Università
Ca'Foscari
Venezia

Corso di Laurea magistrale (*ordinamento ex D.M. 270/2004*)
in Informatica – Computer Science

Tesi di Laurea

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Performance evaluation of garbage collection policies

Relatore
Dott. Andrea Marin

Laureando
Alberto Amadio
Matricola 806052

Anno Accademico
2012 / 2013

Acknowledgements

This thesis concludes my Master studies.
There are many people I have to thank starting from my supervisor, Dr. Andrea Marin, for his valuable guidance and advice.
Thank to Gian-Luca Dei Rossi, Ph.D., who helped me with the models analysis and performance evaluation.
I want also thank the technical staff of Department of Environmental Sciences, Informatics and Statistics (DAIS) for providing the environment and facilities to complete this work.
A special thanks to my family, who always supported me during those years of study full of events.
I have to remember all the other colleagues and friends for giving me extra motivation during those two years.
Finally, I dedicate this thesis to the memory of my grandfather, who passed away when my thesis was nearing completion.

Abstract

Many modern programming languages allow the programmer to allocate the memory in a simple and transparent way, without the need of the explicit deallocation of the memory when an object is not necessary any more. This is achieved by means of the Garbage Collectors, i.e., special threads that use some algorithm to mark unused objects and then reclaim their space. However, the drawback of this approach is twofold. First, the response time of an application using the garbage collection is generally worse than that of an equivalent that explicitly allocates and deallocates the memory because of the process time required by the garbage collection itself. Second, the algorithms used by the collectors are usually CPU-intensive and hence cause a high consumption of CPU-cycles and a consequent waste of energy.

The goal of this thesis is to statistically characterise the memory allocation requirements of some classes of applications, to provide numerically tractable models to predict some performance indices of the system, i.e., throughput and average response time, given different garbage collection policies, and to validate those models through a comparison with experimental results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis aims of studying the impact of Java's garbage collection policies in some types of programs and providing numerically tractable models to predict a set of performance indices of this policies. Both of the stochastic models presented have an underlying Markov chain with the particular structure of Quasi-birth and death processes. This characteristic allows the application of the matrix-geometric method, a technique to compute a numerical tractable solution of the steady-state distribution. These models are then validated through experimentations on a testing environment specifically designed to resemble a real scenario. The parametrization is done according to a set of measurements which are done during the server working-time. Average response time is also measured and used to validate the models.

The document is organized into 7 chapters as following. The first chapter, named Introduction, provides the basic concepts of automatic memory management describing the most fundamental garbage collection techniques and algorithms. Furthermore, performance metrics and design goals are discussed.

Chapter 2, named Java HotSpot Virtual Machine, describes the technology employed to manage the memory and summarizes the design characteristics and the performance goals of the available garbage collectors.

Chapter 3, named Theoretical Background presents the fundamental concepts of Markow Chains and Queueing theory, which are useful to understand the following two chapters.

Chapter 4 and 5, namely A QBD Model for GC and Markow-Modulated Queueing Model for GC respectively, discuss two stochastic models to predict the performances of a garbage collection policy. Moreover, theory con-

cepts and resolution methods previously discussed are used to define the models and to find their numerical solutions.

In Chapter 6, named Validation, the models are validated showing a comparison between experimental measures and model predictions. This chapter describes in detail the test environment, discussing the hardware and software employed, the problems encountered and the applications chosen for the validation.

Finally, in Chapter 7 we discuss some final remarks and possible future works.

1.1 Automatic memory management

Many modern programming languages implements a garbage collection mechanism which allows the programmers to use the memory in a transparent way. In explicit deallocation languages like C, the programmer is responsible for freeing memory of unused variables. As an example, in C functions *malloc()* and *free()* are used to allocate and deallocate memory blocks, respectively. Without discipline it is very difficult to keep track of all the references in the programs, i.e., pointers to memory, especially in complex codes with many lines. Many parts of a typical code share objects and a programmer may incur in two errors: he/she may free the memory too early or forget to do it at the end of data usage. To avoid programming errors and memory problems like memory corruption and memory exhaustion, modern languages have introduced garbage collectors.

A garbage collector is a special algorithm which automatically frees the memory from unused objects which are called *garbage*. Garbage collection is implemented by Sun's Java Language, Microsoft's Common Language Runtime and also by wide-spread scripting languages such as Perl and Python. Garbage collectors exist also for languages with explicit memory management such as the Boehm– Demers–Weiser garbage collector, known as Boehm GC, which is a library for C and C++. The task carried out by a garbage collector is not trivial and over the years several research efforts have been denoted to define optimization policies in order to allow better performances.

1.2 State of the Art

While garbage collection's software engineering benefits are unquestionable, its performance impact is very difficult to quantify and remains controversial. Several studies were conducted to measure the performances of garbage collectors and how they affect the program execution. Diwan *et al.* [7] use trace-driven simulations to conclude that generational garbage collection involves on the worst-case about of 50% of execution time.

Other researchers [9, 8] tried to compare automatic memory management versus explicit memory managers, showing that, in general, languages which implement garbage collection algorithms are less efficient. However, a directly comparison on costs is not possible because programs written in those languages, e.g. Java, do not contain calls to *free* function. For the purpose, Hertz and Berger [9] prosed a new methodology to treat Java program as if they used explicit memory management by the use of oracles which insert calls to *free*. Their results show that the runtime performance of some garbage collectors are competitive with explicit memory management when given enough heap size [8]. They also show that garbage collectors performances decrease when paging occurs.

A very complete study on the impact of garbage collection were conduced by Blackburn *et al.* [4], which examine the behaviour of different collectors: copying semi-space, mark-sweep and reference counting. With experimental results they tried to explain the direct and indirect costs of garbage collection as a function of heap size, in order to guide the users to choice the right collector.

Buytaert *et al.* [5] proposed GCH, a profiled method for guiding garbage collection. Experimental results obtained with SPECjvm98, a benchmarks suite, and two generational garbage collectors show significant reductions on the time required for collections and that the the overall execution time can be reduced by more than 10%. Authors also cite other related works that aim at providing some criteria to select the most appropriate garbage collector give the characteristics of the running application.

However, to the best of our knowledge, the literature does not present any queueing model to predict garbage collection performance except [1]. Our contribution is the continuation to the research carried out by Balsamo *et al.* [1], which propose a queueing model to analyse a system with a garbage collector. We validate this model with real-experiments and we propose Markov-modulated queueing model, a simplified version which try to involve

less parameters. Furthermore, we compare the models analysed and we discuss the experimental results.

1.3 Performance metrics

A garbage collector should provide high application throughput, high memory efficiency and high responsiveness. A modification on one of these characteristics affects the others, thus become important to find a trade-off between them. Setting a heap size too small to preserving memory space, has the consequence of a low throughput and responsiveness, because the garbage collector runs with high frequency. Conversely, if with a large heap the garbage collector runs more rarely with improvements in terms of throughput and system responsiveness, the disadvantages may be the waste of resources and memory fragmentation.

Unfortunately, it is not possible to identify the "best" collector for all possible configurations. Thus, we are going to describe the performance metrics that are useful to chose the most suitable garbage collection algorithm. These metrics are:

- *Throughput*, the percentage of total time not spent in garbage collection, considered over long periods of time.
- *Pause time*, the time when an application appears unresponsive because garbage collection is occurring.
- *Footprint*, the working set of a process, measured in pages and cache lines.
- *Promptness*, the time between when an object becomes dead and when the memory becomes available.

Users have different requirements of garbage collection. For example, some consider the right metric for a web server to be throughput, since pauses during garbage collection may be tolerable, or simply obscured by network latencies. However, in an interactive graphics program even short pauses may negatively affect the user experience. On systems with limited physical memory or many processes, footprint may dictate scalability, while promptness is an important consideration in distributed systems that use the Remote Method Invocation (RMI).

1.4 Algorithms for GC

This section introduces the job carried out by a garbage collection algorithm and the models at the base of garbage collection schemes.

Dynamic memory allocation allows objects to be store in the *Heap*, even if their size is not known at the compiling time. These objects are accessed through *references*, i.e, pointers that address the memory location where an object is stored. Garbage collector's task is to free the memory when there is no pointer from a reachable object, avoiding possible memory leaks derived from programming errors. A garbage-collected program is divided into two semi-independents part [10]:

- The *mutator*, which executes the application code. It is responsible for object allocation and mutates the graph of the objects by changing reference fields. As a result if an object is disconnected, it becomes *unreachable*.
- The *collector*, which executes the garbage collector algorithm to discover unreachable objects and recall their storage.

More specifically, a mutator load pointers from the current set of root objects, which are heap elements accessed directly by the mutator. When an object is unreachable, i.e, a mutator thread has removed all pointers, it cannot be reached again and it can be safety reclaimed by the collector. A garbage collector is correct only if it never reclaims live objects, i.e., reachable objects. A collector is called mainly when a mutator can not allocate an object for insufficient memory space. In general, there are different conditions to call a collector that depend on the algorithm used and the policies adopted.

Once the collector has completed its job, the mutator repeats the allocation request. For example, in a Java code the request is performed by the *New* operator. If the operation fails again for insufficient heap space the program raises an exception. We assume that the mutator can run many threads and there is only a single collector thread. We also assume that while the collector is running all mutator threads are stopped. This policy, called *stop-the-world*, simplifies the design of collectors and we adopt it for the stochastic models presented in this work.

All garbage collection schemes are based on one of four fundamental approaches:

1. mark-sweep collection;

2. mark-compact collection;
3. copying collection;
4. reference counting.

Now we summarize their main characteristics. Finally, we conclude with generational garbage collection which is as the base of G1 algorithm of Sun Microsystems' HotSpot Java virtual machine. For a detailed explanation on schemes presented, refer to The Garbage Collection Handbook [10].

1.4.1 Mark-sweep collection

Mark-sweep collection [10] is an algorithm developed by McCarthy on 1960 and it is a *tracing* algorithm. This kind of algorithms traverse the heap, or a portion of the heap, to determine which objects are reachable and which can be reclaimed. Mark-sweep's name comes from the two phases: *mark* and *sweep*. In the mark phase the collector start traversing the graph of the objects marking encountered objects. Then, in the sweeping phase the collector examines every object: any unmarked element is considered garbage and its space is reclaimed. It is an indirect algorithm because garbage is not collected directly, like in reference counting.

Despite its age, it performs well and remains a valid choice. Since the simplest form imposes no overhead on mutator read and write operations it is also used as a base algorithm for more complex collectors. It offers a good throughput if combined with lazy sweeping, a technique to reduce sweeping costs by delegating the operation to the *allocator*. However, for the *stop-the-world* condition, it requires that all the mutators to be stopped while collectors run and the pause time depends on the program and on the underling system. Since it is a tracing algorithm it requires a large heap. With small heaps the collector is called more frequently, whereas the mark phase is very expensive and hence should be done infrequently. Mark-sweep is also vulnerable to fragmentation, since it does not relocate data in the heap.

1.4.2 Mark-compact collection

Using non moving-collectors the heap can be affected by fragmentation. The available heap space can result insufficient to allocate an object or an allocation operation requires excessive time to find contiguous free space, resulting in an overall performance degradation. A first solution is performed

by the allocators, which can store small objects of same size together in blocks, but with non moving-collects the fragmentation problem still remain resulting in low system performance. Therefore, we can use a new kind of algorithms that operate like mark-sweep and also compact live objects in the heap.

Mark-compact algorithms [10] have more phases than mark-sweep. They start with a marking phase, as described previously. Then, they compact the heap relocating objects and updating the references of moved data. They are moving collectors and the number of compacting phases depend on the specific algorithm which is used, as well as the way in which the operation is carried out. The time required to perform the compacting phase is mainly affected by locality. There are three possible ways in which data may be arranged in the heap:

- *Arbitrary.* Objects are relocated without regard for their original order.
- *Linearising.* Objects are relocated so that they are adjacent to related objects.
- *Sliding.* Objects are slid to one end of the heap, squeezing out garbage, thereby maintaining their original allocation order in the heap.

Compactors with arbitrary order are the most simple to implement, but modern collectors implement the sliding compaction.

Mark-compact algorithms are suitable when heap is large because the compaction strategy results in a faster sequential allocation. However, to have a compacted heap there is a price to pay in terms of additional overheads. Since these collectors tend to increase the number of phases, the resulting throughput is generally worse that of mark-sweep. A common solution is to use mark-sweep collectors as default and switch to mark-compact collection when there is heap fragmentation (determined by some metrics). As we will discuss more in details on the next section, Sun Microsystems' HotSpot Java Virtual Machine uses mark-compact for its old generation when running Serial Collector.

1.4.3 Copying collection

Copying collection [10] is another tracing garbage collection which improves the compaction phase reducing the collection times. Unlike mark-compact collection it requires only a single phase to compact the heap, but reducing

heap size by half. In copying collection, the heap is divided into two regions of equal size called *fromsapce* and *tospace*. Objects are allocated in the fromspace region, until it fills up. When there is no more sufficient space, the collector explores the region marking and copying reachable objects to tospace. Then the collector switches the role of the regions and starts to allocate data in the new fromspace. Old objects can be safely overwritten during the next round. With this solution free space is always contiguous and not affected by fragmentation.

Copying collection offers advantages over previous collectors: fast allocation and elimination of fragmentation. However, this costs a lot in terms of space requirements and it is necessary to sacrifice the half of the heap size. With two regions to maintain, collectors will perform more garbage collection cycles and in terms of performance this depends on trade-off between mutators and collectors, the space available and the characteristics of the application.

1.4.4 Reference counting

Reference counting [10] operates directly on objects. It maintains a counter for each objects in the heap to count the number of references. Once the counter is zero, the corresponding object can be freed and the reference counts of all its children, which are reachable from the graph of objects, are decremented. Reference counting distributes the process of memory management throughout the program. Objects are reclaimed as soon as there are no references to them, and the process is local to each object. Heap space is freed faster than tracing collectors, which waits until the heap is almost full before being invoked. Unfortunately, counters need to be updated every time a reference changes and this introduces extra work. Pure reference counting collectors can not deal with cycles. This problem can be handled by a hybrid system which uses reference counting as default collector and periodically calls a tracing method to collect cycles.

Reference counting algorithms are particular suitable in programs where most of the objects are sufficient simple to be managed explicitly. As an example data like bitmaps will not contain any pointers, so the problem of reclaiming cycles does not exist. Furthermore, they have good locality properties and can reclaim the space of an object as soon as the last reference is removed.

1.4.5 Generational garbage collection

Generational garbage collection is the latest class of algorithm that we describe and is at the base of the collectors available in Sun Microsystems' HotSpot Java Virtual Machine, like G1 [6], a new collector available starting from version 7. Previous methods, tracing and copying collectors, are suitable in presence of few live objects. However, they are inefficient if the algorithms process repeatedly long-lived objects, e.g, moving from one region to another. Furthermore, mark-compact collectors tend to accumulate data in the bottom of the heap and some collectors avoid compacting this area.

Generational collectors make a distinction between youngest and oldest objects, privileging reclamation effort on youngest data. This approach is based on the hypothesis that short-lived objects die young, thus the idea is to maximise the reclaimed space while minimising the effort. Therefore, heap space is organized into *generations*, which are distinct regions containing objects split by age. Younger generations have priority on collections, which usually are frequent and fast. Data that survive long enough are promoted (the technical word is *tenured*) from the generation being collected to an older one. Old generation occupancy grows more slowly and is larger than the young generation. This results in infrequent collections, but significantly longer to complete.

The performances of this class of algorithms depend on the expected pause times for collection of a generation. In general, time taken to collect the youngest generation, usually called *nursery*, depends on its size. By tuning its size it is possible to control collection timing. Since collections are frequent, for a young generation a high speed collector is usually chosen. Conversely, for old generation a more space efficient algorithm is preferred. The throughput can be improved by reducing the collection of long-live objects, though this garbage cannot be reclaimed. We recall thaht tuning generational collectors, and in general garbage collection algorithms, to meet throughput and pause-time goals simultaneously is a very difficult task.

Chapter 2

Java Hotspot Virtual Machine

This section introduces the Java HotSpot Virtual Machine, the technology originally developed by Sun Microsystems, Inc. and now maintained by Oracle Corporation. This Java Virtual Machine (JVM) is the core component of the Java SE platform, which allows programmers to develop and deploy Java applications on desktops and servers.

