment in the shared-object system (Fig. 10d). The blocking period of s on $relay(s, o_3)$ is $\mathcal{B}_s(relay(s, o_3)) = s.D[3]$, which in the shared-object system represents the time that s is waiting to gain access to o_3 and then blocking it, i.e., the period between the work cycle events $\delta_i(s, o_3)$ and $\phi_i(s, o_3)$ (Fig. 10d). After the job completion and the release event of $relay(s, o_3)$ in $\mathcal{CS}(thread, o_2)$, s enters, with probability $\mathcal{R}_s(relay(s, o_3), s) = 1$, an idle period (of possibly zero length) until it starts carrying out a new job.

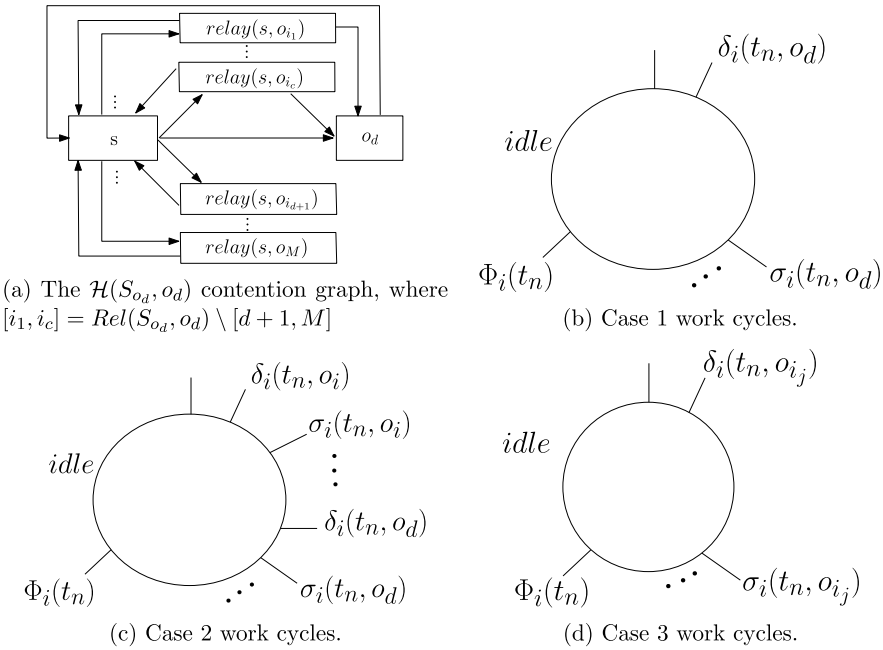
The subsystem $\mathcal{CS}(thread, o_2) = (\mathcal{H}(thread, o_2), (\mathcal{R}_s)_{s \in thread}, (\mathcal{B}_s)_{s \in thread})$, which we described above, is a contention subsystem that represents the dependencies among the thread set, *thread*, and o_2 in the shared-object system.

9.4. The case of systems with M objects

In this section we prove that Theorem 6 follows from Lemmas 7, 8 and 9 (Corollary 10).

Lemma 7. *Consider a contention subsystem* $\mathcal{CS}(S_{o_d}, o_d) = (\mathcal{H}(S_{o_d}, o_d), (\mathcal{R}_s)_{s \in S_{o_d}}, (\mathcal{B}_s)_{s \in S_{o_d}})$, *where* $S_{o_d} \in \{thread\} \cup s_{o_d}$ *and* $s_{o_d} = \{\{o_i\} \mid i \in [1, d-1]\}$. *Suppose that we are given the shared-object system's blocking and item inter-demand periods, as well as the request probabilities* R. *It holds that* $\mathcal{CS}(S_{o_d}, o_d)$ *represents the dependencies among the threads in the shared-object system and the system's state.*

Proof. We show a mapping of the shared-object system's state to the contention subsystem $\mathcal{CS}(S_{o_d}, o_d)$. Given the shared-object system's blocking and item inter-demand periods, as well as the request probabilities, we construct the contention subsystem $\mathcal{CS}(S_{o_d}, o_d) = (\mathcal{H}(S_{o_d}, o_d), (\mathcal{R}_s)_{s \in S_{o_d}}, (\mathcal{B}_s)_{s \in S_{o_d}})$ based on the work cycles related to jobs with $(\bullet, s, \bullet)$ paths, where $s \in S_{o_d}$ and $s_{o_d} = \{\{o_i\} \mid i \in [1, d-1]\}$. We explain the representation of work cycles by the contention graph $\mathcal{H}(S_{o_d}, o_d)$, as well as the representation of the shared-object system's request probabilities and state, i.e., blocking, pairwise



(a) The $\mathcal{H}(S_{o_d}, o_d)$ contention graph, where $[i_1, i_c] = Rel(S_{o_d}, o_d) \setminus [d+1, M]$

(b) Case 1 work cycles.

(c) Case 2 work cycles.

(d) Case 3 work cycles.

Fig. 11. The contention graph for $\mathcal{CS}(S_{o_d}, o_d)$ and the work cycles partitions, $\mathcal{P}(s, o_d) = \cup_{\ell \in [1,3]} \mathcal{P}_\ell$, of $(\bullet, s, \bullet)$ paths, where $\mathcal{P}_1 = \{path \mid path = (\bullet, s, o_d, \bullet)\}$, $\mathcal{P}_2 = \{path \mid path = (\bullet, s, o_i, \bullet, o_d, \bullet) \wedge i \in Rel(S_{o_d}, o_d) \setminus [d+1, M]\}$, $\mathcal{P}_3 = \{path \mid path = (\bullet, s, \bullet) \wedge o_d \notin path\}$.

inter-demand period and delay, by $(\mathcal{R}_s)_{s \in S_{o_d}}$, and respectively, $(\mathcal{B}_s)_{s \in S_{o_d}}$ in the contention subsystem. This construction is the mapping that proves the lemma's statement.

The proof is organized as follows. In the first part, we construct the contention graph $\mathcal{H}(S_{o_d}, o_d) = (\mathcal{V}, \mathcal{E})$ using the work cycles related to $(\bullet, s, \bullet)$ paths. Moreover, in the second part we show that $(\mathcal{R}_s)_{s \in S_{o_d}}$ and $(\mathcal{B}_s)_{s \in S_{o_d}}$ represent the dependencies and the shared-object system's state with respect to $(\bullet, s, \bullet)$ paths. In our construction, we assume the knowledge of the item inter-demand periods $\mathcal{T}_v$, the system's state delay and blocking periods, as well as the request probabilities R.

Part I: The graph $\mathcal{H}(S_{o_d}, o_d)$ represents the work cycles related to the $(\bullet, s, \bullet)$ paths. Let $\mathcal{H}(S_{o_d}, o_d) = (\mathcal{V}, \mathcal{E})$ be the contention graph of $\mathcal{CS}(S_{o_d}, o_d)$ (Fig. 11a) and consider s to be an arbitrary element of S_{o_d}. Recall that $Rel(thread, o_d) = [1, M] \setminus \{d\}$ and $Rel(\{o_\ell\}, o_d) = [\ell + 1, M] \setminus \{d\}$, where $\ell \in [1, d)$. Moreover, let $\mathcal{P}(s, o_d) = \cup_{\ell \in [1,3]} \mathcal{P}_\ell$ be a path partition of $(\bullet, s, \bullet)$ paths, where $\mathcal{P}_1 = \{path \mid path = (\bullet, s, o_d, \bullet)\}$, $\mathcal{P}_2 = \{path \mid path = (\bullet, s, o_\ell, \bullet, o_d, \bullet) \wedge \ell \in Rel(S_{o_d}, o_d) \setminus [d+1, M]\}$ and $\mathcal{P}_3 = \{path \mid path = (\bullet, s, \bullet) \wedge o_d \notin path\}$. We explain the representation of the thread work cycles for jobs with $(\bullet, s, \bullet)$ paths by the graph $\mathcal{H}(S_{o_d}, o_d)$.

We define the elements of $\mathcal{V}$ and $\mathcal{E}$. Let $\mathcal{V} = \cup_{s \in S_{o_d}} \mathcal{V}_s$, where $\mathcal{V}_s = \{s, o_d\} \cup \{relay(s, o_j) \mid j \in Rel(S_{o_d}, o_d)\}$. The nodes $relay(s, o_j)$, $j \in Rel(S_{o_d}, o_d)$, are s's distinct copies of a relay object, o_j, which allow us to distinguish paths with respect to threads. The edges $\mathcal{E} = \cup_{s \in S_{o_d}} \mathcal{E}_s$ follow the three path partition cases, i.e., $\mathcal{E}_s = \cup_{\ell \in [1,3]} \mathcal{E}_\ell$, where $\mathcal{E}_1 = \{(s, o_d), (o_d, s)\}$, $\mathcal{E}_2 = \{(s, relay(s, o_j)), (relay(s, o_j), o_d), (o_d, s)\}$ and $\mathcal{E}_3 = \{(s, relay(s, o_j)), (relay(s, o_j), s)\}$. Let job_i be a job that t_n carries out, such that item s is either included in job_i's object vector or $s = t_n$, and let $cycle(t_n, job_i)$ be the respective work cycle. The edges in the sets $\mathcal{E}_\ell$, where $\ell \in [1, 3]$, represent the events in every possible $cycle(t_n, job_i)$ after the supply of item s, if s is an object, or after the assignment of job_i to s, if $s = t_n$. For brevity, we refer to both events as the supply of item s. The edge sets are defined according to the three sets of the path partition $\mathcal{P}(s, o_d) = \cup_{\ell \in [1,3]} \mathcal{P}_\ell$ as follows.

- $\mathcal{P}_1$'s case refers to work cycles, for which t_n demands access to o_d, immediately after the supply of item s. When t_n gains access to o_d, job_i might require t_n to demand access to another object, $o_{d'}$, where $d' \in [d+1, M]$. Upon job_i's completion t_n releases any acquired object (event $\Phi_i(t_n)$ of the work cycle). Thus, the edges (s, o_d) and (o_d, s) of $\mathcal{E}_1$ (Fig. 11a) represent the subvectors $(\delta_i(t_n, o_d))$, and respectively, $(\sigma_i(t_n, o_d), \ldots, \Phi_i(t_n))$ of the work cycle (Fig. 11b).
- $\mathcal{P}_2$'s case refers to work cycles for which t_n, after the supply of item s, demands access to o_j, where $j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]$, and subsequently to o_d. Note that, by the definition of $\mathcal{P}_2$, t_n might also demand access to other objects in $\{o_{j+1}, \ldots, o_M\} \setminus \{o_d\}$. Thus, the edges $(s, relay(s, o_j)), (relay(s, o_j), o_d), (o_d, s)$ of $\mathcal{E}_2$ (Fig. 11a) represent the subvectors $(\delta_i(t_n, o_j)), (\sigma_i(t_n, o_j), \ldots \delta_i(t_n, o_d))$, and respectively, $(\sigma_i(t_n, o_d), \ldots, \Phi_i(t_n))$ of the work cycle (Fig. 11c).
- $\mathcal{P}_3$'s case refers to work cycles, for which t_n, after the supply of item s, demands access to o_j, where $j \in Rel(S_{o_d}, o_d)$. Note that, by the definition of $\mathcal{P}_3$, t_n might also demand access to other objects in $\{o_{j+1}, \ldots, o_M\}$, but not to o_d. Thus, the edges $(s, relay(s, o_j))$ and $(relay(s, o_j), s)$ of $\mathcal{E}_3$ (Fig. 11a), represent the subvectors $(\delta_i(t_n, o_j))$, and respectively, $(\sigma_i(t_n, o_j), \ldots, \Phi_i(t_n))$, where $j \in Rel(S_{o_d}, o_d)$ of the work cycle (Fig. 11d).

Part II: $(\mathcal{R}_s)_{s\in S_{o_d}}$ and $(\mathcal{B}_s)_{s\in S_{o_d}}$ represent the dependencies and the shared-object system's state with respect to $(\bullet, s, \bullet)$ paths. We refer to an arbitrary job, say job_i, that t_n carries out and explain how the contention subsystem's request probabilities $(\mathcal{R}_s)_{s\in S_{o_d}}$ and blocking periods $(\mathcal{B}_s)_{s\in S_{o_d}}$ represent the request probabilities, and respectively, the blocking periods in the shared-object system. We verify this representation using the work cycle of job_i, when its path is in the path partition $\mathcal{P}_j$, for every $j \in [1, 3]$. We remind that throughout this proof we refer to the supply of item s as the supply of access to o_ℓ for t_n, if $s = o_\ell$, $\ell \in [1, d-1]$, or the assignment of job_i to t_n, if $s = t_n$.

Note that for each such path partition, the period between two consecutive work cycles for jobs that include item s is represented by $\mathcal{B}_s(s) = \mathcal{T}_s$ (Fig. 10). Namely, the blocking period $\mathcal{B}_s(s)$ in the contention subsystem represents the period between the event of t_n releasing item s due to job_i, and consecutively, another thread, say $t_{n'}$, releasing item s due to a job, say $job_{i'}$, in the shared-object system. We show this representation by looking into three cases of path partitions, i.e., (1) paths $(\bullet, s, o_d, \bullet)$ in $\mathcal{P}_1$, (2) paths $r = (\bullet, s, o_j, \bullet)$, where $j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]$ ($r \in \mathcal{P}_2$, if o_d is included in r and $r \in \mathcal{P}_3$, if o_d is not included in r) and (3) paths $r = (t_n, \bullet, s, o_j, \bullet)$, where $j \in Rel(S_{o_d}, o_d)$, and $o_d \notin r$, i.e., $r \in \mathcal{P}_3$.