According to its official description says, it implements the Java Virtual Machine specification, and is delivered as a shared library in the Java Runtime Environment. As the Java bytecode execution engine, it provides Java runtime facilities, such as thread and object synchronization, on a variety of operating systems and architectures. It includes dynamic compilers that adaptively compile Java bytecodes into optimized machine instructions and efficiently manages the Java heap using garbage collectors, optimized for both low pause time and throughput. It provides data and information to profiling, monitoring and debugging tools and applications.

2.1 Memory management

Memory is organized into three generations, as described in [17]. In addition to a young and an old generation, which we defined on Section 1.4.5, there is a *permanent generation*. This space is reserved by the JVM to store useful data like classes and method descriptions. Young generation is also split in other small regions: one called *Eden* and two smaller *Survivor* spaces. Initially, objects are allocated in the Eden space. One survivor space is kept empty at any time, and serves as the destination of any live objects in Eden

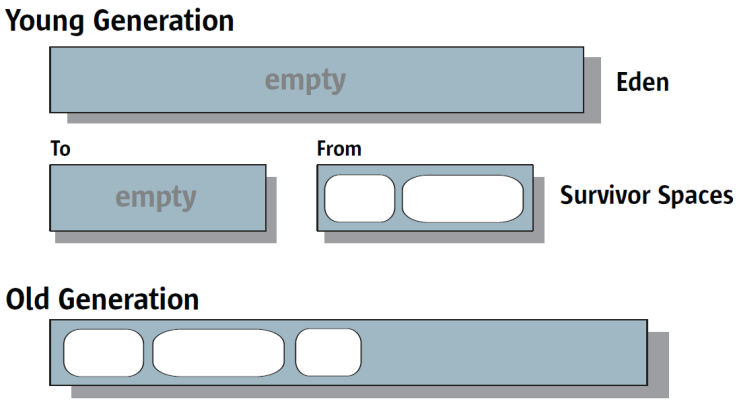

Figure 2.1: Generations after a minor collection.

and the other survivor space during the next copying collection. Objects are copied between the survivor spaces in this way until they are old enough to be *tenured* to the old generation. In case of collection on the young generation the operation is called *minor collection*, while for collection on old generation the operation is called *major collection*. If the old generation cannot receive promoted object, the algorithm used for major collection is performed on the whole heap. Figure 2.1 shows an example of the generations after a minor collection, where only one of the survivor space and old generations contain live objects.

HotSpot can handle multithreaded applications adopting a technique called *Thread-Local Allocation Buffers* (TLABs). Multithreadding programming requires particular attention when access a resource and usually this is done by global locks and thread synchronization. Therefore, allocation may turn to become in a bottleneck affecting performance. TLABs dedicate a buffer for each thread, which can be accessed safely improving allocation throughput. Space wastage is minimized and synchronization is necessary very infrequently, only when buffers need more space.

2.2 Available collectors

Finding the right garbage collection algorithm is not trivial and we have already described the performance metrics to chose the right collector on Section 1.3. The Java HotSpot Virtual Machine implementation provides multiple garbage collectors, each designed to satisfy different requirements. This is an important part of meeting the demands of both large and small

applications. In the configuration is also possible to modify some metrics, e.g., maximum garbage collection pause time and heap size, to reach different performance goals. However, the tuning is not trivial and can affect seriously the application performances.

HotSpot includes multiple collectors, each with different performance characteristics. Collectors are designed to satisfy various system requirements and to manage large and small applications. Users can select the garbage collector that meets their requirements. Otherwise the JVM runs the collector based on the class of the machine (client or server), considering also the system specifications. Starting with Java SE version 5.0 a machine with 2 or more CPU-cores and 2 or more gigabytes of RAM is considered a server-class machine, otherwise a client-class machine. Distinction regards also operating systems, e.g., in machines running Microsoft Windows 32-bit the default runtime compiler is the Client version, independently of system characteristics. Let see a summary of the available collectors, mainly based on [17].

2.2.1 Serial Collector

Description

With Serial Collector all the operations, i.e., minor and major collections, are done serially using a single thread with the adoption of the *stop-the-world* policy. For the young generation collections this collector uses the copying collection algorithm, while for the old generation collection it uses the mark-compact collection. The compaction is carried out by *sliding* the live objects towards the beginning of the old generation (this is also performed for the permanent generation), leaving contiguous free space at the opposite end.

Performance considerations

The Serial Collector is set as default in client-class machines and is suitable for most applications that do not have low pause requirements. It is particularly efficiently on single-core CPUs and with non-trivial applications with very limited heap size.

2.2.2 Parallel Compacting Collector

Description

Parallel Compacting Collector, as the name said, performs the operation in parallel using as many threads as the number of CPUs available. Since the collections are done in parallel exploiting many CPUs, it reduces garbage collection overheads improving throughput. For this reason it is also called the *throughput collector*. Policy adopted is again *stop-the world* and it uses a copying collection algorithm to performs minor collection. If Parallel Collector is specified, the early version of the algorithm already selectable, the old generation collections are performed using mark-compact like for the Serial Collector. In the last version, the old and permanent generations are collected using a parallel version of the mark-compact collector with sliding compaction.

Performance considerations

Usually, when more than two processors are available., this collector performs significantly better than the serial collector. Conversely, with a single processor the performances are not good as for the serial collector because of the overhead required for threads synchronization. Parallel Compacting Collector is particularly suitable for applications with pause time constraints because it improves the performance of major collections. With tuning is possible to reach desired performance goals according to the following priority order: maximum pause time, throughput and minimum footprint.

2.2.3 Concurrent Mark-Sweep Collector

Description

Concurrent Mark-Sweep (CMS) Collector is the right choice when pause time constraints are more important than overall throughput. Usually, young generation collections do not require long pauses and to perform this operations CMS collector works as the Parallel Compacting Collector. Instead, for the major collections it carry out the operations concurrently to the mutator using a mark-sweep collector. Since the collector runs concurrently with the mutator, objects references may be updated during the marking phase. To handle this situation, the mutator stops with a short pause allowing the collector to perform an *initial mark phase* to identify the reachable objects. To be sure that all live objects are been marked after

the *concurrent marking phase*, the mutator stops again for a second pause, called the *remarking phase*, which revisits modified objects. To increase efficiency of remarking phase the operations are carried out in parallel. Finally, a *concurrent sweep phase* reclaims space of identified garbage without compacting generations in order to save time.

Performance considerations

Remarking phase require additional work which increases the overhead. Furthermore, non-compacting spaces on the long-run lead to fragmentation. This also increases minor collections pauses because finding space to store the tenured objects requires time. Since the mutator can allocate memory during collector runs, CMS requires a large heap size than other collectors. Normally it does not provide any benefit on single-core machines as at least one core performs garbage collection concurrently with the application. However, there is a special mode for machines with one or two processors, called *incremental mode*. The incremental mode divides the work done concurrently by the collector into small chunks which are scheduled between minor collections. CMS, unlike others collectors, prevents that old generation becomes full, trying to start the operations early. In conclusion, it is a good choice when concurrency is needed and it reaches his performance goals with costs in terms of throughput and extra heap size requirements.

2.2.4 G1 Garbage Collector

The Garbage-First or G1 garbage collector [6] is available in Java 7 and is designed to be the long term replacement for the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector that has quite a different layout from the other garbage collectors described previously. It is a server-style collector suitable for multi-processor machines with large memories. It can operate concurrently like the CMS collector adding some features to improve performances. Moreover, it can compact the free space without sacrificing throughput and it does not require a larger heap. Comparing to CMS the mainly differences are:

- it is a compacting collector, thus it resolves the fragmentation problem;
- it offers more predictable pauses and allows users to tuning the desired pause targets.

Chapter 3

Theoretical Background

This chapter recalls the concepts and notions used to model and analyse and model the performances of garbage collection policies. It starts providing the definition of Markov process, then it focus on Markov chains. Finally, it introduces queueing theory with particular attention to the models implemented in this work.

3.1 Markov process

A stochastic process (also called random process) is a sequence of random variables used to represent the evolution of a system over time [18]. More formally, a stochastic process is denoted by $\{X(t), t \in T\}$, where $X(t)$ is a random variable and the parameter t runs over an index set T.

Stochastic processes are distinguished by their state space S, the index set T and by the dependence relations between random variables. The state space is the set of all possible values for the random variables $X(t)$. If the index set T is countable, $X[n]$ is a discrete stochastic process and n is the discrete time or a time slot in a computer system. If T is a continuum, $X(t)$ is a continuous-time stochastic process. For example, the number of customer arrivals in our system during a certain time interval is a continuous-time stochastic process.

The statistical characteristics may be dependent on the time t at which the system is started. A process that is invariant, for an arbitrary shift of the time origin, is said to be *stationary*. More formally, for any constant α,

$$\begin{aligned} &Pr(\{X(t_1) \leq x_1, X(t_2) \leq x_2, \ldots, X(t_n) \leq x_n\}) \\ &= Pr(\{X(t_1 + \alpha) \leq x_1, X(t_2 + \alpha) \leq x_2, \ldots, X(t_n + \alpha) \leq x_n\}) \end{aligned} \tag{3.1}$$

for all n and all t_i and x_i with $i = 1, 2, \ldots, n$. Otherwise the process is said *nonstationary*. If the transitions depend upon the amount of time has elapsed, the stochastic process is said to be *nonhomogeneous*. When the transitions are independent of the elapsed time, it is said to be *homogeneous*.

A stochastic process is a Markov process if the conditional probability distribution function satisfies the Markov or memoryless property. This means that the future state of the process only depends on the current state of the process and not on its past history. Discrete-time Markov chains (DTMCs) are defined as discrete-time Markov processes and continuous-time Markov chains (CTMCs) are defined as continuous-time Markov processes with a discrete state space. Formally, a stochastic process $\{X(t),\, t \in T\}$ is a continuous-time Markov process if, for all $t_0 < t_1 < \cdots < t_{n+1}$ of the index set T and for any set $\{x_0, x_1, \ldots, x_{n+1}\}$ of the state space, it holds that

$$\begin{aligned} &Pr(\{X(t_{n+1}) = x_{n+1} \mid X(t_n) = X_n, \ldots, X(t_0) = x_0\}) \\ &\qquad = Pr(\{X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n\}). \end{aligned} \tag{3.2}$$

Similarly, discrete-time Markov process $\{X_k,\, k \in T\}$ is a stochastic process whose state space is a finite or countably infinite set with index set $T = \{0, 1, 2, \ldots\}$ complying

$$\begin{aligned} &Pr(\{X_{n+1} = x_{n+1} \mid X_n = x_n, \ldots, X_0 = x_0\}) \\ &\qquad = Pr(\{X_{n+1} = x_{n+1} \mid X_n = x_n\}). \end{aligned} \tag{3.3}$$

3.2 Discrete-time Markov chains

A discrete-time Markov chain, as we said previously, is a stochastic process that satisfies the *Markov property*, has a discrete state space and evolves according to a discrete time. For simplicity on the notation, we can write conditional probabilities $Pr(\{X_{n+1} = x_{n+1} \mid X_n = x_n\})$ as $Pr(\{X_{n+1} = j \mid X_n = i\})$ that are called the transition probabilities of the Markov chain. These probabilities give the conditional probability of making a transition from state $x_n = i$ to state $x_{n+1} = j$ when the time parameter increases from n to $n + 1$. Transition probabilities are denoted by

$$p_{ij}(n) = Pr(\{X_{n+1} = j \mid X_n = i\}) \tag{3.4}$$

and they form the *transition probability matrix* $P^{(n)}$:

$$P(n) = \begin{pmatrix} p_{00}(n) & p_{01}(n) & \dots & p_{0j}(n) & \dots \\ p_{10}(n) & p_{11}(n) & \dots & p_{1j}(n) & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{i0}(n) & p_{i1}(n) & \dots & p_{ij}(n) & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \tag{3.5}$$

If $P(n)$ satisfies the following two properties:

$$0 \leq p_{ij}(n) \leq 1,$$

$$\forall i \in S \sum_{j \in S} p_{ij}(n) = 1,$$

the matrix is called *stochastic matrix* or *Markov matrix*.

A Markov chain is said to be *time-homogeneous*, usually written simply as homogeneous, if for all states i and j

$$Pr(\{X_{n+1} = j \mid X_n = i\}) = Pr(\{X_{n+m+1} = j \mid X_{n+m} = i\})$$

for $n = 0, 1, \dots$ and $m \geq 0$. Since transitions no longer depend on n, we can replace $p_{ij}(n)$ with p_{ij}

$$p_{ij} = Pr(\{X_{n+1} = j \mid X_n = i\}) \quad \text{for all } n = 0, 1, 2, \dots$$

It follows that the stochastic matrix P(n) can be written simply as P.

3.2.1 Irreducible Markov chains

A Markov chain can also be described by a directed graph, where p_{ij}, provided $p_{ij} > 0$, is represented by an edge from state i to j. If every state is reachable from every other state, the chain is said to be *irreducible* and this simplify significantly Markov theory. An irreducible Markov chain is equivalent to have the associated graph strongly connected. As we know from graph theory, a directed graph is strongly connected if there is a path from node i to node j for any pair of distinct nodes (i, j). The most computationally efficient method to determine irreducibility consists of applying all-pairs-shortest-path algorithms, like Floyd-Warshall or Johnson, on the corresponding Markov graph.

3.2.2 Communication classes

We said that a state j is reachable from another state i, denoted by $i \to j$, if there exist a $n \geq 0$ such that $p_{ij}(n) > 0$. This means that starting from state i, there is a non null probability that the chain will be in state j at time n. If the two states are reachable from one to each other, they are said to *communicate*, denoted by $i \longleftrightarrow j$. The communication is an equivalence relation and we recall here the conditions:

- *Reflexivity.* $i \longleftrightarrow i$.
- *Symmetry.* If $i \longleftrightarrow j$, then $j \longleftrightarrow i$.
- *Transitivity.* If $i \longleftrightarrow k$, and $k \longleftrightarrow j$, then $i \longleftrightarrow j$.

The state space can be partitioned into disjoint subsets (equivalence classes) with the property that all states within communicate. Any Markov chain can be partitioned in this way.

Proposition 3.1 *For each Markov chain, there exists a unique decomposition of the state space S into a sequence of disjoint subsets $C1, C2, \ldots$, $S = \bigcup_{i=1}^{\infty} C_i$, in which each subset in S has the property that all states within it communicate and is called a communication class of the Markov chain.*

If the partition results into only one communication class the Markov chain is irreducible and all states communicate.

3.2.3 Classification of states

We proceed characterizing the states and recall some important properties that define the states of discrete-time Markov chains. States could be distinguished in *recurrent states* and *transient states*. Let r_{ii} denote the *return time* to state i given $X_0 = i$:

$$r_{ii} = min\{n \geq 1 : X_n = i \mid X_0 = i\},\ r_{ii} \stackrel{def}{=} \infty,\ if X_n \neq i,\ n \geq 1.$$

It represents the number of steps until the chain returns to state i given that it started from state i. A return occurs if and only if $r_{ii} < \infty$, otherwise the chain never returns to the state i. We set $f_i \stackrel{def}{=} Pr(\{r_{ii} < \infty\})$ the probability of a chain to return to state i given that the chain started in state i. If a state i has $f_i = 1$, it is visited with the same probability an

infinite number of time. For the Markov property the chain will return to state i an infinite number of times, independent by the past. In this case we call the state *recurrent*. If $f_i < 1$ there is a nonzero probability that the Markov chain will never return to this state. In this case the state i will only be visited a finite random number of times and it is call *transient*.

Proposition 3.2 *For any communication class C, all states in C are either recurrent or all states in C are transient. Thus, if i and j communicate and i is recurrent, then also j is recurrent. Equivalently if i and j communicate and i is transient, then also j is transient. In particular, for an irreducible Markov chain all states are recurrent or all states are transient.*

When state j is recurrent we define the *mean recurrence time* of j as $M_{jj} = \sum_{n=1}^{\infty} n f_{jj}^{(n)}$. M_{jj} is the average number of steps taken to return to state j for the first time after leaving it. If $M_{jj} = \infty$, we say that state j is a *null recurrent* state. Otherwise if M_{jj} is finite the state j is called a *positive recurrent* state.