(1) Consider the case where job_i's path is in partition $\mathcal{P}_1$, i.e., t_n carries out job_i with path $(\bullet, s, o_d, \bullet)$. In Fig. 11b, t_n demands access to o_d immediately after the supply of item s. This event is represented in the contention subsystem by t_n demanding access to o_d, with probability $\mathcal{R}_s(s, o_d) = R(s, o_d)/K(S_{o_d}, o_d)$ (by the definition of R), where $K(S_{o_d}, o_d) = \Sigma_{v\in Rel(S_{o_d},o_d)\cup\{d\}} R(s, o_v)$ is a normalizing constant. Furthermore, t_n blocks o_d for a period of $\mathcal{B}_s(o_d) = s.B[d]$ in the contention subsystem, which represents the period between the events $\sigma_i(t_n, o_d)$ and $\Phi_i(t_n)$ in the work cycles presented in Figs. 11b and 11c. This is due to the fact that $s.B[d]$ also considers requests for accessing o_d while following either job path $(t_n, \bullet, s, o_d, \bullet)$ or $(t_n, s, \bullet, o_d, \bullet)$ (Approximation 4). After the job completion and the release event of o_d, another supply event of item s occurs or it becomes idle. Namely, if $s = t_n$, s will either start carrying out the next pending job or it will become idle, and if $s = o_\ell$, the thread on s's queue top will gain access to s in case the queue is not empty, otherwise s will become idle. These are certain events in the shared-object system and therefore occur with probability $\mathcal{R}_s(o_d, s) = 1$ in the contention subsystem.

(2) Consider the case where t_n carries out job_i with path $r = (\bullet, s, o_j, \bullet)$, where $j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]$. Note that $r \in \mathcal{P}_2$, if o_d is included in r and $r \in \mathcal{P}_3$, if o_d is not included in r. We present the request probabilities among $s, relay(s, o_j)$ and o_d, $j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]$, as well as the blocking times on each of these items in the contention subsystem. We present (i) the probability $\mathcal{R}_s(s, relay(s, o_j))$, (ii) the blocking period $\mathcal{B}_s(relay(s, o_j))$ and (iii) the probabilities $\mathcal{R}_s(relay(s, o_j), o_d)$ and $\mathcal{R}_s(relay(s, o_j), s)$.

(i) The probability $\mathcal{R}_s(s, relay(s, o_j)) = R(s, o_j)/K(S_{o_d}, o_d)$ (by the definition of R) denotes the contention subsystem event of t_n demanding access to $relay(s, o_j)$ after the supply of item s, which represents t_n demanding access to o_j immediately after the supply of item s in the shared-object system (Figs. 11c and 11d, when $j \in Rel(S_{o_d}, o_d) \setminus [d+1, M]$).

(ii) Consider the case where t_n might demand access to objects in $\{o_{j+1}, \ldots, o_M\}$, after the supply of o_j in the shared-object system. Let $W(s, o_j, o_d)$ as the period during which t_n blocks o_j minus t_n's possible blocking period on o_d in the shared-object system, due to a path in $\mathcal{P}_2$. We refer to $X(s, o_j, o_d) = \Sigma_{\ell=j}^{d-1} \Pr[(s, o_j, \bullet, o_\ell, o_d)] \cdot o_\ell.B[d]$ as the possible blocking period of t_n to o_d, after gaining access to o_j in the shared-object system. Note that the probability $\Pr[(s, o_j, \bullet, o_\ell, o_d)] = R(s, o_j) \cdot [\Sigma_{k=1}^{j-\ell} R^k(o_\ell, o_j)] \cdot R(o_\ell, o_d)$ denotes the event in which t_n, after the supply of item s, demands access to o_j, possibly to other objects in $\{o_{j+1}, \ldots, o_{\ell-1}\}$, to o_ℓ and successively to o_d (since R is a stochastic matrix). Namely, let $X(s, o_j, o_d)$ denote the time between the work cycle events $\delta_i(t_n, o_d)$ and $\Phi_i(t_n)$ times the probability of t_n to demand access to o_j, after gaining access to s, and subsequently to o_d (Fig. 11c). Thus, in the shared-object system $W(s, o_j, o_d) = s.D[j] - X(s, o_j, o_d)$. Therefore, the period during which t_n blocks $relay(s, o_j)$, after the supply of item s, and before possibly demanding access to o_d in the contention subsystem is represented by $\mathcal{B}_s(relay(s, o_j)) = W(s, o_j, o_d)$.

(iii) Let $\mathcal{F}_{(s,o_j,o_d)}$ and $\mathcal{F}'_{(s,o_j,o_d)}$ denote the event in which t_n, demands access to o_j, after the supply of item s, and subsequently to o_d in the shared-object system, and respectively, the event in which t_n, after the supply of item s, demands access to $relay(s, o_j)$ and consecutively to o_d in the contention subsystem. Note that t_n might demand access to other objects in $\{o_{j+1}, \ldots, o_{d-1}\}$ in the shared-object system. The event contention subsystem $\mathcal{F}'_{(s,o_j,o_d)}$ represents the event $\mathcal{F}'_{(s,o_j,o_d)}$ in the shared-object system, and therefore, the probability of $\mathcal{F}'_{(s,o_j,o_d)}$ is given by $\mathcal{R}_s(relay(s, o_j), o_d)$ which equals to $\Pr[(s, o_j, o_d)]$ (Fig. 11c). Moreover, when the event $\mathcal{F}'_{(s,o_j,o_d)}$ (and thus the event $\mathcal{F}_{(s,o_j,o_d)}$) does not occur (Fig. 11d), the respective job is completed and t_n becomes idle or starts a new job with probability $\mathcal{R}_s(relay(s, o_j), s) = 1 - \mathcal{R}_s(relay(s, o_j), o_d)$.

(3) Consider the case where t_n carries out job_i with path $r = (t_n, \bullet, s, o_j, \bullet)$, where $j \in Rel(S_{o_d}, o_d)$, and $o_d \notin r$, i.e., $r \in \mathcal{P}_3$. In the contention subsystem, t_n, after the supply of item s, demands access to $relay(s, o_j)$, with probability $\mathcal{R}_s(s, relay(s, o_j)) = R(s, o_j)$ (by the definition of R), which represents t_n demanding access to o_j immediately after the supply of item s in the shared-object system (Fig. 11d). The blocking period of t_n on $relay(s, o_j)$ is $\mathcal{B}_s(relay(s, o_j)) = s.D[j]$, which in the shared-object system represents the time that t_n is waiting to gain access to o_j, blocks o_j, possibly demands access to other objects (except for o_d), and then releases all of its acquired objects. Namely, $\mathcal{B}_s(relay(s, o_j))$ represents the period between the work cycle events $\delta_i(t_n, o_j)$ and $\Phi_i(t_n)$, after the supply of item s

(Fig. 11d). After the job completion and the release event of $relay(s, o_j)$ in the contention subsystem, either another thread gains access to item s or item s becomes idle (no thread is accessing it), i.e., this event is certain to happen in the shared-object system.