Proposition 3.3 *In a finite irreducible Markov chain no state is null recurrent. At least one state must be positive recurrent, i.e., not all states can be transient.*

An interesting property of the states in a Markov chain is *periodicity*. A state j is *periodic* with period d if on leaving the state a return is possible in a number of transitions that is a multiple of $d > 1$. Thus, given a state j in a Markov chain with $p_{ij}(k) > 0$ for some $k \geq 1$, the period d_j is the gcd (greatest common divisor) of those k for which $p_{ij}(k) > 0$. A state whose period is $d = 1$ is said to be *aperiodic*. A state that is positive recurrent and aperiodic is said to be *ergodic*. If all the states of a Markov chain are ergodic, then also the Markov chain is said to be ergodic. If two states communicate then their period is the same and all the states in an irreducible Markov chain have common period d. An irreducible Markov chain is said *aperiodic* if $d = 1$, and a sufficient condition is that $p_{ii} > 0$ for some state i. Most Markov chains of practical interest are aperiodic and this is also our case as we shall see on the next chapters.

3.2.4 The steady-state distribution

Of particular interest is the probability that a homogeneous DTMC is in a given state at a particular time step. We denote by $\pi_i(n)$ the probability

that a Markov chain is in state i at step n, i.e.,

$$\pi_i(n) = Pr(\{X_n = i\})$$

and in vector notation the resulting row vector is

$$\pi(n) = (\pi_1(n), \pi_2(n), \ldots, \pi_i(n), \ldots).$$

The state probabilities at any time step n may be obtained from a knowledge of the initial probability distribution ($\pi(0)$) and the matrix of transition probabilities P. From the law of total probability we have

$$\begin{aligned} \pi_i(n) &= \sum_{k \in S} Pr(\{X_n = i \mid X_0 = k\})\pi_k(0) \\ &= \sum_{k \in S} p_{ki}^{(n)} \pi_k(0) \end{aligned} \tag{3.6}$$

which in matrix notation becomes

$$\pi(n) = \pi(0)P^{(n)} = \pi(0)P^n. \tag{3.7}$$

The probability distribution π is called the *transient distribution*, since it gives the probability of being in the various states of the Markov chain at a particular instant in time.

Definition 3.1 (Limiting distribution) *Let P be the transition probability matrix of a homogeneous discrete-time Markov chain and let $\pi(0)$ be an initial probability distribution. If the limit*

$$\lim_{n \to \infty} P^{(n)} = \lim_{n \to \infty} P^n$$

exists, then the probability distribution

$$\pi = \lim_{n \to \infty} \pi(n) = \pi(0) \lim_{n \to \infty} P^n$$

exists and is called the limiting distribution of the Markov chain.

When the Markov chain is ergodic or is finite, irreducible and aperiodic then the limiting distribution always exists and is unique.

Definition 3.2 (Steady-state distribution) *Let π a limiting distribution. π is a steady-state distribution if it converges, independently of the initial*

starting distribution $\pi(0)$, to a vector whose components are strictly positive and sum to 1. If a steady-state distribution exists, it is unique.

A component i of a steady-state distribution represents the probability to find, randomly, the Markov chain in state i after a long period of time. Steady-state distributions are also called *equilibrium distributions* and *long-run distributions*. If the Markov chain contains a finite number of states and is irreducible, all the states are positive recurrent and exists a unique stationary distribution. If the chain is also aperiodic, thanks to the aperiodicity property, the distribution is also the unique steady-state distribution.

In general there are two ways of computing the stationary distribution π: via the limiting process or via solving the set of linear equations.

3.3 Continuous-time Markov chains

A continuous-time Markov chain (CTMC), as we said at beginning, is a stochastic process with continuous time and discrete state space that satisfies the Markov property. This mean that, considering the formal definition (3.2), the future behaviour of the model depends only on the current state and not on the historical behaviour. In a CTMC a change of state may occur at any time and the amount of time already spent in the current state is also irrelevant as the previously visited states in the chain.

More formally we say that the stochastic process $\{X(t), t \geq 0\}$ is a CTMC if for states i, j, k and for all time instants s, t, u with $t \geq 0$, $s \geq 0$ and $0 \leq u \leq s$, we have

$$\begin{aligned} &Pr(\{X(s+t) = k \mid X(s) = j, X(u) = i\}) \\ &\quad = Pr(\{X(s+t) = k \mid X(s) = j\}). \end{aligned} \tag{3.8}$$

In case the chain is nonhomogeneous, we have

$$p_{ij}(s,t) = Pr(\{X(t) = j \mid X(s) = i\}),$$

where X(t) denotes the state of the Markov chain at time $t \geq s$. Otherwise, if the chain is homogeneous the transitions probabilities depend on the difference $\tau = t - s$ and we have

$$p_{ij}(\tau) = Pr(\{X(s+\tau) = j \mid X(s) = i\}) \quad \forall s \geq 0.$$

This is the probability that the chain is in state j after an interval of length

τ starting from the state i. Since the chain must be at any time in one of states j, with $j \in S$, we have for all values of τ

$$\sum_{j \in S} p_{ij}(\tau) = 1.$$

For each $t \geq 0$ there is a transition matrix

$$P(t) = (p_{ij}(t)),$$

and $P(0) = I$, the identity matrix.

Differently from the discrete-time case, in a CTMC there is a continuum of possible times t. Probability $p_{ij}(t)$ could be studied by use of calculus and differential equations and this make the analysis more difficult than DTMCs.

3.3.1 Infinitesimal generator

For Continuous-time Markov chain the interactions between states are not based on transition probabilities like for discrete-time Markov chains, but depend on the rate $q_{ij}(t)$ at which a transition from state i to state j occurs. While for DTMCs we have the transition probability matrix $P^{(n)}$ at step n, for CTMCs we have the *transition-rate matrix* $Q(t)$ at time t. The matrix Q is called the *infinitesimal generator* of the chain and contains all the rate information for the chain. For $i \neq j$ the elements q_{ij} are non-negative and describe the process transition from state i to state j.

Before explain the infinitesimal generator, also called the *transition-rate matrix*, we have to discuss the concepts of probability and rate. The probability that a transition occurs depend not only on the source state i, but also on the length of the interval of observation. Thus, we consider a period of observation $\tau = \Delta t$ and $p_{ij}(t, t + \Delta t)$ is the probability that a transition occurs in the interval $[t, t + \Delta t]$ from state i to state j. If this interval is very small also the observed probability is small, so for $i \neq j$ if $\Delta t \to 0$ then $p_{ij}(t, t + \Delta t) \to 0$. We also have for $\Delta \to 0$ that $p_{ii}(t, t + \Delta t) \to 1$, thanks to the conservation of probability. Instead, if Δt is very large also the probability increases. Interval of observation are chosen sufficiently small that the probability of observing multiple events in any observation period is of order $o(\Delta t)$.

A rate of transition $q_{ij}(t)$ denotes the number of transitions per unit time that occurs from state i to state j at time t and, despite probability,

does not depend on the length of the interval Δt. More formally, we have

$$q_{ij}(t) = \lim_{\Delta t \to 0} \left\{ \frac{p_{ij}(t, t+\Delta t)}{\Delta t} \right\} \quad \text{for } i \neq j. \tag{3.9}$$

And for probabilities it follows that

$$p_{ij}(t, t+\Delta t) = q_{ij}(t)\Delta t + o(\Delta t) \quad \text{for } i \neq j. \tag{3.10}$$

Now for the conservation of probability and last results we have

$$\begin{aligned} 1 - p_{ii}(t, t+\Delta t) &= \sum_{j \neq i} p_{ij}(t, t+\Delta t) \\ &= \sum_{j \neq i} q_{ij}(t)\Delta t + o(\Delta t). \end{aligned} \tag{3.11}$$

Taking the limit and dividing by Δt we find

$$\lim_{\Delta t \to 0} \left\{ \frac{\sum_{j \neq i} q_{ij}(t)\Delta t + o(\Delta t)}{\Delta t} \right\} = \sum_{j \neq i} q_{ij}(t).$$

In the continuous-time Markov chains the transition rate corresponding to the system remaining in place is defined by the equation

$$q_{ii}(t) = -\sum_{j \neq i} q_{ij}(t). \tag{3.12}$$

This quantity is what we already called the transition *rate* and it is defined as a derivative. If state i is an absorbing state, $q_{ii}(t) = 0$. Substituting Equation (3.12) into Equation (3.11) provides the result we found in Equation (3.10). Now we rewrite all results together:

$$\begin{aligned} p_{ij}(t, t+\Delta t) &= q_{ij}(t)\Delta t + o(\Delta t) \quad \text{for } i \neq j, \\ p_{ii}(t, t+\Delta t) &= 1 + q_{ii}(t)\Delta t + o(\Delta t). \end{aligned}$$

When the CTMC is homogeneous the rate become

$$\begin{aligned} q_{ij} &= \lim_{\Delta t \to 0} \left(\frac{p_{ij}(\Delta t)}{\Delta t} \right) \quad \text{if } i \neq j \\ q_{ij} &= \lim_{\Delta t \to 0} \left(\frac{p_{ij}(\Delta t) - 1}{\Delta t} \right). \end{aligned} \tag{3.13}$$

Finally, we have the infinitesimal generator for the CTMC, that is the

matrix $Q(t)$ whose ij^{th} element is the rate $q_{ij}(t)$:

$$Q(t) = \lim_{\Delta t \to 0} \left\{ \frac{P(t, t+\Delta t) - I}{\Delta t} \right\}.$$

$P(t, t+\Delta t)$ is the transition probability matrix and I is the identity matrix. Considering Equation (3.12) we note that the sum of rows of $Q(t)$ must be equal to zero. Furthermore when the CTMC is homogeneous the infinitesimal generator is written simply as Q because the transitions rates q_{ij} are independent of time.

3.3.2 The steady-state distribution

We consider with $\pi(t)$ the probability that the system is in state i at time t, i.e., $\pi(t) = Pr(\{X(t) = i\})$. Let Δt an interval of time, we know for the Markov property that the probability to observe the system in state i at time $t + \delta t$ must be equal to the probability to observe it in state i at time t. Furthermore this probability does not change state in the period $[t, \Delta t)$ and we have to consider also the probability that the system is in same state $k \neq i$ at time t and moves to state i in the interval Δt:

$$\pi(t + \Delta t) = \pi(t) \left(1 - \sum_{\text{all } j \neq i} q_{ij}(t) \Delta t \right) + \left(\sum_{\text{all } k \neq i} q_{ki}(t) \pi_k(t) \right) \Delta t + o(\Delta t).$$

After a simplification and taking the limit

$$\lim_{\Delta t \to 0} \left(\frac{\pi_i(t + \Delta t) - \pi_i(t)}{\Delta t} \right) = \lim_{\Delta t \to 0} \left(\sum_{\text{all } k} q_{ki}(t) \pi_k(t) + \frac{o(\Delta t)}{\Delta t} \right),$$

we have

$$\frac{d\pi_i(t)}{dt} = \sum_{\text{all } k} q_{ki}(t) \pi_k(t).$$

When the chain is homogeneous and in matrix notation it becomes

$$\frac{d\pi_i(t)}{dt} = \pi(t) Q.$$

This leads to the solution

$$\pi(t) = \pi(0) e^{Qt} = \pi(0) \left(I + \sum_{n=1}^{\infty} \frac{Q^n t^n}{n!} \right). \tag{3.14}$$

This result can be obtained directly from the solution to the *Kolmogorov forward equations*

$$\frac{dP(t)}{dt} = P(t)Q \tag{3.15}$$

since, by definition

$$\pi(t) = \pi(0)P(t) = \pi(0)e^{Qt}.$$

Summarizing, in a homogeneous CTMC the i^{th} element of the distribution vector $\pi(t)$ is the probability that the chain is in state i at time t and the state probabilities are governed by the system of differential equations

$$\frac{d\pi(t)}{dt} = \pi(t)Q$$

Definition 3.3 (Steady-state distribution) *A steady-state distribution exists and is unique when exists the limiting distribution, when all its components are strictly positive and when it is independent of the initial probability vector* $\pi(0)$.

This distribution always exists and is identical to the stationary distribution of the chain in case the chain is a finite, irreducible, CTMC. The steady-state distribution may be obtained by solving the system of linear equations

$$\pi Q = 0, \tag{3.16}$$

subject to the condition that $\|\pi\|_1 = 1$.

3.4 Computation of the stationary distribution of a Markov chain

This section discusses the computation of the stationary distribution of Markov chains. In particular we consider finite, irreducible Markov chains that have a unique stationary distribution π and, as we previously said, when a chain is also aperiodic π is the steady-state distribution.

In general we can distinguish two possible cases. In the first case the number of states of the Markov chain is finite. In this situation the system

of equations, for DTMC

$$\pi P = \pi$$
$$\sum_{i=0}^{n} \pi_i = 1$$

and for CTMC

$$\pi Q = 0$$
$$\sum_{i=0}^{n} \pi_i = 1,$$

can be solved explicitly with direct methods based on Gaussian elimination, point and block iterative methods such as point and block Gauss-Seidel and decompositional methods that are especially well suited to Markov chains that are almost reducible. In the second case when the Markov chain has an infinite state space the solution may be computed using the matrix geometric and matrix analytic methods. In our models we have stationary homogeneous Markov chains with an infinite number of states and we focus only on matrix geometric method.

3.4.1 Quasi-Birth and Death process

Before introduce the matrix geometric approach we have to define a typical Markov chain structure that can be efficiently solved using this method. The model is called *Quasi-Birth and Death* (QBD) process and the name comes from his common application to study demographic trends. As in demography the transitions are *births* and *deaths*. When a customer arrives at the system this event is identified with a *birth*, when it leaves the system we say that it departs and the event is referred to as a *death*. The peculiarity of this class of Markov chains is that the interactions between states occur only between nearest neighbors. QBD processes are also known as *skip-free processes* since all intermediate states must be visited.

Consider a Markov chain with state space $S = \bigcup_{i \geq 0} \{(i, j) : 1 \leq j \leq m\}$. The first component i is called the *level* of the chain, the second component j is called the *phase* of the chain and m is an integer that can be finite or infinite. The Markov chain is a QDB process if the one step transition from a state is restricted to the same level or to the two adjacent levels. More formally let $(i, j) \in S$ with $i \geq 1$, then the Markov chain is a QDB process

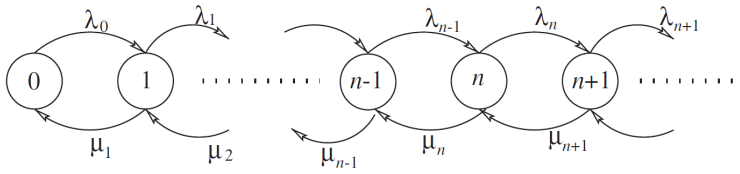

Figure 3.1: Quasi-birth and death process state diagram.

if

$$(i,j) \longleftrightarrow (i,j'); \qquad (i-1,j') \longleftrightarrow (i,j) \longleftrightarrow (i+1,j'').$$

The QDB process is an extension of the standard birth and death process whose state space consists only of the level i as we can see in Figure 3.1. If the transition rates are level independent, the resulting QBD process is called *homogeneous* or *level-independent* QBD (LIQBD) process; else it is called *inhomogeneous* or *level-dependent* QBD (LDQBD) process [11]. The process is specified by birth rates, denoted with λ_i, and death rates, denoted with μ_i, with $i = 0, 1, \ldots, \infty$.

This model is defined by the infinitesimal generator matrix

$$Q = \begin{pmatrix} -\lambda_0 & \lambda_0 & 0 & 0 & 0 & 0 & \ldots \\ \mu_1 & -(\lambda_1+\mu_1) & \lambda_1 & 0 & 0 & 0 & \ldots \\ 0 & \mu_2 & -(\lambda_2+\mu_2) & \lambda_2 & 0 & 0 & \ldots \\ 0 & 0 & \mu_3 & -(\lambda_3+\mu_3) & \lambda_3 & 0 & \ldots \\ & & \ddots & \ddots & \ddots & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix},$$

a tridiagonal matrix that is irreducible if all $\lambda_i > 0$ and $\mu_i > 0$.

3.4.2 Matrix geometric/analytic methods

Matrix geometric and matrix analytic methods were introduced by Marcel F. Neuts [14, 15] and studied by several researches since late seventies. This methods are suitable for the Markov chains whose transition matrices have a particular block structure and provide tools to construct and analyse a wide class of stochastic models, queueing systems in particularly, using a matrix formalism to develop algorithmically tractable solutions.