The $\mathcal{CS}(S_{o_d}, o_d) = (\mathcal{H}(S_{o_d}, o_d), (\mathcal{R}_s)_{s \in S_{o_d}}, (\mathcal{B}_s)_{s \in S_{o_d}})$ contention subsystem, which we described above, represents the dependencies among the item set, S_{o_d}, and o_d in the shared-object system. Therefore, the proof is complete. □

Lemma 8. *The framework of Baynat and Dallery can approximate the delay $s.D[d]$ and inter-demand period $s.T[d]$ through the contention subsystem $\mathcal{CS}(S_{o_d}, o_d)$.*

Proof. We first give the definition of a *Ramesh–Perros subsystem* (RPS) that is introduced in [23] and show that a contention subsystem can be directly mapped to an RPS. Ramesh and Perros show that we can find the pairwise inter-demand period, and delay of (blocking) communications in an RPS using a framework proposed by Baynat and Dallery in [4]. Thus, we can use the Baynat–Dallery framework to estimate the values of $s.T[d]$ and $s.D[d]$, for every $s \in S_{o_d}$, given a contention subsystem $\mathcal{CS}(S_{o_d}, o_d)$ as an input. In Section 9.2, we presented an adapted version of the Baynat–Dallery framework (Algorithm 2) and explained the calculation of the delay $s.D[d]$ and inter-demand period $s.T[d]$, given the contention subsystem $\mathcal{CS}(S_{o_d}, o_d)$.

A Ramesh–Perros subsystem is defined as follows. Let $\{o[1], \ldots, o[M]\}$ be a set of servers, such that tier-d includes only server $o[d]$, where $d \in [1, M]$, and $\{t[1], \ldots, t[N]\}$ be a set of clients. Moreover, let $s_{o_d} = \{\{o[1]\}, \ldots, \{o[d-1]\}\}$ (i.e., $s_{o_d} = \emptyset$ when $d = 1$) and $S_{o_d} \in \{\{t[1], \ldots, t[N]\}\} \cup s_{o_d}$. A Ramesh–Perros subsystem, is denoted by $\mathcal{RP}(\mathtt{S}_{o_d}, o_d) = (\mathtt{H}(S_{o_d}, o_d), (\mathtt{R}_x)_{x \in S_{o_d}}, (\mathtt{B}_x)_{x \in S_{o_d}})$ and defined as follows:

(1) The *RPS graph* $\mathtt{H}(S_{o_d}, o_d) = (\mathtt{V}, \mathtt{E})$ has the set of vertices $\mathtt{V} = \cup_{x \in S_{o_d}} \mathtt{V}_x$, and the set of edges $\mathtt{E} = \cup_{x \in S_{o_d}} \mathtt{E}_x$, such that $\mathtt{V}_x = \{x\} \cup \{o[d]\} \cup \{relay(x, o_j) \mid j \in \mathtt{Rel}(S_{o_d}, o_d)\}$ and $\mathtt{E}_x = \mathtt{E}_x^1 \cup \mathtt{E}_x^2 \cup \mathtt{E}_x^3$, where $\mathtt{Rel}(\{t[1], \ldots, t[N]\}, o_d) = [1, M] \setminus \{d\}$, $\mathtt{Rel}(\{o[i]\}, o_d) = [i+1, M] \setminus \{d\}$, where $o[i] \in s_{o_d}$, $\mathtt{E}_x^1 = \{(x, o[d]), (o[d], x)\}$, $\mathtt{E}_x^2 = \cup_{j \in \mathtt{Rel}(S_{o_d}, o_d) \setminus [d+1, M]} \{(x, relay(x, o_j)), (relay(x, o_j), o[d]), (o[d], x)\}$, and $\mathtt{E}_x^3 = \cup_{j \in \mathtt{Rel}(S_{o_d}, o_d)} \{(x, relay(x, o_j)), (relay(x, o_j), x)\}$. Note that $\mathtt{H}(S_{o_d}, o_d)$ is a simple directed graph, i.e., there are no multiple edges between two vertices.
(2) The *RPS request probability matrices* $\mathtt{R}_x$ for $\mathtt{H}(S_{o_d}, o_d) = (\mathtt{V}, \mathtt{E})$ and x is an element of a set in S_{o_d}, where, $\mathtt{R}_x[v_u, v_\ell]$ is the probability that the process at vertex $v_u \in \mathtt{V}_x$ forwards the client request to the server at vertex $v_\ell \in \mathtt{V}_x$, for an edge (v_u, v_ℓ) in $\mathtt{E}_x$.
(3) The *RPS blocking periods*, $(\mathtt{B}_x)_{x \in S_{o_d}}$, where $\mathtt{B}_x$ is a function over $\mathtt{V}_x$, and $x \in S_{o_d}$ refers to client or server processes. Note that the edges in $\mathtt{E}_x^i$ form a directed circle in the graph $\mathtt{H}(S_{o_d}, o_d)$, where $x \in S_{o_d}$ and $i \in [1, 3]$. A client request of $x \in \mathtt{V}_x$ to server $o[d]$, is blocking $o[d]$ for $\mathtt{B}_x(o[d]) = x.\mathtt{B}[d]$ time. A client request of $x \in \mathtt{V}_x$ to server $relay(x, o_j)$, blocks that server for a period of $\mathtt{B}_x(relay(x, o_j)) = x.\mathtt{D}[j]$, if $j \in [d+1, M]$ and $\mathtt{B}_x(relay(x, o_j)) = \mathtt{W}(x, relay(x, o_j), o[d])$ otherwise ($j \in \mathtt{Rel}(S_{o_d}, o_d) \setminus [d+1, M]$). Once the servers return to process $x \in \mathtt{V}_x$ with an answer, after a period of $\mathtt{B}_x(x) = \mathtt{T}_x$, process x sends a new client request to a server in $\mathtt{V}_x \setminus \{x\}$. Note that $x.\mathtt{B}[d]$, $x.\mathtt{D}[j]$, $\mathtt{W}(x, relay(x, o_j), o[d])$ and $\mathtt{T}_x$ are functions of x and $o[d]$, x and j, $x, relay(x, o_j)$ and $o[d]$, and respectively, x.

Note that this definition of an RPS is a special case of the definition given in [23], which is adapted to our purposes. A contention subsystem $\mathcal{CS}(S_{o_d}, o_d)$ is directly mapped to an RPS $\mathcal{RP}(\mathtt{S}_{o_d}, o_d)$, by setting $o[d] = o_d$, $t[n] = t_n$, a message to be a demand request, $\mathtt{H}(S_{o_d}, o_d) = \mathcal{H}(S_{o_d}, o_d)$, $(\mathtt{R}_x)_{x \in S_{o_d}} = (\mathcal{R}_s)_{s \in S_{o_d}}$, and $(\mathtt{B}_x)_{x \in S_{o_d}} = (\mathcal{B}_s)_{s \in S_{o_d}}$, where $d \in [1, M]$ and $n \in [1, N]$. Namely, we set $x.\mathtt{B}[d] = s.B[d]$, $x.\mathtt{D}[j] = s.D[j]$, $\mathtt{W}(x, relay(x, o_j), o[d]) = W(x, relay(s, o_j), o_d)$ and $\mathtt{T}_x = \mathcal{T}_s$, where $x \in \mathtt{S}_{o_d}$ equals to the respective $s \in S_{o_d}$ following the mapping that we described above. □

Lemma 9. *The running time of each framework iteration is $O(M \cdot N^4)$.*

Proof. The running time per iteration of the Baynat and Dallery algorithm [4] is in $O(|stations| \cdot |classes|^3)$, where $|stations|$ and $|classes|$ are the numbers of the network's stations and classes, respectively. In the context of shared-object systems, the number of *stations*, $|stations|$, corresponds to the number of vertices $|\mathcal{V}|$ of the contention graph $\mathcal{H}(S_{o_d}, o_d) = (\mathcal{V}, \mathcal{E})$ and the number of classes to S_{o_d}'s cardinality, where $d \in [1, M]$. Thus, the running time of each iteration is in $O(|\mathcal{V}| \cdot |S_{o_d}|^3)$. Notice that $|\mathcal{V}| = N + N \cdot (M-1) + 1$ and $|S_{o_d}| = N$, if $S_{o_d} = thread$, and $|\mathcal{V}| = M - \ell + 1$ and $|S_{o_d}| = 1$, if $S_{o_d} = \{o_\ell\}$, where $\ell \in [1, d-1]$. The result follows by taking the maximum $|\mathcal{V}|$ and $|S_{o_d}|$ of these two cases. □

Corollary 10. *Lemmas 7, 8 and 9 imply Theorem 6.*

10. Finding an ε-OSE

We present a procedure (Algorithm 3) for finding ε-OSEs. We give a detailed explanation of Algorithm 3 (Section 10.1) and analyze its running time (Lemma 11 of Section 10.5). We also detail the algorithm's functions (sections 10.2, 10.3, and respectively, 10.4), which are *initializeSystemState*(), *updateStates*(), and *recalcB*() and *recalc*$\mathcal{T}$().

Algorithm 3: The procedure for finding an ε-OSE.

1 **Input:** Number of objects, M and threads, N; jobs $\{job_i\}_{i\in[1,J]}$ and their arrival rates to threads $\{\lambda_{i,n}\}_{n\in[1,N],i\in[1,J]}$; R request probability matrix; ε;
2 **Variables:** The item states are recorded in t_n, $n \in [1, N]$, and o_d, $d \in [1, M-1]$, such that t_n has the form $\langle$(inter-demand period)$T[1, M]$, (delay)$D[1, M]$, (blocking period) $B[1, M]\rangle$ and o_d is of the form $\langle$(inter-demand period)$T[d+1, M]$, (delay) $D[d+1, M]$, (blocking period) $B[d+1, M]\rangle$; $\mathcal{T}_v$, item v's inter-demand period, for every $v = t_n$ such that $n \in [1, N]$ and $v = o_d$ such that $d \in [1, M-1]$; *loopEnd* (Boolean) this variable is true when the function *augmThrBlock*() decides that no OSE can be found and thus the loop should stop.
3 **Output:** $(\{t_n\}_{n\in[1,N]}, \{o_d\}_{d\in[1,M-1]}, \{\mathcal{T}_v\}_{v\in allStates}, loopEnd)$;
4 **Macros:** $objThrdSet() = \{\langle tag, i, \tau\rangle | (tag = \#o \wedge \mathcal{T}_{o_i} = \tau) \vee (tag = \#t \wedge \mathcal{T}_{t_i} = \tau)\}$;
5 $converged(prev, curr) = (\nexists \langle tag, i, \tau\rangle \in prev, \langle tag, i, \tau'\rangle \in curr : |\tau - \tau'| \geq \varepsilon)$;
6 $blocking(n) = A + \Sigma_{d=1}^{M} R(t_n, o_d) \cdot t_n.D[d]$;
7 $allStates = \{t_n | n \in [1, N]\} \cup \{o_d | d \in [1, M-1]\}$;
8 **begin**
9 $\quad initializeSystemState(N, M, \{job_i\}_{i\in[1,J]}, \{\lambda_{i,n}\})$;
10 $\quad loopEnd \leftarrow$ **false**;
11 $\quad$ **repeat**
12 $\quad\quad$ **let** $prevSet \leftarrow objThrdSet()$;
13 $\quad\quad$ **for** $d = M$ **to** 1 **do** $updateStates(\#B, d, thread, object, \{\mathcal{T}_v\}_{v\in allStates}, R)$;
14 $\quad\quad$ **foreach** $n \in [1, N]$ **do** $(\mathcal{T}_{t_n}, loopEnd) \leftarrow augmThrBlock(\Sigma_i \lambda_{i,n}, blocking(n))$;
15 $\quad\quad$ **for** $d = 1$ **to** $M-1$ **do** $updateStates(\#\mathcal{T}, d, thread, object, \{\mathcal{T}_v\}_{v\in allStates}, R)$;
16 $\quad$ **until** $converged(prevSet, objThrdSet()) \vee loopEnd =$ **true**;
17 $\quad$ **return** $(\{t_n\}_{n\in[1,N]}, \{o_d\}_{d\in[1,M-1]}, \{\mathcal{T}_v\}_{v\in allStates}, loopEnd)$;
18 **procedure** $initializeSystemState(N, M, \{job_i\}_{i\in[1,J]}, \{\lambda_{i,n}\})$ **begin**
19 $\quad$ **for** $d = M$ to 1 **do**
20 $\quad\quad$ **foreach** *item s such that* (s, o_d) *is an edge in* $\mathcal{G}$ **do**
21 $\quad\quad\quad s \leftarrow initRecord(s, d, \{t_n\}_{n\in[1,N]}, \{o_d\}_{d\in[1,M-1]})$;
22 $\quad$ **for** $n = 1$ to N **do** $\mathcal{T}_{t_n} \leftarrow augmThrBlock(\Sigma_i \lambda_{i,n}, blocking(n))$;
23 $\quad$ **for** $d = 1$ to $M-1$ **do** $\mathcal{T}_{o_d} \leftarrow \Sigma_{\ell=d+1}^{M} R(o_d, o_\ell) \cdot o_d.T[\ell]$;
24 **procedure** $updateStates(tag, d, thrSet, objSet, (\mathcal{T}_v)_{v\in allStates}, R)$ **begin**
25 $\quad \mathcal{CS}(thrSet, d) \leftarrow defContentionSubsystem(\{t_n\}_{n\in[1,N]}, \{o_d\}_{d\in[1,M-1]}, d, (\mathcal{T}_v)_{v\in allStates}, R)$;
26 $\quad (thrSet[n].T[d], thrSet[n].D[d])_{n\in[1,N]} \leftarrow BDF(\mathcal{CS}(thrSet, d))$;
27 $\quad$ **for** $j = 1$ **to** $d-1$ **do**
28 $\quad\quad \mathcal{CS}(\{objSet[j]\}, d) \leftarrow defContentionSubsystem(\{objSet[j]\}, \{o_d\}_{d\in[1,M-1]}, d, (\mathcal{T}_v)_{v\in allStates}, R)$;
29 $\quad\quad (objSet[j].T[d], objSet[j][n].D[d]) \leftarrow BDF(\mathcal{CS}(thrSet, d))$;
30 $\quad$ **if** $tag = \#B$ **then** $recalcB(d)$, **else if** $tag = \#\mathcal{T}$ **then** $recalc\mathcal{T}(d)$;