In the simplest case, matrices are infinite block tridiagonal matrices in which the three diagonal blocks repeat after some initial period. We report

below the typical transition rate matrix

$$Q = \begin{pmatrix} B_{00} & B_{01} & 0 & 0 & 0 & 0 & \dots \\ B_{10} & A_1 & A_2 & 0 & 0 & 0 & \dots \\ 0 & A_0 & A_1 & A_2 & 0 & 0 & \dots \\ 0 & 0 & A_0 & A_1 & A_2 & 0 & \dots \\ & & & \ddots & \ddots & \ddots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}, \tag{3.17}$$

where A_0, A_1, A_2 are square matrices with the same dimension; B_{00} is also a square matrix, but the dimension could be different than A_1; B_{01} and B_{10} are matrices whose sizes are defined in accordance with B_{00} and A_1 respectively. This particular block structure is exactly the same that we have define for the QBD processes. In particular, with a block tridiagonal structure the states are grouped into *levels* according to their i value and the transitions are permitted only between neighbors, i.e., the states of the same level (diagonal blocks) can interact only with states in the adjacent levels (superdiagonal or subdiagonal blocks). After considered boundary conditions (given by the initial B submatrices) the transition rates are identical from level to level.

To introduce the method to compute the solution we firstly consider a special type of QBD process: the $M/M/1$ queueing system, which has the submatrices of Q are reduced to a single element. We consider the probability to move from a state k to a state $k+1$ as $\frac{\lambda}{(\lambda+\mu)}$ and the probability to move from a state k to a state $k-1$ as $\frac{\mu}{(\lambda+\mu)}$. This is also a typical *random walk* problem as defined in [18]. Now from equation $\pi Q = 0$ we obtain the following general equation

$$\lambda\pi_{i-1} - (\lambda + \mu)\pi_i + \mu\pi_{i+1} = 0 \tag{3.18}$$

and we proceed by induction to show that

$$\pi_{i+1} = \left(\frac{\lambda}{\mu}\right)\pi_i \quad \text{for } i \geq 1.$$

The basic case is

$$\pi_1 = \left(\frac{\lambda}{\mu}\right)\pi_0$$

and from the inductive hypothesis

$$\pi_i = \left(\frac{\lambda}{\mu}\right)\pi_{i-1},$$

we have

$$\pi_{i+1} = \left(\frac{\lambda+\mu}{\mu}\right)\pi_i - \left(\frac{\lambda}{\mu}\right)\pi_{i-1} = \left(\frac{\lambda}{\mu}\right)\pi_i$$

which gives the desired result. It follows that

$$\pi_i = \rho^i \pi_0,$$

where $\rho = \frac{\lambda}{\mu}$.

The problem now, once π_0 is known, is to find π_i recursively. When Q is a QBD process ρ becomes a square matrix R of order m. Let Q the infinitesimal generator as defined in Matrix (3.17) and π, the stationary distribution vector obtained from $\pi Q = 0$, partitioned conformally with Q

$$\pi = (\pi_0, \pi_1, \pi_2, \ldots).$$

Furthermore, the components π_i become subvectors of length m, i.e.,

$$\pi_i = (\pi(i,1), \pi(i,2), \ldots, \pi(i,m)) \quad i \geq 0,$$

where $\pi(i,j)$ is the probability of finding the system in state (i,j) at steady state. From $\pi Q = 0$ we have the following equations:

$$\begin{aligned}
\pi_0 B_{00} + \pi_1 B_{10} &= 0, \\
\pi_0 B_{01} + \pi_1 A_1 + \pi_2 A_0 &= 0, \\
\pi_1 A_2 + \pi_2 A_1 + \pi_3 A_0 &= 0, \\
&\vdots \\
\pi_{i-1} A_2 + \pi_1 A_1 + \pi_{i+1} A_0 &= 0, \quad i \geq 2
\end{aligned}$$

As we can see there exists a constant matrix R and vector π may be computed with the following scheme

$$\pi_i = \pi_{i-1} R \quad i \geq 2 \tag{3.19}$$

where π_0, π_1 and constant matrix R are defined in [14, 15]. Once the rate matrix R is obtained, the station distribution can be computed. The iter-

atively methods proposed by Neuts to compute R may require long time to converge, but there are other possible choices like the *logarithmic reduction algorithm* developed by Latouche and Ramaswami [12], or the cyclic reduction algorithm [2].

To derive π_0 and π_1 we rewrite in matrix form the first two equation of $\pi Q = 0$:

$$(\pi_0, \pi_1) = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & A_1 + RA_0 \end{pmatrix} = (0, 0). \tag{3.20}$$

After the system has been solved the distribution vector obtained needs to be normalized to respect the condition $\sum_i \pi_i = 1$.

Following steps summarize the procedure to solve a QBD process by the matrix geometric method:

1. Ensure that the matrix has the requisite block structure.
2. Ensure that the chain is ergodic with the following condition

   $$\pi_A A_2 e < \pi_A A_0 e,$$

   where π_A is the stationary distribution of the infinitesimal generator $A = A_0 + A_1 + A_2$.
3. Compute the rate matrix R.
4. Solve the system of Equations (3.20) for π_0 and π_1.
5. Normalize π_0 and π_1.
6. Use Equation (3.19) to compute the remaining components of the stationary distribution vector π.

To find the stationary distribution for the models presented in this work we have included *SMC-Solver* in our matlab code for solving structured Markov chains. This very useful package contains the most advanced algorithms for solving QBD, $M/G/1$ and $G/M/1$ problems [3].

In conclusion this chapter recall all the theoretical concepts that are useful to understand the following chapters. Of particular interest is the last section in which we have introduced QDB processes, one of the most fundamental models that extends the basic $M/M/1$ queueing systems. We have discussed also the matrix geometric method to find the numerical solution of the corresponding Markov chain.

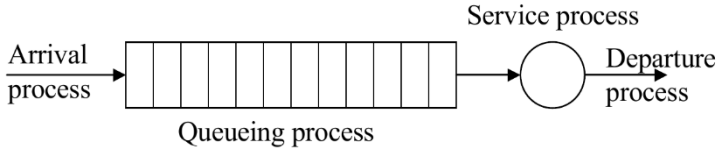

Figure 3.2: The main processes in a general queueing system.

3.5 Queueing theory concepts

In this section we recall some concepts from queueing theory that will be useful in later chapters.

Following the Kleinrock's definition (1975), queueing systems are those systems in which arrivals place demands upon a finite-capacity resource can be broadly termed a *queueing system*. Queueing theory [18, 16] describes the basic phenomena in a queueing system and Figure 3.2 illustrates its mainly components: the arriving items (customers), a buffer where items wait for service, a service center (server) and departures from the system. This components has a stochastic nature. Let see their characteristics.

3.5.1 The arrival process

The arrival process can be described in two ways:

- the number of arrivals per unit time, called the *arrival rate*;
- the time between successive arrivals, called the *interarrival time*.

The mean arrival rate is usually denoted by the variable λ and $1/\lambda$ denotes the mean time between arrivals. If the input process is a stochastic process there is an associated probability distribution, denoted by $A(t)$. Let τ the time between arrivals, then the probability distribution between interarrival time of customers is

$$A(t) = Pr(\{\tau \leq t\}).$$

and

$$\frac{1}{\lambda} = \int_0^\infty t \mathrm{d}A(t),$$

where $\mathrm{d}A(t)$ the probability that the interarrival time is between t and $t+\mathrm{d}t$. We assume that the interarrival times are *i.i.d.* (independent and identically distributed). Otherwise every different class of customers has its own probability distribution to describe its arrival process.

3.5.2 The service process

As for the arrival process, the service process may be described in two ways:

- the number of customers served per unit time, called the *service rate*;
- the time required to serve a customer.

Usually the variable μ denotes the mean service rate, and hence the mean service time is denoted by $1/\mu$. The probability distribution of the demand placed on the system is denoted by $B(x)$, defined as

$$B(x) = Pr(\{servicetime \leq x\})$$

and

$$\frac{1}{\mu} = \int_0^\infty x \mathrm{d}B(x),$$

where $\mathrm{d}B(x)$ is the probability that the service time is between x and $x+\mathrm{d}x$. It is important to remark that the service time is equal to the length of time spent in service and does not include the waiting time. Furthermore, depending on how many servers are available, the service may be *batch*, in which several customers can be served simultaneously, or *single*.

3.5.3 Queuing discipline

There are many ways to select customers from the queue and take them into service. This ways are usually called *scheduling rules* and if not specified they assume that the time spent to select customers is zero. This means that the customers selection and customers departure are two simultaneously events. We can distinguish two kinds of scheduling rules: *preemptive policies*, that can interrupt the service of the customer in service, and *nonpreemptive policies*, that cannot perform this operation. Preemptive policies are useful in presence of different types of customers, having different service priorities.

Within the context of preemptive and nonpreemptive service, several scheduling rules exists. The most common and simplest is *First-In-First-Out* (FIFO), a nonpreemptive policy that serves the customers in their arrival order. Another example is *Processor Sharing* (PS), a preemptive policy in which the processor is "shared" among processes.

3.5.4 Kendall's notation

A useful notation to characterize queueing systems was introduced by Kendall [16]. It is given by $A/B/C/X/Y/Z$, where

A indicates the interarrival time distribution;

B indicates the service time distribution;

C indicates the number of server;

X indicates the system capacity;

Y indicates the size of the customer population;

Z indicates the queueing discipline.

The first three parameters are always provided and there are different possible distributions for A and B. As an example, the $M/M/1$ queue means that the arrival process and service process are both Markovian (M) and there is a single server.

3.5.5 Measures of effectiveness

When we analyse a queueing system we are interested to obtain some useful values that are called measures of effectiveness, like:

- the number of customers in the system;
- the waiting time for a service, i.e., the average response time.

Let N be the random variable that describe the number of customers present at steady state. The average number of customers in the system is given by the following formula

$$E[N] = \sum_{c=1}^{\infty} c\pi_c, \tag{3.21}$$

where $\|\pi\|_1 = 1$.

We call *response time* the time that a customer spends in the system, from the instant of its arrival to the queue to the instant of its departure from the server [16]. We denote this time by R and its mean value by $E[R]$. Response time is composed by the *waiting time*, the time that a customer spends in the queue, and by service time.

The fraction of time in which the server is busy is called *system utilization* and we use ρ to denote it. This is a fraction of customer arrival rate and customer service rate:

$$\rho = \frac{\lambda}{\mu} \tag{3.22}$$

Formula (3.22) can depend on the number of customers admitted in the queue.

Finally, we report a very important result, found by Little, that relates the average waiting time and the average number of customers waiting for a service [18].

Theorem 3.1 (Little's Law) *The average number of packets (customers) in the system $E[N_S]$ (or in the queue $E[_Q]$) equals the average arrival rate λ times the average time spent in the system $E[T]$ (or in the queue E[w]),*

$$\begin{aligned} E[N_S] &= \lambda E[T], \\ E[N_Q] &= \lambda E[w]. \end{aligned} \tag{3.23}$$

Chapter 4

A QBD Model for GC

In this chapter we start presenting the queueing models studied to analyse and quantify the performances of a class of garbage collection algorithms. The first model we describe and summarize here was presented at the ASMTA 2011 Conference [1]. Balsamo *et al.* propose a queueing model to analyse a system with a garbage collector, in which customers arrive according to a Poisson process and the service time distribution depends on the amount of free memory. Furthermore, they propose a heuristic based on this model to derive an appropriate and effective garbage collector activation rate in order to minimise the average system response time.

4.1 Model description

The system is a client-server architecture in which the garbage collector is represented as a two-state system $\{\text{OFF}, \text{ON}\}$ that adopts a *stop-the- world* policy when it is active, i.e., on state ON. A stop the world policy implies a stop of the applications currently running to allow the garbage collector to work. Unfortunately, this situation has serious implications on a lot of applications. As an example we can take the undesired effects that the activation of a garbage collector with the stop-the-world policy causes on a video game: nobody would want to be interrupted during a session while he/she is playing with his/her favourite title. This is just an example of a possible scenario and the goal of a good policy is to make this events unlikely.

The goal of software engineers designing a new policy consists in deciding the frequency of the garbage collections and this choice is quite difficult to take. There is a clear trade-off in deciding the activation rate of a garbage

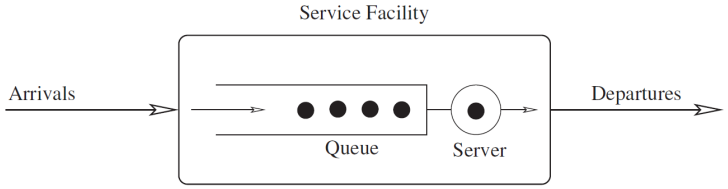

Figure 4.1: Single-server queue

collector: on one hand a high frequency causes frequent "interruptions" of the computations, but on the other hand the system performances tend to degrade when available memory is finishing due to the swapping, i.e, the virtual memory usage.

4.1.1 Queueing model

The queueing model is a single-server queue in which customers arrive according to a homogeneous Poisson process with exponentially distributed rate λ, as shown in Figure 4.1. A process activation corresponds to a customer arrival. Assuming an empty system with no other customers in the queue and that the garbage collector does not start, the service time is exponentially distributed and independent of the arrival time.

4.1.2 Memory assumptions

Memory considerations require particularly attention and are the main difference from our proposal on the next chapter. The memory is divided into B blocks and b blocks are occupied when a customer arrive at the system. This number b is sampled from a discrete random variable with a certain probability distribution and for simplicity in the article [1] the authors assume $b = 1$. The memory blocks may be allocated on all the memory, which includes the physical memory (RAM) and the virtual memory (disk space). Therefore, the performances of the system, and the service rate in particular, depend on the memory availability. When the system needs to allocate the virtual memory, the costs of the swapping activity increase significantly the service time. The model considers the service rate μ_i as a function of the occupied memory blocks i, with $0 \leq i \leq B$.

4.1.3 Garbage collector's role

The task of the garbage collector is to free the unused memory blocks and to keep low the number of occupied memory blocks in order to improve the service rate. We assume that the stop-the-world policy is adopted, which means that during an operation of garbage collection and memory optimization the customer service is suspended. The following activation conditions for the garbage collector have been identified in the paper:

- The system memory is full and a customer arrival event occurs.
- A timer set up with an exponentially distributed delay expires. The mean duration of these delays is α^{-1}, where α is the *activation rate*.

The activation rate may be assumed as a function of the occupied memory blocks, i.e., α_i. During the garbage collections there is a random exponentially time necessary to analyse each memory block. The rate of this operation is γ_i, with $1 \leq i \leq B$ and it depends on the quantity of the memory allocated. The rate of the garbage collection operation, called *deactivation rate*, is β_i with $1 \leq i \leq B$ and is the rate of exponentially distributed random variable. Finally, if the system is empty and no customer is being processed the garbage collector can immediately free all the memory. Parameters $\alpha_i, \beta_i, \gamma_i$ and μ_i are set using system statistics and we discuss it more in details in Chapter 6.

4.1.4 State of the system

The *state* of the system is defined by a triplet (c, i, g), where c is the number of customers in the system, i is the number of occupied memory blocks and g is the state of the garbage collector, e.g., the state $(5, 3, \text{ON})$ means that there are 5 customers in the system, 3 memory blocks allocated and that the garbage collector is running. Considering the initial case with the empty queue and garbage collection not active, the state space of the process is $E = (0, 0, \text{OFF}) \cup \{(c, i, g) | c \in \mathbb{N}_{>0}, i \in \{1 \dots B\}, g \in \{\text{ON}, \text{OFF}\}\}$. Figure 4.2 shows the structure of the regular portion of the resulting continuous-time Markov chain. As we can see it has a regular pattern for every number of customers $c > 1$. In particular, we can note that the interactions occur only between neighbouring states. This is a quasi-birth-death (QBD) process and we have described it in details in Section 3.4.1.

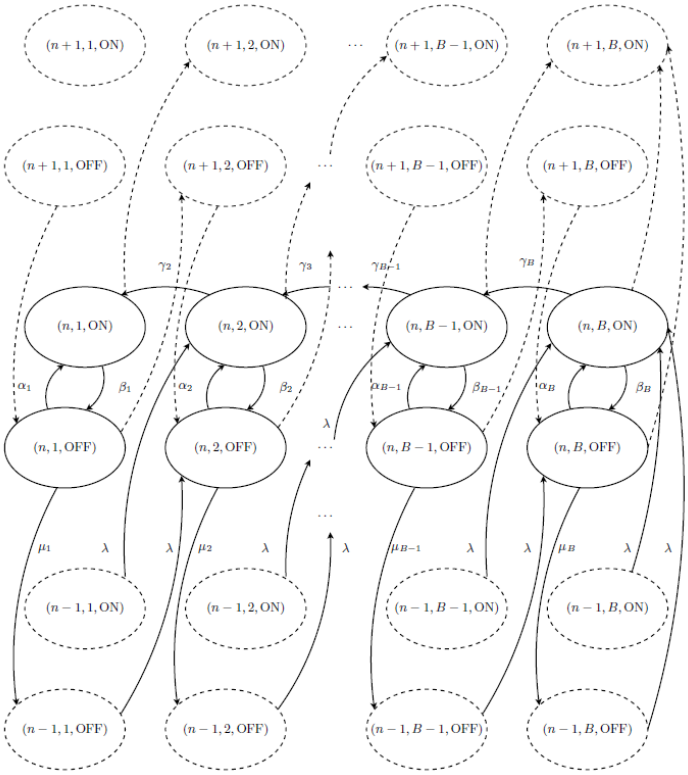

Figure 4.2: The regular blocks of the model for $b = 1$.

4.2 Transition rate matrix

The corresponding transition rate matrix Q of the CTMC is formed by infinite block tridiagonal matrices with the typical regular block structure:

$$Q = \begin{pmatrix} B_{00} & B_{01} & 0 & 0 & 0 & 0 & \dots \\ B_{10} & A_1 & A_2 & 0 & 0 & 0 & \dots \\ 0 & A_0 & A_1 & A_2 & 0 & 0 & \dots \\ 0 & 0 & A_0 & A_1 & A_2 & 0 & \dots \\ & & & \ddots & \ddots & \ddots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}, \tag{4.1}$$

where A_0, A_1, A_2 are squared matrices of size $2B$ and $, B_{00}, B_{01}, B_{10}$ are column vector of size $2B$. We report below the submatrices as defined in the article:

$$A_0(i,j) = \begin{cases} \mu_{\frac{i+1}{2}} & \text{if } i = j \text{ and } i,j \text{ are odd} \\ 0 & \text{otherwise} \end{cases}$$

$$A_1(i,j) = \begin{cases} \alpha_{\frac{i+1}{2}} & \text{if } j = i+1 \ \wedge \ i \text{ is odd} \\ \beta_{\frac{i}{2}} & \text{if } j = i-1 \ \wedge \ i \text{ is even} \\ \gamma_{\frac{i}{2}} & \text{if } j = i-2 \ \wedge \ i \text{ is even} \\ -\sum_{\forall k \neq i} (A_0(i,k) + A_1(i,k) + A_2(i,k)) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$$A_2(i,j) = \begin{cases} \lambda & \text{if } j = i+2 \\ \lambda & \text{if } (i = 2B \vee i = 2B-1) \ \wedge \ j = 2B \\ 0 & \text{otherwise} \end{cases}$$

$$B_{00}(1) = -\lambda$$

$$B_{01}(j) = \begin{cases} \lambda & \text{if } j = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$B_{10}(i) = \begin{cases} \mu_{\frac{i+1}{2}} & \text{if } i \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

We also report the example in case $B = 4$:

$$A_0 = \begin{pmatrix} \mu_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mu_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$A_1 = \begin{pmatrix} * & \alpha_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \beta_1 & * & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & * & \alpha_2 & 0 & 0 & 0 & 0 \\ 0 & \gamma_2 & \beta_2 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & * & \alpha_3 & 0 & 0 \\ 0 & 0 & 0 & \gamma_3 & \beta_3 & * & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & * & \alpha_4 \\ 0 & 0 & 0 & 0 & 0 & \gamma_4 & \beta_4 & * \end{pmatrix},$$

$$A_2 = \begin{pmatrix} 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda \end{pmatrix},$$

where diagonal elements, denoted with $*$, are given by Equation (4.2).

$$B_{10} = \begin{pmatrix} \mu_1 \\ 0 \\ \mu_2 \\ 0 \\ \mu_3 \\ 0 \\ \mu_4 \\ 0 \end{pmatrix}, \qquad B_{00} = \begin{pmatrix} -\lambda \end{pmatrix}, \qquad B_{01} = \begin{pmatrix} \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

4.3 Numerical Solution of Markov Chain

Since the model has an underlying QBD process we can apply the matrix geometric method as described in previous chapter. Thanks to SMC-Solver package we have implemented a MATLAB code of the stochastic model specifically designed for our test suite. MATLAB is a numerical computing environment that allows matrix manipulations and it is particularly suitable for this kind of jobs. We assume that parameters like α, β, γ and μ are given and obtained from experimental measures. Parameter λ is triggered in order to test the system under different load conditions. The input matrices are

built as defined in previous equations to form the infinitesimal generator Q with the requisite block structure to apply the matrix-geometric method. After verified the condition $\pi_A A_2 e < \pi_A A_0 e$, the rate matrix R is computed by the *QBD _FI* function. This is the most computationally expensive operation and in some cases we limit the analysis to a fixed memory size in order to to obtain the results in a reasonable amount of time. The solution of the system of equations $\pi Q = 0$ is easily computed by the *QBD _pi*, which returns the stationary distribution vector π and then we partition vector conformally with Q. Finally, by the closed-matrix-form expression for the mean number of customers in the system derived in [1]:

$$E[N] = \sum_{k=1}^{\infty} k\|\pi_k\|_1 = \|\pi_1(I-R)^2\|_1,$$

and by Little's Law, equation (3.23), we obtain the mean response time

$$E[R] = \frac{E[N]}{\lambda}.$$

During the experimental phase we carried out some tuning to the model and in particular to the construction of the infinitesimal generator submatrices. When the system is busy and we are not under stability conditions with $\rho \geq 1$, i.e., when there are no more memory blocks available, there is no service to the customers. We then considered this situation by setting to zero the service rate when all memory blocks are occupied.

Chapter 5

Markov-Modulated Queueing Model for GC

In this chapter we present our contribution to analyse and quantify the performances of a class of garbage collection policies. As before, we consider a single server in which the garbage collector is represented as a simple two-states system $\{\text{OFF}, \text{ON}\}$ that adopt a *stop-the-world* policy when it is active, i.e., on state ON. We introduce two queueing models that describe the garbage collector behaviour.

5.1 Simplest version

The purpose of this work is to obtain a queueing model as simple as possible. In order to achieve this goal we propose a way to reduce the number of parameters involved with a reasonable loss of accuracy.

The model proposed here is composed by a server that provides a service to the clients by answering their requests, as shown in Figure 4.1. It is very close to the model describe in [1], unless for some conditions that we are going to describe.

Description and assumptions

- Customers are undistinguishable and part of an infinite set that arrive at the system according to a homogeneous Poisson process with rate λ.
- Assuming customer arrivals at an empty system, no other customer arrivals during service time and garbage collector always inactive. Un-

der these conditions the service time is exponentially distributed and independent of the arrival time.

- The scheduling discipline is Processor Sharing.
- The system service rate, unlike [1], does not depend on the amount of memory blocks occupied as long as the system does not use the virtual memory.
- The purpose of the garbage collector is to free the unused memory blocks. Due to the *stop-the-world* policy the customer service is suspended during the operation. The garbage collector is activated according to the following cases:
  (a) The system memory is full and a customer arrival event occurs.
  (b) Activation and deactivation depend on a timer expiration, which is set up with an exponentially distribution. The mean duration of those delays is α^{-1}, and we call α the *activation rate* of the garbage collector. The time taken to the garbage collection operation is β^{-1}, and we call β the *deactivation rate*.
- If the system is empty, i.e., no customer is being processed, the garbage collector can immediately free all the memory blocks.

We proceed by defining the state of the model and the transition diagram. We represent the garbage collector as a switch with two possible states $\{\text{OFF}, \text{ON}\}$. We adopt the stop-the-world policy, thus when the garbage collector is switched on, all the server activities are suspended. This has the result that customers cannot be served, i.e., no departures in Figure 4.1. More formally, the model definition is quite similar to [1], but without considering the rate of freeing memory blocks.

The *state* of the system is defined as a pair (c, s), where c is the number of customers in the system and s is the state of the garbage collector, e.g., the state $(4, \text{ON})$ means that there are 4 customers in the system and that the garbage collector is running. Considering the initial case with the empty queue and garbage collection not active, the state space of the process is $E = (0, \text{OFF}) \cup \{(c, s) | c \in \mathbb{N}_{>0}, s \in \{\text{OFF}, \text{ON}\}\}$. The corresponding Continuous-time Markov Chain is shown on Figure 5.1 and as we can see it has a regular pattern for every number of customers. In particular, we can note that interactions occur only between neighbouring states. This is a quasi-birth-death (QBD) process and we have described it in details in Section 3.4.1.

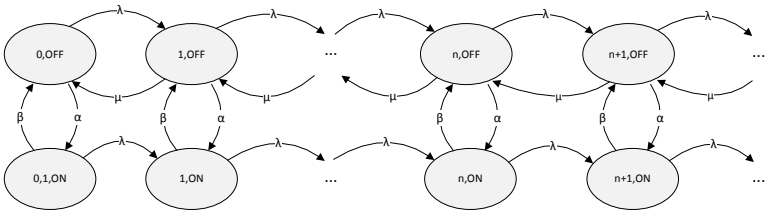

Figure 5.1: CTMC transitions for the queue from 0 to ∞ customers.

According to Kendall's notation, described in Section 3.5.4, this is a $M/G/1$ queueing model, where M stay for Markovian interarrival time distribution, G for General service time distribution and 1 denotes one server. We omitted the letters that specify the system capacity and we considered default values, i.e., infinite size for capacity and population. We assume that all customers who arrive at the server are admitted. Under stability condition the server cannot be busy all the time and this implies a $\rho < 1$.

5.1.1 Transition rate matrix

The corresponding transition rate matrix Q of the QBD Markov chain is formed by infinite block tridiagonal matrices with the following regular block structure:

$$Q = \begin{pmatrix} B_{00} & B_{01} & 0 & 0 & 0 & 0 & \dots \\ B_{10} & A_1 & A_2 & 0 & 0 & 0 & \dots \\ 0 & A_0 & A_1 & A_2 & 0 & 0 & \dots \\ 0 & 0 & A_0 & A_1 & A_2 & 0 & \dots \\ & & & \ddots & \ddots & \ddots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}, \tag{5.1}$$

where $A_0, A_1, A_2, B_{00}, B_{01}, B_{10}$ are square matrices with the same dimension 2. The submatrices are defined as follow:

$$A_0(i,j) = B_{10}(i,j) = \begin{cases} \mu & \text{if } i = j = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$A_1(i,j) = \begin{cases} \alpha & \text{if } i = 1 \ \wedge \ j = 2 \\ \beta & \text{if } i = 2 \ \wedge \ j = 1 \\ -\sum_{\forall k \neq i} (A_0(i,k) + A_1(i,k) + \mathbf{A}_2(i,k)) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

$$A_2(i,j) = B_{01} = \begin{cases} \lambda & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

$$B_{00}(i,j) = \begin{cases} \alpha & \text{if } i = 1 \ \wedge \ j = 2 \\ \beta & \text{if } i = 2 \ \wedge \ j = 1 \\ -\sum_{\forall k \neq i} (B_{00}(i,k) + A_2(i,k)) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Here we can see an example. Diagonal elements, denoted with $*$, are given by Equations (5.2, 5.3).

$$A_0 = B_{10} = \begin{pmatrix} \mu & 0 \\ 0 & 0 \end{pmatrix}, \qquad A1 = \begin{pmatrix} * & \alpha \\ \beta & * \end{pmatrix},$$

$$A_2 = B_{01} = \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}, \qquad B_{00} = \begin{pmatrix} * & \alpha \\ \beta & * \end{pmatrix}.$$

5.2 Improved version

Since initial experimental results have shown that the models had not a very exponential trend, we propose an improved version in which we consider the times (service, response, activation and deactivation) with a more general distribution. To reduce the variance and fit better the data we extend the first model. Until now we have considered the deactivation rate as a single exponential distribution. We split now the time taken to perform the garbage collection operation, i.e., the time that a customer spends waiting for service, into a succession of r exponential phases. These phases are distributed with the same parameter β, they are independent and r is chosen according to the experimental measures. When the garbage collector is active, a client must wait r consecutive interval of time before being served, i.e, the garbage collection deactivation time. We recall that our system is composed of a single-queue server, thus there cannot be more than one client in service at any time.

The new distribution is a typical *Erlang-r distribution*, where r is a parameter and indicates the number of exponential phases. In an Erlang-r distribution the random variable has a lower variance than an exponentially distributed random variable with the same mean, while maintaining the desirable mathematical properties of the exponential [16].

Formally, the density function of an exponentially distributed random variable X with parameter λ is given by

$$f_X(x) \equiv \frac{dF_X(x)}{dx} = \lambda e^{-\lambda x}, \quad x \geq 0,$$

and has expected value and variance $E[X] = \frac{1}{\lambda}$, $\sigma_X^2 = \frac{1}{\lambda^2}$ that are the expectation and variance per phase.

The overall mean and variance of the sum of r independent and identically distributed random variables are given by the following formula:

$$E[X] = r\left(\frac{1}{\lambda}\right) = \frac{r}{\lambda}, \qquad \sigma_X^2 = r\left(\frac{1}{\lambda}\right)^2 = \frac{r}{\lambda^2}. \tag{5.4}$$

For our model we want that the garbage collection operation is distributed according to an Erlang-r distribution with expected value $1/\beta$. We can substitute $E[X] = 1/\beta$ in Equation (5.4) obtaining

$$\frac{1}{\beta} = \frac{r}{\lambda} \Rightarrow \lambda = r\beta.$$

This means that the long run rate at which events occur in the Erlang-r distribution is the number of phases r multiplied by β, the reciprocal of the expected value that is also the rate parameter of the exponentially distributed random variable with the same expectation.

After these considerations we have to redefine the state of the system introducing a new parameter for phase representation. This leads to a triplet (c, i, s) where c is again the number of customers in the system, i, with $(1 \leq i \leq r)$, denotes the current phase in the Erlang-r distribution and s is the state of the garbage collector. For example the state $(4, 3, \text{ON})$ means that there are 4 customers in the system, the garbage collector is running and it is on the third phase. Note that if $r = 1$ we have the same model with the same infinitesimal generator as described in previous section.

Figure 5.2 shows the new state transition diagram where the states are arranged into levels according to the number of deactivation phases and the number of customers present. From the structure of the state transition

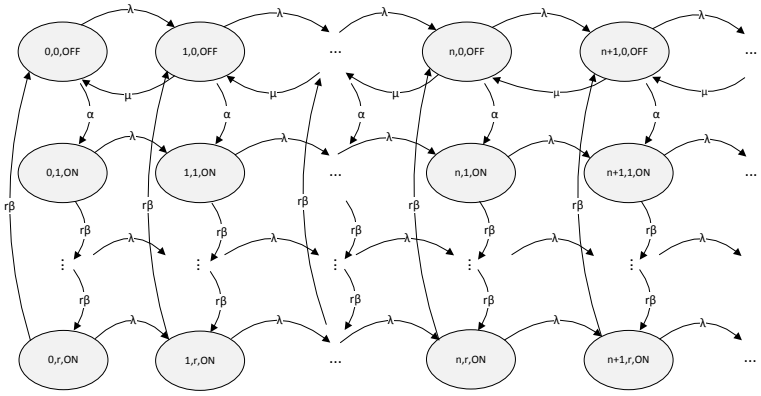

Figure 5.2: CTMC transitions for the queue from 0 to ∞ customers with r phases.

diagram we can see that the transition rate matrix has the typical block tridiagonal form and, as the early version, it is again a QBD process.