10.1. The ε-OSE solver

The procedure always halts and computes an approximate equilibrium, ε-OSE, when such is reachable. Namely, whenever the job arrival and completion rates differ by at most ε, the procedure returns the system state in an ε-OSE, or indicates that an OSE is not a state that the system can be in. The procedure sets initial values to the system state, $c[s, o_d]$, and then uses the proposed methods (sections 6 to 9) for estimating $c[s, o_d]$ iteratively, until convergence. The decision on when to stop considers the system inter-demand period, $\{\mathcal{T}_{item}\}_{item\in V\setminus\{o_M\}}$, and stops whenever there is no $item \in V \setminus \{o_M\}$ for which the change in $\mathcal{T}_{item}$ is greater than ε since the previous iteration, where $\mathcal{G} = (V, E)$ is the acquisition graph.

The procedure's input includes the system parameters, i.e., number of threads N and objects M, the jobs, job_i, $i \in [1, J]$, and their arrival rates $\{\lambda_{i,n}\}_{i\in[1,J],n\in[1,N]}$ to each t_n, $n \in [1, N]$, as well as the request probability matrix, R. The algorithm uses the Boolean variable *loopEnd*, which is true when the algorithm decides that no OSE can be found. The procedure's output includes the delay D, inter-demand period T and blocking period B between all system items, as well as, the item inter-demand periods $\mathcal{T}_v$, for every item $v \neq o_M$, and *loopEnd*.

The procedure starts with a system state that represents the case in which all queues are empty (see the function *initializeSystemState*()). It then estimates the state of a system in which threads can block one another, and the delay grows as more requests are pending in the queues. The main part of the pseudocode (Algorithm 3) consists of a repeat-until loop (lines 11–17) that follows the procedure's initialization (line 9). Before the procedure can return its output value, the loop has to end either when the ε-OSE conditions are satisfied or when the procedure detects that an OSE cannot be reached. Each iteration aims at further improving the ε-OSE estimation. The repeat-until loop exits when no item changes by at least ε between every two iterations (and thus the system state satisfies the conditions of an ε-OSE).

In every iteration, the procedure computes (i) the blocking periods of demand requests for objects, $s.B[d]$ (the function *updateStates*()), (ii) the thread inter-demand periods, $\mathcal{T}_{t_n}$ (the function *augmntThreadBlock*()), and (iii) the object inter-demands, $\mathcal{T}_{o_d}$ (which is the function *updateStates*()), where (s, o_d) is an edge of the acquisition graph $\mathcal{G}$, $n \in [1, N]$ and $d \in [1, M-1]$. The repeat-until loop repeats the steps (i), (ii) and (iii), which deals with interdependencies using alternating backward and forward iterations. Namely, it resolves the forward dependencies in which (s, o_d)'s blocking period, $s.B[d]$, depends on o_ℓ's delay by iterating backward, where $\ell \in (d, M]$, i.e., starting from $d = M$ and counting downwards, we can estimate $s.B[d]$, because (in a system that its state satisfies the equilibrium conditions) all of (s, o_d)'s forward dependencies can be resolved. Similarly, it uses forward iterations for resolving backward dependencies with respect to o_d's item inter-

demand period, $\mathcal{T}_{o_d}$, because all of o_d's backward dependencies are resolved. Moreover, the function $augmntThreadBlock()$ allows the repeat-until loop to stop whenever the job inter-arrival time becomes less or equal than the time it takes that thread to complete such jobs, i.e., there's no OSE.

The blocking period estimation, step (i), starts from the last system object, o_M, where there are no dependencies on the delay of subsequent demand requests. For every s such that (s, o_M) is an edge of $\mathscr{G}$, we calculate the delay, $s.D[M]$, and the pairwise inter-demand period, $s.T[M]$, of demand requests to o_M through the $updateStates(\#B, d, thread, object, \{\mathcal{T}_v\}_{v \in allStates}, R)$ function (line 13), where $allStates = \{t_n \mid n \in [1, N]\} \cup \{o_d \mid d \in [1, M-1]\}$. Therefore, the procedure can compute the blocking period of demand requests to o_{M-1}, $s.B[M-1]$, because it has just estimated the dependencies of subsequent demand requests for o_M. Repeating this process for $d = M-1, \ldots, 1$, the procedure compute the blocking periods of demand requests to any o_d, $s.B[d]$, where $d \in [1, M]$. Note that this is possible, since in every step d, $d = M-1, \ldots, 1$, of this for loop, we have already computed the demand request delays and the inter-demand periods of every $o_{d'}$, $o_d.D[d']$, and respectively, $o_d.T[d']$, where $d' \in [d+1, M]$ (Section 7).

After computing the blocking periods, delays and pairwise inter-demand periods, the procedure estimates the threads' inter-demand periods (Section 8.1), which can be used to estimate the job completion rates. It does this in step (ii), through the function $augmntThreadBlock(\Sigma_i \lambda_{i,n}, blocking(n))$ (line 14) and by using the thread idle probability (Section 3), where $blocking(n) = A + \Sigma_{d=1}^{M} R(t_n, o_d) \cdot t_n.D[d]$. Moreover, note that the $augmntThreadBlock()$ function (Section 8.1) allows the repeat-until loop to stop whenever it detects that the inter-arrival time of jobs to a thread becomes less or equal than the time it takes that thread to complete such jobs (i.e., there's no OSE). Note that the repeat-until loop breaks when the Boolean variable $loopEnd$ is true (line 14).

Step (iii) uses the $updateStates()$ function (line 15) for calculating the object inter-demand periods, after calculating the new estimates for the delay and the pairwise inter-demand periods. The procedure estimates the inter-demand period $\mathcal{T}_{o_d}$, for each o_d, $d \in [1, M-1]$, via the item inter-demand period of demand requests for items that precede o_d in a job path $(\bullet, item_j, o_d, \bullet)$ (Section 8.2). Therefore, the procedure estimates the inter-demand periods for each o_d in the order of $d = 1, \ldots, M-1$.

Once the procedure verifies the satisfaction of the ε-OSE conditions (Section 5), the repeat-until loop end (line 16) and the procedure returns.