5.2.1 Transition rate matrix

The corresponding transition rate matrix Q of the QBD Markov chain is formed by infinite block tridiagonal matrices with the following regular block structure:

$$Q = \begin{pmatrix} B_{00} & B_{01} & 0 & 0 & 0 & 0 & \dots \\ B_{10} & A_1 & A_2 & 0 & 0 & 0 & \dots \\ 0 & A_0 & A_1 & A_2 & 0 & 0 & \dots \\ 0 & 0 & A_0 & A_1 & A_2 & 0 & \dots \\ & & & \ddots & \ddots & \ddots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}, \tag{5.5}$$

where $A_0, A_1, A_2, B_{00}, B_{01}, B_{10}$ are square matrices with the same dimension $r + 1$. The submatrices are defined as follow:

$$A_0(i,j) = B_{10}(i,j) = \begin{cases} \mu & \text{if } i = j = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$A_1(i,j) = \begin{cases} \alpha & \text{if } i = 1 \ \wedge \ j = 2 \\ r\beta & \text{if } (i \in [2,r] \ \wedge \ j = i+1) \\ & \quad \vee \ (i = r+1 \ \wedge \ j = 1) \\ -\sum\limits_{\forall k \neq i} (A_0(i,k) + A_1(i,k) + \mathbf{A}_2(i,k)) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

$$A_2(i,j) = B_{01} = \begin{cases} \lambda & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

$$B_{00}(i,j) = \begin{cases} \alpha & \text{if } i = 1 \ \wedge \ j = 2 \\ r\beta & \text{if } (i \in [2,r] \ \wedge \ j = i+1) \\ & \quad \vee \ (i = r+1 \ \wedge \ j = 1) \\ -\sum\limits_{\forall k \neq i} (B_{00}(i,k) + A_2(i,k)) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

Here we can see an example with three phases, i.e., $r = 3$. Diagonal elements, denoted with $*$, are given by Equations (5.6, 5.7).

$$A_0 = B_{10} = \begin{pmatrix} \mu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \qquad A1 = \begin{pmatrix} * & \alpha & 0 & 0 \\ 0 & * & r\beta & 0 \\ 0 & 0 & * & r\beta \\ r\beta & 0 & 0 & * \end{pmatrix},$$

$$A_2 = B_{01} = \begin{pmatrix} \lambda & 0 & 0 & 0 \\ 0 & \lambda & 0 & 0 \\ 0 & 0 & \lambda & 0 \\ 0 & 0 & 0 & \lambda \end{pmatrix}, \quad B_{00} = \begin{pmatrix} * & \alpha & 0 & 0 \\ 0 & * & r\beta & 0 \\ 0 & 0 & * & r\beta \\ r\beta & 0 & 0 & * \end{pmatrix}.$$

5.3 Numerical Solution of Markov Chains

The purpose of this work is to obtain a numerically tractable model that allows us to to predict the performances in terms of throughput and average response time. Since there is no customer loss or creation in the queue, under stability conditions the throughput is equal to the arrival rate λ and by Little's Law (3.1) we can derive the mean response time of the system:

$$E[R] = \frac{E[N]}{\lambda}. \tag{5.8}$$

The average number of customers in the system is given by Formula (3.21).

Our goal is to find the stationary distribution π that is the numerical solution of the corresponding Markov chain. Furthermore, we know that both the stochastic models presented here have the typical behaviour of QBD processes, thus the equations and matrix geometric method described in Section 3.4.2 can be applied to find the solution.

5.3.1 Assumptions

To build the model and solve the system of equations $\pi Q = 0$ we use MATLAB. Thanks to SMC-Solver [3] (we already mention it on Chapter 3) we can use a useful package to compute efficiently the rate matrix R and the stationary vector π.

We assume that parameters α, β and μ are given and in particular they are obtained from experimental results by a series of tools and scripts specifically designed for this purpose. Parameter λ is given and it is triggered by us in order to test the system under different load situations. Refer to the following chapter for validation and technical details.

5.3.2 Matrix geometric approach

We apply the matrix geometric method to the Markov chains of simplified and improved version of our Markov-modulated queue models. As we already said the infinitesimal generators Q have the correct QBD structure and stability conditions can be verified with inequality $\pi_A A_2 e < \pi_A A_0 e$. We proceed with the computation of rate matrix R and for this operation we call the *QBD_FI* function provided by SMC-Solver. Once R is obtained we call the *QBD_pi* function that returns the stationary distribution vector π. The solution is not yet complete because the components π_i need to be transformed into subvectors of length m (the phase), i.e., $\pi_i = (\pi(i,1), \pi(i,2), \ldots, \pi(i,m))$. Finally, Equations (3.21, 5.8) are applied to obtain the mean response time of the system.

The following code show the function for the implementation of the matrix geometric method and as we can see the advantages provided by the SMC-Solver package are remarkable. We omitted the part for the matrices creation and the rest of the program.

Code 5.1: Matlab implementation of matrix geometric method

```
function [er] = mean_response_time(r,A0,A1,A2,B0,B1)
    % size of subvectors
    % r is the number of Erlang-r distribution phases
    dim = r+1;
    % SMC-Solver package inclusion
    addpath('QBD');
    % compute the rate matrix R
    [ G,R ] = QBD_FI(A0,A1,A2);
    % system resolution
    % the stationary distribution vector is returned
    pigc=QBD_pi(B0,B1,R);
    it=1;
    jt=1;
    % new distribution vector
    %partitioned conformally with Q
    pig = zeros(1, size(pigc,2)/dim);
    while (it < size(pigc,2))
        % sum block probabilities
        % to have subvectors of length 'dim'
        for col=it:it+dim-1
            pig(jt) = pig(jt) + pigc(col);
        end
        it = col+1;
        jt = jt+1;
    end
    % average number of customers in the system
    umean = 0;
    for it=1:size(pig,2)
        umean = umean+(it-1)*pig(it);
    end
    % mean response time computation
    er=umean/l;
end;
```

Chapter 6

Model validation

This chapter describes the validation of the models described in Chapter 4 and 5. It starts presenting the architecture of the test environment, then it shows a comparison between the real data collected from the experiments and the predictions provided by the numerical solution of QBD Model for GC and Markov-Modulated Queueing Model for GC.

The presented measurements have twofold purposes. First, they provide an overview of the mean response time and memory usage of an application under different workload conditions. Second, part of the values obtained are used to parametrize the models and to validate them.

6.1 Architecture

To test the data we set a test environment to reproduce a real scenario as far as possible. The main components are a server and a client located in the server room of the *Department of Environmental Sciences, Informatics and Statistics* (DAIS). Table 6.1 reports their technical specifications of particular interest for our experiments. The two machines are directly connected via a point-to-point connection and the client has the role of *gateway*. This allows us to connect remotely to the server, but avoiding other network traffic that may be invalidate the measures. Initially the system was connected through the department LAN and the mean response time resulted higher than what we expected. After some testing we figured out that the problem was the network overhead due to the high traffic in the network department, so we decided to connect directly the server to the client with a dedicated cable.

Hardware specifications

	CPU	Cores	RAM
Client	Intel(R) Xeon(R) E5410 @ 2.33 GHz	8	16 GB
Server	AMD Sempron(tm) Processor 3400+	1	4 GB

Table 6.1: Client and Server hardware specifications.

6.2 Server overview

6.2.1 Applications

Our purpose is to study the garbage collection policies of the Java technology, hence we have looked for a real application that can be reached remotely. In this way we can physically split the tasks for the client and the server, allowing us to monitor only the server performances like CPU usage and memory allocation. The application that satisfied this requirements is Magnolia CMS[1], an open-source content management system (CMS) developed by Magnolia International Ltd.

Magnolia CMS allows companies to orchestrate services, sales and marketing across all digital channels and devices. It is used by many companies in various sectors like: defense, news channels, insurances, financial services and much more. Magnolia CMS is based on the *Content Repository API for Java* (JCR), a standard that defines a repository for managing content. Magnolia offers a free Community Edition and a commercial licensed Enterprise Edition. While the company will not offer support for the community edition it is the same code base as the core of Enterprise Edition. The Enterprise Edition has a more robust feature set aimed at enterprise customers and is certified to work with different application servers and an alternative repository. We chose to use the Community Edition (CE) that can be obtained under the *GNU General Public License* (GPL) and it is sufficient to validate the analytical models presented here. Application's configuration is a huge task that requires some expertises with the environment. Since we are interested in performance evaluation, we do not require a specialized configuration and for our purposes it was sufficient to set up the demo Website disabling all caching by default.

We also developed a servlet in Java that fill the memory with some random squared matrices when a client request occurs. In order to introduce some computation and not just memory allocation to increase the customer

[1]Website: http://www.magnolia-cms.com/

service time, the servlet also performs the matrix multiplication of two matrices. The product matrix is computed by the Efficient Java Matrix Library (EJML), a faster single threaded linear algebra library for manipulating dense matrices released under Apache v2.0 license. With this application and several matrices we tested the trend of the models under different load conditions in terms of customer arrivals and memory occupancy.

6.2.2 Configuration

The server is a desktop PC with specifications described in Table 6.1. It is configured with Ubuntu Server 12.04.3 LTS 64-bit version and Magnolia CE 5.1 which also include Apache Tomcat 7.0.40, an open source software implementation of the Java Servlet and JavaServer Pages technologies. To deploy the Java applications we have installed the OpenJDK Runtime Environment (version 7) and to test the models we left the Java Virtual Machine (JVM) configuration as default without performing any tuning operations. Tuning a JVM, as we said in Chapter 1, is a very difficult operation and it is necessary for every single application. We assume that the default configuration sets correctly the parameters to obtain the best performances as specified in the documentation. In particular, we are interested in measuring the garbage collection frequency, i.e., the activation rate of the garbage to validate the queueing models analysed. The only changes regard the log file configuration. As default Java does not log to the disk any information about the garbage collection. Accordingly to the Java HotSpot manual we log every garbage collection event occurred, which includes activation timestamps, duration, heap size before and after collections. We could use a profiling software like VisualVM to monitor the JVM, but this would introduce a memory overhead as we noticed with some experimentations and thus we design a specific solution.

The machine has 4 GB of available RAM and for our tests it can be excessive, although it is quite limited for modern systems. With less memory available we can quickly observe the system behaviour when it reach the memory boundaries. However, as we noticed from experimental results the system never fills up the heap under stability conditions.

During the parameters acquisition phase we realized there were performance problems. The mean response time measured in a queue with many customers resulted significantly less than service time, i.e, the system performed better with an high arrival rate than a single arrival! Apparently

this is very strange, but we forgot to consider that modern computer architectures adopt the *dynamic frequency scaling*, a technique of ramping a processor's frequency in order to achieve performance gains. Dynamic frequency scaling reduces the number of instructions a processor can issue in a given amount of time and modern CPU are strongly optimized for low power idle state. Thus when we try to measure service time, the mean time taken by single requests which are sent after a given amount of time, the system requires more time to process the requests and this results in a rough measure. Therefore, we disabled this feature.

6.3 Client description and tools

The client is a very powerful machine used to test the server applications and performs the data analysis (see Table 6.1). It includes the operating system Ubuntu Server 12.04.3 LTS 64-bit version, the OpenJDK Java software, MATLAB, and Python 2.7.3.

At the beginning we spent a lot of time in finding out the best way to test and monitor the server applications. With Java software are included a series of useful graphical tools, like JConsole and VisualVM, specifically designed to monitor JVM status. This applications have multiple advantages like real time profiling and performance analysis. However the fastest and most lightweight technology to get the data is implemented, there is always a minimal overhead on monitored applications and they are recommended for development and prototyping, but not for production environments. Since we want to simulate a production environment as accurate as possible, we chose to do only a posteriori performance analysis to not overload the JVM with others activities.

IBM provides tooling and documentation to assist in the understanding, monitoring, and problem diagnosis of applications and deployments running IBM Runtime Environments for Java, like the "IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer" (GCMV). GCMV is a tool which allow users to visualize and analyse the memory usage and garbage collection activity. Furthermore, it can also analyse log file of normal JVM (with specific logging options) that does not runs on IBM systems. Another possible tool is "gclogviewer", a free open source tool to visualize data produced by the JVM logging options. We tried all this stuff, but we needed something more versatile and we decided to implement a personalized solution. For the purpose, we developed a series

of Python and Shell scripts to parse the JVM log file and collect the data in the practical CSV file format. In this way the client interacts with the server only for service request, avoiding possible JVM overload and network overhead.

Once the parser was implemented we looked for a tool for benchmarking the server applications with the following characteristics:

- support to multithreading HTTP requests;
- requests rate variation;
- number of connections variation;
- mean response time computation.

At the beginning we have developed a Python script to retrieve URLs from the server application. The code was programmed to use Requests, an elegant and simple HTTP library to measure the response time of URL retrieval, and "multiprocessing" package which offers concurrency and the possibility to leverage multiple processors on the client machine (we discarded multithreading because threads are not well supported in Python). Unfortunately the processes creation requires high resources than we expected, resulting in an upper bound for the requests rate, i.e., the customer arrival rate. We tried to reproduce the same code in Java using multithreading programming, but the mean response time measured, resulted very high compared to Python results. We suppose that the Java code execution introduces some overhead on the measures and we can't trust them.

Finally we found Httperf[2], a tool for measuring web server performances maintained by Martin Arlitt of HP Research, Mark Nottingham of Yahoo! and Ted Bullock. Httperf, which is released under GPL v2 license, provides a flexible facility for generating various HTTP workloads and for measuring server performances. Its focus is not on implementing one particular benchmark but on providing a robust, high-performance tool that facilitates the construction of both micro- and macro-level benchmarks. The three distinguishing characteristics of Httperf are its robustness, which includes the ability to generate and sustain server overload, support for the HTTP/1.1 and SSL protocols, and its extensibility to new workload generators and performance measurements. Mosberger and Tai [13] describe design and implementation of Httperf. They also discuss some of the experiences and

[2]Website: http://www.hpl.hp.com/research/linux/httperf/

insights gained while realizing this tool. Citing the authors Conclusion: Httperf has proven useful in a number of web-related measurement tasks and is believed to be flexible and performant enough that it could provide a solid foundation to realize macro-level benchmarks such as SPECweb.

We therefore decided to use this tool to test ours server applications. After checking that there were no problems and limitations as experienced with our previous solutions, we developed a Python code as a practical interface to call *httperf* command and to parse the output. More specifically, with our script we can specify:

- the test duration or the number of requests;
- the step of the request rates, i.e., customer arrival rates to test;
- the HTTP method to use (GET or POST);
- the server application to benchmark.

All the results, e.g, the mean response time, are collected in a CSV file format in order to be analysed with MATLAB.

6.4 Experiments

6.4.1 Description

This section explains how we conduct tests and how we analysed the collected data. We simulated a real production environment to validate the models keeping themselves as simple as possible. Therefore, our architecture is very simple and since the server has a single-core CPU we forced the JVM to adopt only the serial garbage collection algorithm (see Section 2.2.1 for the Serial Collector's description).

We proceed now describing typical test execution under stability conditions and without the need of virtual memory usage. We remember that the application and client caching are disable. The test session starts measuring the mean service time, if not already done, when the application is up and running. The service time is the time taken to retrieve a Web page and we repeat this operation whenever the application is started. The mean service time is computed from a sample of 100 requests every 5 seconds, a time interval sufficient to be sure that there are no other customers in the queue. Once the operation complete, we do not perform any request for a certain time interval, so that the server can wipe the queue and reduce the

workload. In the next step the client starts to send requests according to a homogeneous Poisson process with the starting rate λ for a long time to allow the application to reach equilibrium. Usually we set the duration of a single test to 30 minutes, then the client takes a break allowing the server to rest. Tests continuing on until the ending rate λ is reached at the specified step.

When the testing session has been completed, the script downloads the log file from the server, parses it to keep only the events corresponding to each single task and prepares two CSV files which include all the information needed for the analysis. The first file contains the results: the arrival rate λ_i used for the specific test i, the mean service time and the mean response time measured. Httperf provides a precision of 10^{-4} seconds for time measures. Its output shows this values in units of milliseconds and we decided to store them in seconds. The second one contains the garbage collection logs:

- the timestamps of events incrementally distributed from the application starting time;
- the heap size before the garbage collections;
- the heap size after the garbage collections;
- the max heap size allocated;
- the duration of the garbage collection operations.

The JVM log file represents heap dimensions in kilobytes and garbage collection durations in seconds with a precision of 10^{-6} seconds. We also stored a file to match the results with the corresponding garbage collection events in the log file. The whole process is automated by our Python script.

6.4.2 Analysis of data

All data collected during the testing sessions are processed by a MATLAB code. At the beginning, a function computes the parameters needed to test the models. More specifically:

- the service rate, μ, computed with equation

$$\mu = \frac{1}{E[T]},$$

  where $E[T]$ is the mean service time measured;

- the activation rate, α, which is the reciprocal of the mean activation time, a vector of time differences between the timestamp values and the corresponding previous value;
- the deactivation rate, β, computed with equation

$$\beta = \frac{1}{E[D]},$$

where $E[D]$ is the average of the garbage collection operation durations of a single test.

In addition, for the QBD Model for GC the following parameters are computed:

- The garbage collection rate, γ. Obtained dividing the average quantity of memory freed, F, times the block size, B, by $E[D]$ (defined as before)

$$\gamma = \frac{F \times B}{E[D]}.$$

- The size of the memory, S. It is given dividing the max heap size by the block size B.