10.2. The $initializeSystemState()$ function

The procedure $initializeSystemState()$ initializes Algorithm 3 assuming that there is no contention, i.e., all queues have zero length in the shared-object system (lines 9 and 18). In the context of shared-object systems, no contention means that the delay of each demand request equals its blocking period, $s.D[d] = s.B[d]$, for all items s and o_d. Thus, the procedure initializes the blocking periods $s.B[d]$ through the function $initRecord()$ (lines 19–21). It uses the average demand completion period f_{s,o_d}, as proposed in Section 7.1, which is a random variable with a known distribution. That is, $s.B[M] = A + R(s, o_d) \cdot R(o_d, o_d) \cdot f_{s,o_d}$, if $d = M$ and $s.B[d] = A + R(s, o_d) \cdot R(o_d, o_d) \cdot f_{s,o_d} + \Sigma_{\ell=d+1}^{M} R(d, o_\ell) \cdot d.D[\ell]$, if $d \in [1, M-1]$, where $s \in \{t_n \mid n \in [1, N]\} \cup \{o_i \mid i \in [1, d-1]\}$. Namely, the blocking period of a request to access (demand) the last object equals to the acquisition period plus the average demand completion period on that object and the blocking period of a demand to o_d, $d \in [1, M-1]$, is recursively computed as the sum of the acquisition period, A, the average demand completion period $R(s, o_d) \cdot R(o_d, o_d) \cdot f_{s,o_d}$ plus the average blocking period of a subsequent demand to one of the following objects $d.B[\ell]$, $\ell \in [d+1, M]$, weighted by the probability of sending such a demand, $R(d, o_\ell)$, $\ell \in [d+1, M]$. Moreover, the $s.T[d]$ pairwise inter-demand periods are set to equal $s.B[d]$.

The $\mathcal{T}_{t_n}$ (thread) inter-demand periods are computed (line 22) through the function $augmThrBlock(\Sigma_i \lambda_{i,n}, blocking(n))$, where $blocking(n)$ is defined in line 6. Furthermore, the $\mathcal{T}_{o_j}$ (object) inter-demand periods are set to $\Sigma_{\ell=j+1}^{M} R(o_j, o_\ell) \cdot o_j.T[\ell]$ (line 23). Algorithm 3 iteratively finds the correct values of the pairwise and item inter-demand periods (arbitrary initialization of the system's state is proposed in [23]).

10.3. The $updateStates()$ function

This function updates the blocking periods and the item inter-demand periods with respect to o_d (lines 24–30). If the input tag is $\#B$, the function updates the blocking periods $s.B[d]$ for every $s \in \{t_n \mid n \in [1, N]\} \cup \{o_j : j \in [1, d-1]\}$, as in Section 7. Otherwise, if the input tag is $\#\mathcal{T}$, the function updates the inter-demand period of o_d, $\mathcal{T}_{o_d}$, as in Section 8.

The procedure $updateStates()$ defines the contention subsystem for every set of item sources S_{o_d} using the function $defContentionSubsystem()$ (line 25 if $S_{o_d} = thread$ and line 28 if $S_{o_d} = \{o_j\}$, $j \in [1, d-1]$). It then calculates the delay, $s.D[d]$, and pairwise inter-demand period, $s.T[d]$, for every $s \in S_{o_d}$ using the $BDF()$ function of Section 9.2 (line 26 if $S_{o_d} = thread$ and line 29 if $S_{o_d} = \{o_j\}$, $j \in [1, d-1]$). The procedure ends with the computation of the blocking periods or the inter-demand periods, depending on the tag with which the procedure was called.

10.4. The $recalcB()$ and $recalc\mathcal{T}()$ functions

We present the exact formulas that give the first three moments of $s.B[d]$, i.e., the function $recalcB()$, and $\mathcal{T}_{o_d}$, i.e., the function $recalc\mathcal{T}()$. We find these formulas using the equations in sections 7, and respectively, 8.

Let $F_{s,o_d} = A + R(s, o_d) \cdot R(o_d, o_d) \cdot f_{s,o_d}$, be the sum of the acquisition time and the job completion period times the related probabilities (A and F_{s,o_d} can be bound by a constant and thus can be treated as such). The first three moments of the blocking time for $d = M$ are

$$E(s.B[M]^m) = (F_{s,M})^m, \text{ for } m = 1, 2, 3 \tag{16}$$

and if $d \in [1, M-1]$, we have that

$$E(s.B[d]) = F_{s,o_d} + \Sigma_{d'=d+1}^{M}(\Pr[(s, \bullet, d, o_{d'})] \cdot E(d.B[d']) \tag{17}$$

$$E(s.B[d]^2) = (F_{s,o_d})^2 + \Sigma_{d'=d+1}^{M} \Pr[(s, \bullet, d, o_{d'})](E(d.D[d']^2) + 2F_{s,o_d} \cdot E(d.D[d'])) \tag{18}$$

$$E(s.B[d]^3) = (F_{s,o_d})^3 + \Sigma_{d'=d+1}^{M} \Pr[(s, \bullet, d, o_{d'})](E(d.D[d']^3) + 3(F_{s,o_d})^2 \cdot E(d.D[d']) + 3F_{s,o_d} \cdot E(d.D[d']^2)) \tag{19}$$

Moreover, the first three moments of $\mathcal{T}_{o_d}$ are

$$E(\mathcal{T}_{o_d}) = \frac{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d) \cdot (F_{s,o_d} + E(\mathcal{T}_s))}{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d)} \tag{20}$$

$$E(\mathcal{T}_{o_d}^2) = \frac{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d) \cdot ((F_{s,o_d})^2 + E(\mathcal{T}_s^2) + 2F_{s,o_d} \cdot E(\mathcal{T}_s))}{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d)} \tag{21}$$

$$E(\mathcal{T}_{o_d}^3) = \frac{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d) \cdot ((F_{s,o_d})^3 + E(\mathcal{T}_s)^3) + 3(F_{s,o_d})^2 E(\mathcal{T}_s) + 3F_{s,o_d} E(\mathcal{T}_s^2))}{\Sigma_{s \in arrivals(o_d)} \omega(s, o_d)} \tag{22}$$

We remind that $arrivals(o_d) = thread \cup \{o_j \mid j \in [1, d-1]\}$ and $\omega(s, o_d) = R(s, o_d) \cdot s.T[d]$ (Section 8.2).

10.5. Running time

Notice that the running times of Algorithm 2 (Baynat–Dallery framework) and Algorithm 3 depend on the number of iterations of these algorithms. Lemma 11 bounds the procedure running time for one iteration.