The block size B is computed measuring the quantity of memory occupied to serve a single request. For this operation we used a specific test and a specific servlet that explicitly call the garbage collector by the Java function *System.gc()*. We forced the garbage collections and client requests frequencies at the rate of 1 every 5 seconds. The requests are spaced respect to the garbage collection activations in order to have a request between two garbage collection events. More specifically, the test waits to start until a new log line has been written which means that a garbage collection operation has completed. This solution allows us to measure the memory allocated necessary to process a single arrival and this measure is computed as the difference between the heap size before and after collection. The resulting block size B is the average value of a sample of 100 measurements. The accuracy of this value was proven with the Matrices servlet. We know the memory size needed to store the matrices and we noticed a close result comparing this value in kilobytes to B.

Java documentation explains that the use of *System.gc()* function is only a suggestion to the JVM, which decides by itself the right moment to the collect garbage. However, after some experiments we noticed that if we

set a deterministic call and the system has a low workload the garbage operation takes place when invoked. This is not true when we tried to force the garbage collector activation following a Poisson process, i.e, setting the activation rate accordingly to an exponential distribution.

The next step of the MATLAB code is the implementation of the two stochastic models. It builds the matrices according to the equations presented in the respectively chapters and it computes the stationary distribution vectors π using the SMC-Solver package. The program repeats the operation until all the tests have been analysed and in the meanwhile it compiles a report file in the CSV format.

6.5 Results

Results of the testing sessions on Magnolia CMS and Matrices applications are now presented, including comparisons between the stochastic models and experimental data. This results are collected and plotted from the Report files obtained by the MATLAB code. Moreover, every table shows the following values:

- the customer arrival rate tuned, λ, expressed in requests per second;
- the service rate, μ, which is the same for a whole test session and expressed in customers served per second;
- the GC activation rate, α, expressed in activations per second;
- the GC deactivation rate, β, expressed in deactivations per second;
- the garbage collection rate, γ, expressed in freed memory blocks per second;
- the experimental mean response time, R, expressed in seconds;
- the mean response time, Q, given by the QBD Model for GC and expressed in seconds;
- the mean response time, M, given by the Markov-Modulated Queueing Model for GC and expressed in seconds;
- the prediction error of the two models, denoted with $Error(Q)$ and $Error(M)$.

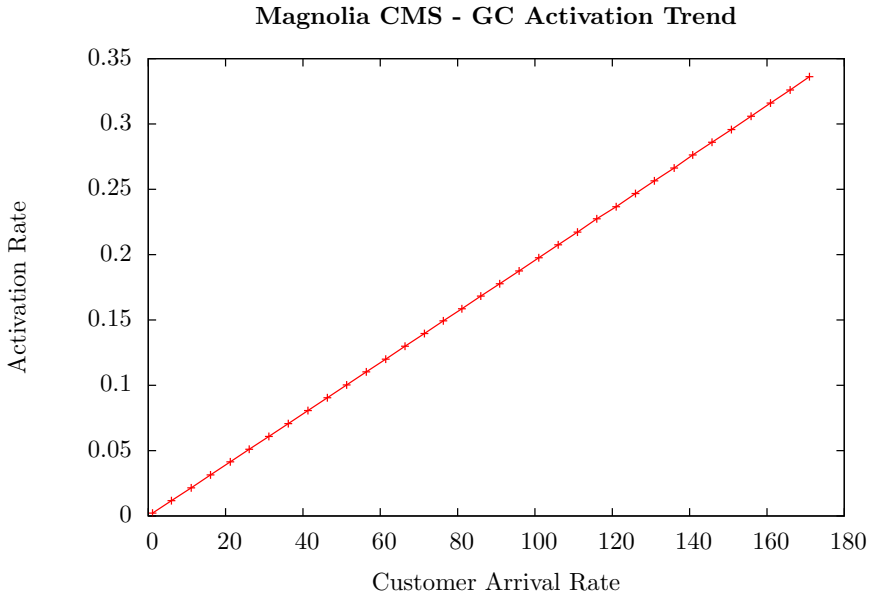

Figure 6.1: α in function of λ, in Magnolia CMS.

6.5.1 Results for Magnolia CMS

The first results we discuss have been collected running several test sessions with Magnolia CMS and here we report one of these. We remember that a single test run for about 30 minutes in order to take the measures at steady-state and consists of several requests at specified rate λ. At the time we tested the application the available version was Magnolia CE 5.1 which included Apache Tomcat 7.0.40. We left the default configuration which sets an heap size of 512 MB and we limit the server's RAM to 1 GB.

Table 6.2 shows the indices for the models parametrization for each arrival rate tested. In this session we tested the customer arrival rate $\lambda \in [1; 171]$ with step $k = 5$ and for this interval the system is under stability conditions with $\rho < 1$. For values of $\lambda > 171$ the mean response time increases significantly and the client can't maintains the specified customer arrival rate because the server is unstable, thus the maximum ρ reached is $\rho = 0.84$. From the values on Table 6.2 we can observe a relation between all the parameters, except for μ that is constant for the whole session. We can see that with the increases of λ increases the frequency of activation of the garbage collector (α), but the rates of collection (β) and freeing memory (γ) slow down. The frequency of garbage collection is linear increasing (Figure 6.1) and it is strictly related to the heap occupancy (Figure 6.2).

Table 6.3 shows the performance indices for each arrival rate tested and

Magnolia CMS - Rates

λ	μ	α	β	γ
1.10	204.0816	0.0022	17.5085	2143.3980
6.00	204.0816	0.0117	17.1879	2107.0523
11.10	204.0816	0.0215	17.1066	2097.1198
16.10	204.0816	0.0314	17.0105	2085.3680
21.20	204.0816	0.0414	16.9224	2074.5031
26.10	204.0816	0.0510	16.9275	2075.1714
31.20	204.0816	0.0609	16.8684	2067.9282
36.20	204.0816	0.0707	16.7719	2056.0498
41.30	204.0816	0.0807	16.7539	2053.7907
46.30	204.0816	0.0905	16.7361	2051.6066
51.30	204.0816	0.1003	16.6985	2047.0156
56.40	204.0816	0.1103	16.5526	2029.0483
61.40	204.0816	0.1200	16.5578	2029.6628
66.40	204.0816	0.1299	16.4987	2022.4360
71.40	204.0816	0.1396	16.4862	2020.8798
76.30	204.0816	0.1493	16.4170	2012.3451
81.10	204.0816	0.1586	16.4738	2019.3140
86.00	204.0816	0.1683	14.0150	1722.1657
90.90	204.0816	0.1777	16.3198	2000.3062
95.90	204.0816	0.1875	16.2629	1993.3129
101.00	204.0816	0.1976	16.1638	1981.1060
106.00	204.0816	0.2075	16.1551	1980.0095
111.00	204.0816	0.2172	16.1094	1974.2518
116.00	204.0816	0.2274	13.7558	1691.8825
121.00	204.0816	0.2367	15.9742	1957.4421
126.00	204.0816	0.2468	15.7727	1932.5461
130.90	204.0816	0.2565	15.8374	1940.5389
136.00	204.0816	0.2663	15.6496	1917.1883
140.80	204.0816	0.2763	13.9953	1716.8053
145.80	204.0816	0.2860	14.2170	1746.3696
150.80	204.0816	0.2957	15.2109	1862.7776
155.90	204.0816	0.3059	13.7725	1688.3637
160.90	204.0816	0.3160	13.9119	1707.5468
166.00	204.0816	0.3260	13.3626	1636.8564
171.00	204.0816	0.3362	13.4628	1651.1633

Table 6.2: Rates of Magnolia CMS to parametrise the models.

Magnolia CMS - Performance indices

Customer Arrival Rate	$R(s)$	$Q(s)$	$M(s)$	**Error(Q)**	**Error(M)**
1.10	0.0049	0.0049	0.0049	0.01	0.01
6.00	0.0049	0.0051	0.0051	0.03	0.04
11.10	0.0049	0.0052	0.0053	0.06	0.07
16.10	0.0051	0.0053	0.0054	0.05	0.07
21.20	0.0052	0.0055	0.0056	0.06	0.09
26.10	0.0054	0.0057	0.0058	0.05	0.08
31.20	0.0056	0.0058	0.0061	0.04	0.08
36.20	0.0059	0.0060	0.0063	0.02	0.07
41.30	0.0061	0.0063	0.0065	0.03	0.07
46.30	0.0064	0.0065	0.0068	0.01	0.06
51.30	0.0067	0.0067	0.0071	0.00	0.06
56.40	0.0071	0.0070	0.0074	0.02	0.04
61.40	0.0075	0.0073	0.0077	0.03	0.03
66.40	0.0078	0.0076	0.0081	0.03	0.03
71.40	0.0084	0.0079	0.0084	0.06	0.00
76.30	0.0088	0.0083	0.0088	0.06	0.00
81.10	0.0092	0.0086	0.0092	0.06	0.00
86.00	0.0128	0.0093	0.0101	0.28	0.21
90.90	0.0105	0.0096	0.0102	0.09	0.03
95.90	0.0114	0.0101	0.0108	0.12	0.05
101.00	0.0123	0.0107	0.0114	0.13	0.07
106.00	0.0135	0.0113	0.0121	0.16	0.10
111.00	0.0144	0.0121	0.0129	0.16	0.10
116.00	0.0222	0.0134	0.0146	0.40	0.34
121.00	0.0178	0.0139	0.0148	0.22	0.17
126.00	0.0203	0.0150	0.0160	0.26	0.21
130.90	0.0218	0.0162	0.0172	0.26	0.21
136.00	0.0247	0.0177	0.0188	0.28	0.24
140.80	0.0363	0.0202	0.0215	0.44	0.41
145.80	0.0412	0.0223	0.0235	0.46	0.43
150.80	0.0397	0.0244	0.0253	0.39	0.36
155.90	0.0600	0.0299	0.0301	0.50	0.50
160.90	0.0647	0.0366	0.0341	0.43	0.47
166.00	0.0959	0.0561	0.0408	0.41	0.57
171.00	0.1265	0.1243	0.0484	0.02	0.62

Table 6.3: Performance indices of Magnolia CMS. From left to right: Customer Arrival Rate; Mean Response Time measured, R; Predictions Q and M; Prediction errors of Q and M.

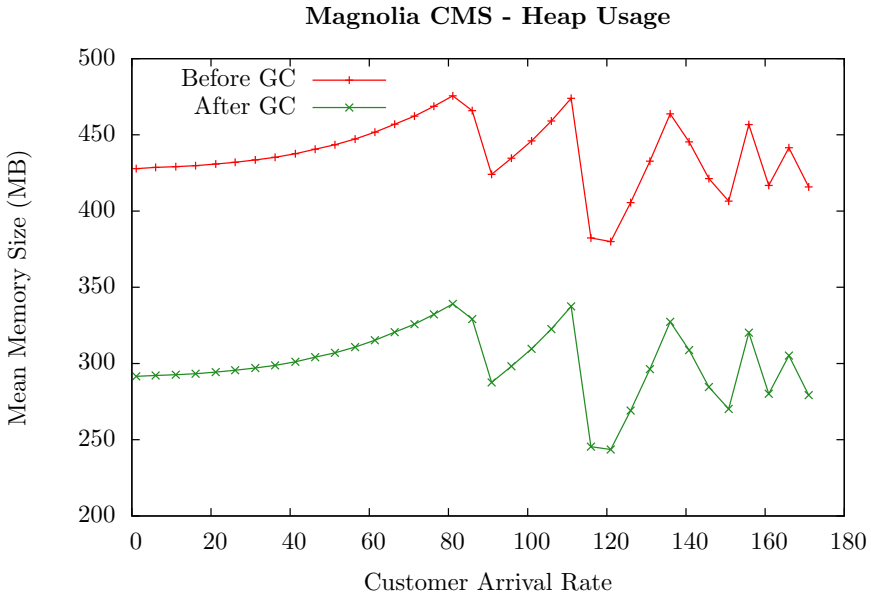

Figure 6.2: Mean memory size before and after garbage collection events, in Magnolia CMS.

the prediction errors of the two queuing models. Results are quite good, except for some outliers on the measurements due to possible overhead introduced by the server, and are plotted on Figure 6.3 for a better view. As we can see, both of the models estimate correctly the mean response time until the server has a sufficient memory availability and the customer arrival rate is moderate. When the frequency of garbage collection and the size of garbage collected increase, also the mean response time and prediction errors increase. In this situation, that happen when $\rho > 0.5$, the CPU bounds are noticeable and the server spends time to handle customer arrivals and to collect a lot of garbage, as we can see on Figure 6.2. QBD Model for GC fits better the experimental data since it take care of the number of memory blocks occupied and γ, the rate to free a block of memory. The overall results are acceptable and both the models predict correctly the mean response time as the policy adopted by the JVM in presence of a moderate workload.

Figure 6.4 compares the two versions of Markov-Modulated Queueing Model for GC. In particular, it includes the prediction of mean response time obtained with two Erlang-r distributions of parameters $r = 2$ and $r = 3$. From this result we can conclude that the mean response time has an exponential trend and the Erlang-r distributions do not add any

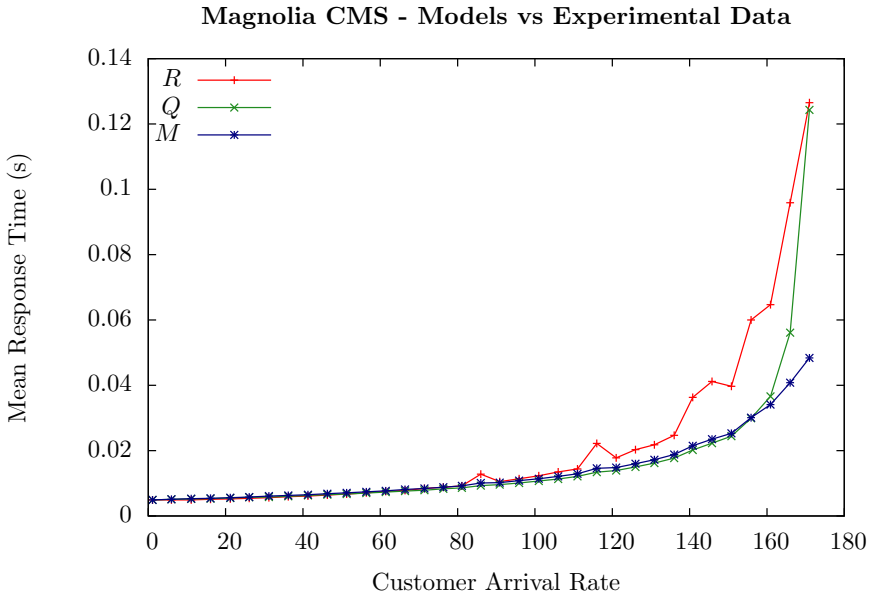

Figure 6.3: Comparison between experimental data R, predictions Q and M.

improvements. Therefore, we also avoided to report the performance indices obtained with these distributions because the predictions are poor respect those on Table 6.3.

6.5.2 Results for Matrices

The second and last results we discuss is a test session obtained with our Matrices servlet. As for Magnolia CMS we run several testing sessions and every single measurement for about 30 minutes. The application runs under Apache Tomcat 7.0.42, the last available version at the time we used it. For this testing phase we limit the server RAM to 1 GB and we sets the maximum heap size of the JVM to this value. When a customer arrive at the server the application allocates the memory with some matrices and perform the matrix multiplication of two squared matrices of dimension 150. This is a sufficient value to introduces additional computation before serve a request. The total amount of memory necessary to serve a customer is about 1.5 MB.