Lemma 11. *The running time of one iteration of Algorithm 3 is in $O(M^2 \cdot N^4 + M^3)$.*

Proof. We look at the running time of each step of the repeat-until loop to find the algorithm's running time. Steps (i) and (iii) call the function $updateStates()$ so at most M; M, and respectively $M-1$ times. Note that the function $updateStates()$ calls at most $1 + (M-1)$ times the function $BDF()$, because the input parameter d, which denotes the object whose state is to be updated, is at most M (line 27). The first call is done by setting $S_{o_d} = thread$ and (at most) $M-1$ calls are done by setting $S_{o_d} = \{o_j\}$, $j \in [1, d-1]$. The $BDF()$ function has running time in $O(M \cdot N^4 \cdot I_f)$ and $O(M \cdot I_f)$ (Lemma 9), when $BDF()$ is called for $S_{o_d} = thread$, and respectively, for $S_{o_d} = \{o_j\}$, $j \in [1, d-1]$, where I_f denotes the maximum number of framework iterations (Lemma 9). It also holds that the function $augmntThreadBlock()$ is called N times in step (ii) and it's the running time is practically constant (see Section 3 with respect to the findings of Latouche and Ramaswami [16]). Thus, the running time of one iteration of Algorithm 3 is in $O(M \cdot (M \cdot N^4 \cdot I_f) + N + M \cdot ((M-1) \cdot M \cdot I_f)) = O(M^2 \cdot N^4 + M^3)$. □

11. Conclusions

We consider a resource allocation problems that can be modeled as generalized dynamic dining philosophers problems. We formulate questions that are associated with equilibrium situations in such systems, where input and output rates match. We believe that the way we find the equilibrium as well as estimate the delay and throughput in such systems can be the basis for an analysis of further generalizations of the problem studied here, such as the ones that are described in the literature on resource allocation, e.g., non-sequential scheduling, such as parallel resource acquisition (2-phase locking) and resource acquisition that is reactive to contention conditions [14,19,10,11,22]. Another research direction is to consider the approach proposed by Reif and Spirakis [24] that does deterministic object acquisition via FIFO queues. They rather consider resource granting systems for satisfying probabilistically the changing user requests for resource allocation, using local communication between granting and requesting processes.

References

[1] Ivo Adan, Jacques Resing, Queuing Theory, Eindhoven University of Technology, Eindhoven, 2002, can be accessed via www.win.tue.nl/~iadan/queueing.pdf.
[2] Altiok Tayfur, On the phase-type approximations of general distributions, IIE Trans. 17 (2) (1985) 110–116.
[3] Forest Baskett, K. Mani Chandy, Richard R. Muntz, Fernando G. Palacios, Open, closed, and mixed networks of queues with different classes of customers, J. ACM 22 (2) (1975) 248–260.
[4] Bruno Baynat, Yves Dallery, A product-form approximation method for general closed queuing networks with several classes of customers, Perform. Eval. 24 (3) (1996) 165–188.

[5] Gunter Bolch, Stefan Greiner, Hermann de Meer, Kishor S. Trivedi, Queuing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications, John Wiley & Sons, 2006.
[6] Jeffrey P. Buzen, Computational algorithms for closed queuing networks with exponential servers, Commun. ACM 16 (9) (1973) 527–531.
[7] Junghoo Cho, Hector Garcia-Molina, Synchronizing a database to improve freshness, in: Weidong Chen, Jeffrey F. Naughton, Philip A. Bernstein (Eds.), SIGMOD Conference, ACM, 2000, pp. 117–128, SIGMOD Rec. 29 (2) (June 2000).
[8] Richard Martin Feldman, Ciriaco Valdez-Flores, Applied Probability and Stochastic Processes, Springer, 2010.
[9] William J. Gordon, Gordon F. Newell, Closed queuing systems with exponential servers, Oper. Res. 15 (2) (1967) 254–265.
[10] Phuong Hoai Ha, Marina Papatriantafilou, Philippas Tsigas, Self-tuning reactive distributed trees for counting and balancing, in: Teruo Higashino (Ed.), Principles of Distributed Systems, 8th International Conference, Revised Selected Papers, OPODIS 2004, Grenoble, France, December 15–17, 2004, in: Lecture Notes in Computer Science, vol. 3544, Springer, 2004, pp. 213–228.
[11] Phuong Hoai Ha, Philippas Tsigas, Reactive multi-word synchronization for multiprocessors, J. Instr.-Level Parallelism 6 (2004) 1–25.
[12] Mor Harchol-Balter, Takayuki Osogami, Alan Scheller-Wolf, Adam Wierman, Multi-server queuing systems with multiple priority classes, Queuing Syst. 51 (3–4) (2005) 331–360.
[13] Timothy L. Harris, Keir Fraser, Ian A. Pratt, A practical multi-word compare-and-swap operation, in: International Symposium on Distributed Computing, Springer, 2002, pp. 265–279.
[14] Maurice Herlihy, Nir Shavit, The Art of Multiprocessor Programming, Elsevier, 2012, revised reprint.
[15] James R. Jackson, Jobshop-like queuing systems, Manage. Sci. 50 (12 Suppl.) (2004) 1796–1802.
[16] Guy Latouche, V. Ramaswami, A logarithmic reduction algorithm for quasi-birth-death processes, J. Appl. Probab. 30 (3) (1993) 650–674.
[17] Victor Luchangco, Mark Moir, Nir Shavit, Nonblocking k-compare-single-swap, in: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2003, pp. 314–323.
[18] Nancy A. Lynch, Upper bounds for static resource allocation in a distributed system, J. Comput. System Sci. 23 (2) (1981) 254–278.
[19] Nancy A. Lynch, Distributed Algorithms, Morgan Kaufmann, 1996.
[20] Marcel F. Neuts, Matrix-Geometric Solutions in Stochastic Models – An Algorithmic Approach, Dover Publications, 1994.
[21] Takayuki Osogami, Analysis of Multi-Server Systems via Dimensionality Reduction of Markov Chains, PhD thesis, IBM, Tokyo, 2005.
[22] Marina Papatriantafilou, Philippas Tsigas, On distributed resource handling: dining, drinking and mobile philosophers, in: Alain Bui, Marc Bui, Vincent Villain (Eds.), On Principles of Distributed Systems, Proceedings of the 1997 International Conference, Chantilly, France, December 10–12, Hermes, 1997, pp. 293–308.
[23] Sridhar Ramesh, Harry G. Perros, A multi-layer client-server queuing network model with non-hierarchical synchronous and asynchronous messages, Perform. Eval. 45 (4) (2001) 223–256.
[24] John H. Reif, Paul G. Spirakis, Real time resource allocation in distributed systems, in: Robert L. Probert, Michael J. Fischer, Nicola Santoro (Eds.), ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, August 18–20, 1982, ACM, 1982, pp. 84–94.
[25] Iosif Salem, Elad M. Schiller, Marina Papatriantafilou, Philippas Tsigas, Shared-object system equilibria: delay and throughput analysis, arXiv:1508.01660, 2015.
[26] Iosif Salem, Elad M. Schiller, Marina Papatriantafilou, Philippas Tsigas, Shared-object system equilibria: delay and throughput analysis, in: Proceedings of the 17th International Conference on Distributed Computing and Networking, ICDCN '16, ACM, New York, NY, USA, 2016, pp. 30:1–30:10.