Table 6.4 shows the indices for the models parametrization for each arrival rate tested and again we report only the measurements under stability conditions with $\rho < 1$. We tested the customer arrival rate $\lambda \in [1; 66]$ with step $k = 5$ and for this session the maximum ρ reached is $\rho = 0.80$. As for previous results the parameters trend is the same. When the system

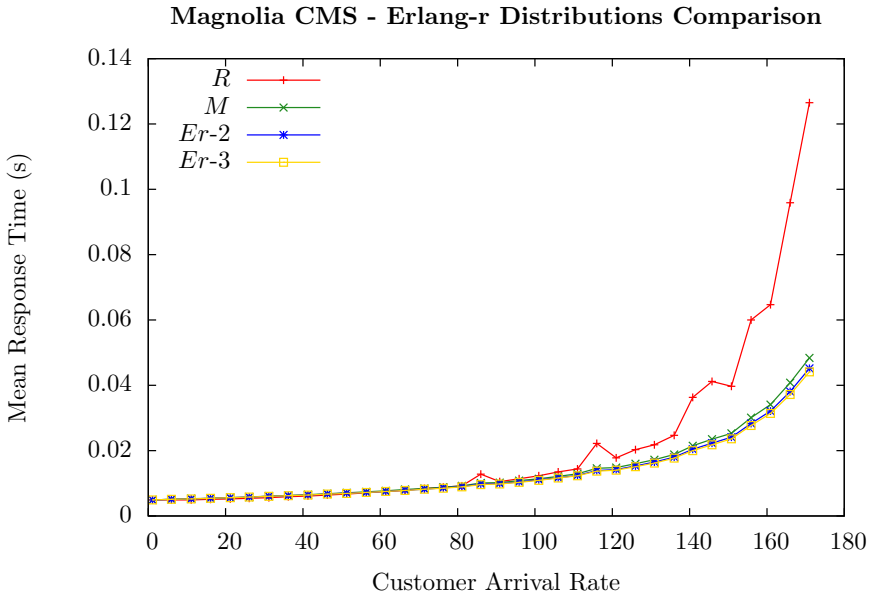

Figure 6.4: Comparison between experimental data R, prediction M and two Erlang-r distributions predictions.

increases the garbage collection frequency to freeing the memory, we notice also the increase of response times. Figure 6.5 and Figure 6.6 clearly show this behaviour. We also tried to increase the amount of memory allocated for each customer arrival, but the server could not handle the workload with the result of performances significantly degraded.

Table 6.5 shows the performance indices for each arrival rate tested and the prediction errors of the two queuing models. Again, as for Magnolia CMS we have good results with $\rho < 0.5$ and we notice and increasing of mean response time when the memory usage increases. From Figure 6.7 we can see the exponential trend of the model predictions and this time without outliers. The two queueing models are overlapping and provide the same results for all the customer arrival rates tested. Furthermore, the prediction error is acceptable and both the models provide the mean response time as the policy adopted by the JVM in presence of a moderate workload.

Also for Matrices we have compared the prediction obtained with two Erlang-r distributions of parameters $r = 2$ and $r = 3$. Figure 6.8 clearly shows that the three predictions are overlapping and we can derive that the Erlang-r distributions do not add any improvements. Even this time we avoided to report the performance indices obtained with these distributions because the values are the same of those on Table 6.5.

Matrices - Rates

λ	μ	α	β	γ
1.10	81.9672	0.0111	171.7992	16195.3701
6.00	81.9672	0.0638	176.1770	16610.8468
11.10	81.9672	0.1179	166.4668	15701.3183
16.10	81.9672	0.1708	171.5742	16180.5551
21.20	81.9672	0.2249	161.0354	15188.0489
26.30	81.9672	0.2792	160.0612	15095.8847
31.30	81.9672	0.3324	152.7656	14406.2771
36.20	81.9672	0.3854	158.7720	14973.9193
41.30	81.9672	0.4390	152.6841	14399.7961
46.40	81.9672	0.4931	138.7065	13080.4519
51.40	81.9672	0.5468	128.3331	12103.6680
56.40	81.9672	0.5998	117.8698	11102.5763
61.50	81.9672	0.6554	99.8260	9398.1063
66.40	81.9672	0.7447	59.5522	5339.4518

Table 6.4: Rates of Matrices to parametrise the models.

Matrices - Performance indices

Customer Arrival Rate	$R(s)$	$Q(s)$	$M(s)$	**Error**(Q)	**Error**(M)
1.10	0.0122	0.0124	0.0124	0.01	0.01
6.00	0.0130	0.0132	0.0132	0.01	0.01
11.10	0.0141	0.0141	0.0141	0.00	0.00
16.10	0.0155	0.0152	0.0152	0.02	0.02
21.20	0.0172	0.0165	0.0165	0.04	0.04
26.30	0.0191	0.0180	0.0180	0.06	0.06
31.30	0.0215	0.0198	0.0198	0.08	0.08
36.20	0.0243	0.0220	0.0220	0.10	0.10
41.30	0.0281	0.0248	0.0248	0.12	0.12
46.40	0.0332	0.0284	0.0284	0.15	0.14
51.40	0.0398	0.0331	0.0332	0.17	0.17
56.40	0.0491	0.0399	0.0399	0.19	0.19
61.50	0.0659	0.0504	0.0504	0.24	0.23
66.40	0.1135	0.0697	0.0699	0.39	0.38

Table 6.5: Performance indices of Matrices. From left to right: Customer Arrival Rate; Mean Response Time measured, R; Predictions Q and M; Prediction errors of Q and M.

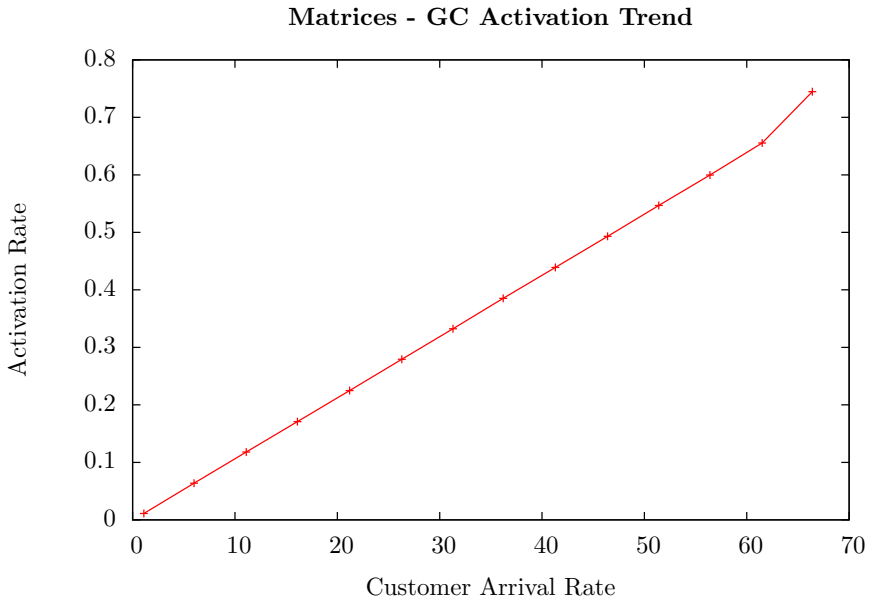

Figure 6.5: α in function of λ.

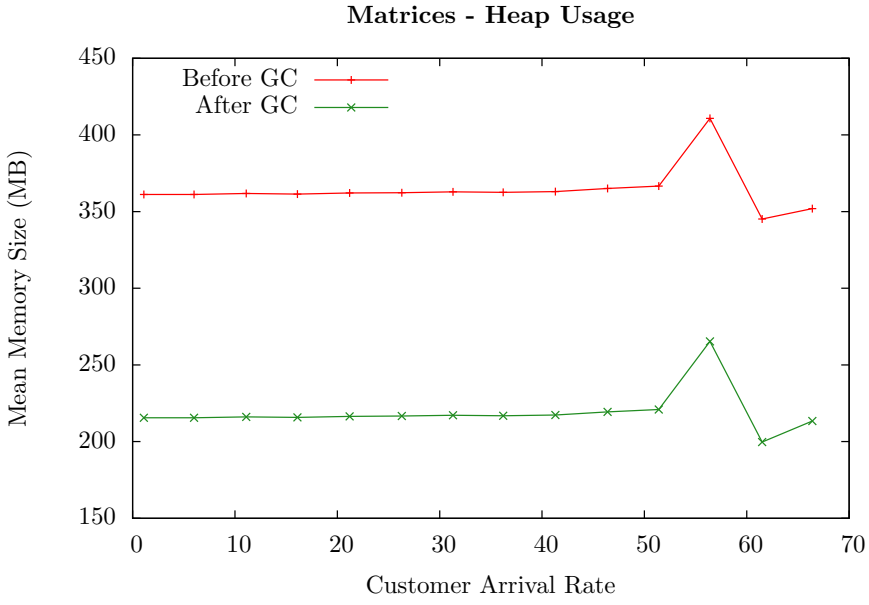

Figure 6.6: Mean memory size before and after garbage collection events.

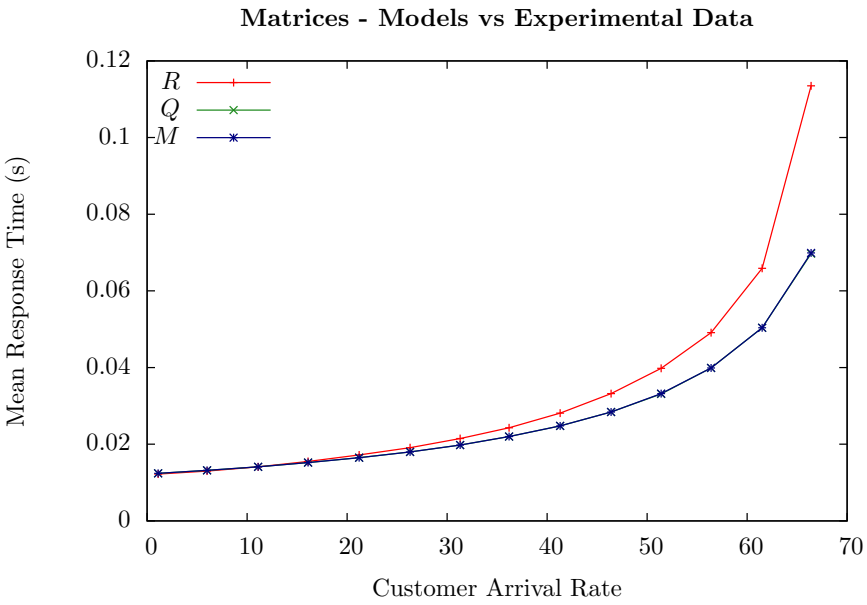

Figure 6.7: Comparison between experimental data R, predictions Q and M.

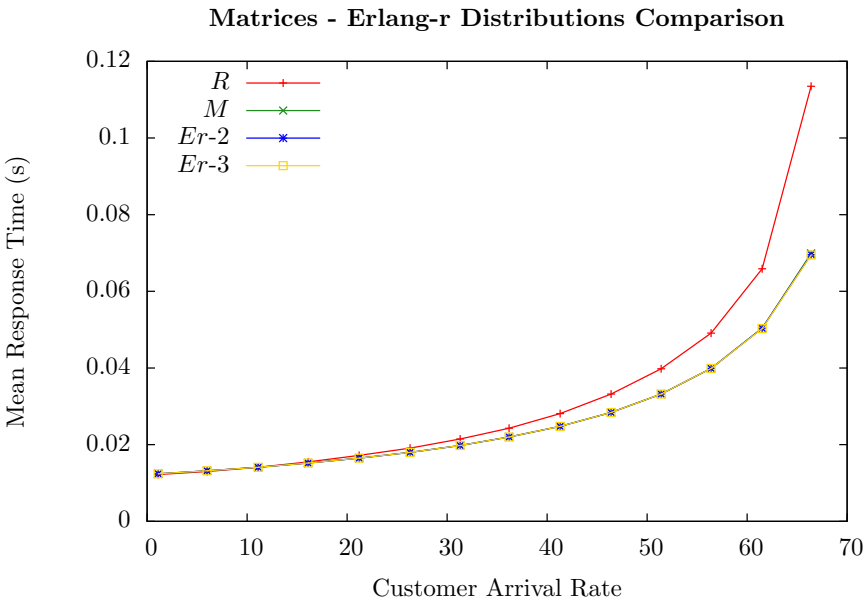

Figure 6.8: Comparison between experimental data R, prediction M and two Erlang-r distributions predictions.

Chapter 7

Conclusions

This thesis investigates the automatic memory management in modern programming languages focusing on the solutions and garbage collection policies adopted by HotSpot, the technology at the core of Java Virtual Machine. It statistically characterises the memory allocation requirements of some classes of applications, providing numerically tractable models to predict some performance indices of the system, i.e., throughput and average response time. These models are validated through a comparison with experimental results.

7.1 Contributions

After recalling the research effort during the years to quantifying the performances of garbage collection techniques (Chapter 1) and introducing the schemes at the base of the algorithms implemented by the languages like Java (Chapter 2), are given the theoretical concepts to understand the analytical models proposed (Chapter 3). QBD Model for GC (Chapter 4) and the new Markov-Modulated Queuing Model for GC (Chapter 5) proposed, have both an underlying Markov chain with the particular structure of Quasi-Birth and Death process. This peculiarity allows the application of the matrix-geometric method to the models to find the numerical solutions of the Markov chains. The solutions are numerically tractable problems and can be computed with automatic tools, which provide the steady-state distribution. The queueing models are then used to provide performance indices of the garbage collection policy analysed. More specifically, the steady-state distribution is used to derive the mean number of customers and the mean response time of a system which adopts the *stop-the-world* policy.

Finally, the queueing models for garbage collection are parametrized and validated through some applications that resemble a real scenario (Chapter 6). Magnolia CMS is an open-source content management system used to simulate a web application and Matrices is a servlet specifically designed to observe the behaviour of the system with intensive memory load conditions. For the purpose some tools have been developed to collect and analyse the measurements on a dedicated testing environment. Several system statistics have been collected, which included the frequency of garbage collections, the time taken to carried out the operations, the time required to service customers and the status of the memory. The mean response time prediction provided by the two models is validated comparing values with experimental results and providing the prediction error.

7.2 Results and Future Works

Results are quite good with moderate workload, then the system boundary become noticeable and the models prediction tend to diverge from the experimental results obtained with the JVM activation policy. Comparing the two queueing models, the QBD Model for GC on some cases fit better than Markov-Modulated Queuing Model for GC. This is not surprisingly since it involves more parameters and takes advantage from a more accuracy. However, also results for our simpler model are acceptable and they overlapping with the other predictions, especially for moderate workload.

Since the algorithms used by garbage collectors are usually CPU-intense, they cause a high consumption of CPU-cycles and a consequent waste of energy. This can be an issue for mobile devices relying on batteries. Therefore, future research effort should extend the models to predict the performances of a garbage collection policy also in terms of expected CPU energy consumption. Other efforts should extend the models validation to more complex architecture which make use of parallel and concurrency collectors.

Bibliography

[1] S. Balsamo, G.-L. D. Rossi, and A. Marin. Optimisation of virtual machine garbage collection policies. In *Proceedings of the 18th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, ASMTA'11, pages 70–84, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] D. Bini and B. Meini. On cyclic reduction applied to a class of toeplitz-like matrices arising in queueing problems. In W. Stewart, editor, *Computations with Markov Chains*, pages 21–38. Springer US, 1995.

[3] D. A. Bini, B. Meini, S. Steffé, and B. Van Houdt. Structured markov chains solver: Software tools. In *Proceeding from the 2006 Workshop on Tools for Solving Structured Markov Chains*, SMCtools '06, New York, NY, USA, 2006. ACM.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM.

[5] D. Buytaert, K. Venstermans, L. Eeckhout, and K. Bosschere. Transactions on high-performance embedded architectures and compilers i. chapter GCH: Hints for Triggering Garbage Collections, pages 74–94. Springer-Verlag, Berlin, Heidelberg, 2007.

[6] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[7] A. Diwan, D. Tarditi, and E. Moss. Memory system performance of programs with intensive heap allocation. *ACM Trans. Comput. Syst.*, 13(3):244–273, Aug. 1995.

[8] M. Hertz and E. D. Berger. Automatic vs. explicit memory management: Settling the performance debate. Technical report, Citeseer, 2004.

[9] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM.

[10] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011.

[11] J. P. Kharoufeh. *Level-Dependent Quasi-Birth-and-Death Processes.* Wiley Encyclopedia of Operations Research and Management Science, 2011.

[12] G. Latouche and Y. Ramaswami. A logaritmic reduction algorithm for Quasi Birth and Death processes. *J. of Appl. Prob.*, 30:650–674, 1994.

[13] D. Mosberger and T. Jin. Httperf: a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, Dec. 1998.

[14] M. Neuts. *Matrix-geometric Solutions in Stochastic Models: An Algorithmic Approach.* Dover books on advanced mathematics. Dover Publications, 1981.

[15] M. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications.* Probability: Pure and Applied. Taylor & Francis, 1989.

[16] W. J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling.* Princeton University Press, Princeton, NJ, USA, 2009.

[17] Sun Microsystems, Inc. Memory management in the Java HotSpot™ Virtual Machine, 2006.

[18] P. Van Mieghem. *Performance Analysis of Communications Networks and Systems*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.