Washington University in St. Louis
Washington University Open Scholarship

McKelvey School of Engineering Theses & Dissertations

McKelvey School of Engineering

Spring 5-15-2021

Real-Time Virtualization and Coordination for Edge Computing

Haoran Li
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

 Part of the Computer Sciences Commons

Recommended Citation

Li, Haoran, "Real-Time Virtualization and Coordination for Edge Computing" (2021). *McKelvey School of Engineering Theses & Dissertations*. 624.
https://openscholarship.wustl.edu/eng_etds/624

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

Washington University in St. Louis

McKelvey School of Engineering

Department of Computer Science and Engineering

Dissertation Examination Committee:
Chenyang Lu, Chair
Sanjoy Baruah
Christopher D. Gill
Linh Thi Xuan Phan
Ning Zhang

Real-Time Virtualization and Coordination for Edge Computing

by

Haoran Li

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2021
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

My greatest thanks go to my advisor: Prof. Chenyang Lu. I am more than grateful to him for granting me such a wonderful research journey — the Ph.D. program in Washington University in St. Louis. Prof. Lu has always been amazing advisor, who introduced me to the real-time area, encouraged me to become a systems researcher, inspired me with invaluable insights, and guided me in finishing this dissertation. Without him, none of the work described in this dissertation would have been possible. His enthusiasm for exploring the unlimited possibilities in research has encouraged me to discover the insights presented in this dissertation.

I would like to thank my colleagues, Meng Xu and Chong Li, who provided invaluable input and contributions to this work. Every research meeting with you in particular brought new ideas and knowledge to me.

I would also like to thank my other committee members: Prof. Christopher Gill, who helped me polish every single of my paper. Prof. Sanjoy Baruah, who taught me fundamental real-time scheduling theory which served an important role in my M/D(DS)/1 queueing model. Prof. Ning Zhang, who revealed the potential security challenges in real-time virtualization to me via the RT-TEE project. Prof. Linh Phan, who actively provided theoretical supports on both the M2-Xen and VAS projects. I would like to thank all the great members in Cyber-Physical Systems Lab, Dr. Jing Li, Dr. Yehan Ma, Ruixuan Dai, Dingwen Li and many more for the invaluable discussion we have on every seminar session and lab meeting. I

will miss the time we spent on polishing presentations and papers, and on discussing research ideas. I would like to thank my family for their encouragement and unconditional love and support.

Haoran Li

Washington University in Saint Louis
May 2021

To my wife, Yao.

ABSTRACT OF THE DISSERTATION

Real-Time Virtualization and Coordination for Edge Computing

by

Haoran Li

Doctor of Philosophy in Computer Science

Washington University in St. Louis, May 2021

Research Advisor: Professor Chenyang Lu

Recent years have witnessed the emergence of edge computing as an enabling platform for time-sensitive services. However, existing edge computing platforms face a multitude of challenges in meeting the latency requirements of time-sensitive applications. (1) Traditional real-time virtualization platforms require offline configuration of the scheduling parameters of virtual machines (VMs) based on their worst-case workloads. However, this static approach results in pessimistic resource allocation when the workloads in the VMs change dynamically. (2) Edge computing operators must deliver consistent tail latency performance for time-sensitive applications deployed on different edge sites. Traditionally, significant effort is required to test, tune, and configure a time-sensitive service for each edge host. This is a labor-intensive process that cannot scale effectively for a large number of edge sites. (3) Many time-sensitive services need to handle aperiodic requests for stochastic arrival processes, which differs from periodic and sporadic models in traditional real-time systems. It is necessary to provide a new latency analysis for predicting the latency distributions of those

aperiodic requests on an edge environment. (4) Fault-tolerant coordination for maintaining consistency in distributed applications. However, traditional failure recovery approaches employed by coordination services incur excessive recovery latency unacceptable to time-sensitive applications.

This dissertation makes the following contributions to the field of real-time edge computing through the design, implementation, and experimentation of four novel system technologies and architectures.

- **Multi-Mode-Xen (M2-Xen): Dynamic CPU Resource Provisioning.** M2-Xen enables dynamic resource allocation on a real-time virtualization platform, allowing VMs to switch modes with different CPU resource requirements at run-time while maintaining desired real-time performance.
- **Virtualization-Agnostic Scheduling (VAS): Scheduling for Consistent Latency.** VAS provides a novel scheduling framework to maintain similar latency distributions for time-sensitive tasks on different virtualized hosts.
- **Stochastic Response Time Analysis for Aperiodic Tasks on Virtualization Platforms.** To achieve predictable latency for aperiodic time-sensitive services, we establish a novel queueing model, M/D(DS)/1, for stochastic response time analysis of aperiodic services following a Poisson arrival process on computing platforms that schedule time-critical services as deferrable servers.

- **RT-ZooKeeper: Fast Recovery in Replicated Coordination Services.** RT-Zookeeper employs novel leader election and recovery protocols to address the limitations of Apache ZooKeeper, the prevailing open-source coordination service. Implemented based on ZooKeeper version 3.5.8, RT-ZooKeeper significantly reduces the latency in recovering from leader failures in replicated coordination services.

Chapter 1

Introduction

With the proliferation of the Internet of Things (IoT) [8], an increasing volume of data and processing accelerates the evolution of time-sensitive application towards embracing *edge clouds* [117] as a general computing infrastructure located close to the sources of data [59, 49]. For example, a recently released O-PAS standard[127] for *Open Process Automation System* by *The Open Group* addressed using on-premises edge cloud for executing controller (IEC 62264 Level 2 and 3 functions).

Edge computing brings about several major advantages for building modern time-sensitive applications:

- **Cost-Effective Computational Resource.** Instead of installing proprietary hardware, service providers can purchase relative low-cost switches, storage and servers to run virtual machines that perform computational and management services.
- **Uniform Hardware Architecture.** The uniform hardware architecture of general purpose servers alleviates the hassle caused by various types of subsystem. Traditionally, significant effort is required to test, tune, and configure an edge service for each edge cloud. This is a labor-intensive process that cannot scale effectively for a large

number of edge sites. A uniform hardware architecture brings about the possibility of an effective system management solution for time-sensitive workloads on edge clouds.

- **Flexibility.** The dynamic and elastic feature required in next-gen agile manufacturing can be delivered by software-defined solutions on edge. Besides, the abundance of open software and libraries enables fast prototyping and evaluation.

While consolidating services within edge hosts makes systems much flexible and cost-effective, services from different time-sensitive applications should be isolated from each other due to the security and privacy concerns. Similarly, due to the timeliness concerns, time-sensitive and non-time-sensitive ones requires performance isolation with each other. Thus, the computing platform needs to provide *run-time isolation* and *performance isolation* among different services.

Fortunately, leveraging *virtualization technology* on the edge, we can address those isolation concerns. For run-time isolation, services from different time-sensitive applications can be partitioned into different virtual machines (VMs). For performance isolation, virtual CPUs from of virtual machines can be prioritized, throttled by scheduling servers, and used for differentiating time-sensitive and non-time-sensitive services.

1.1 Challenges

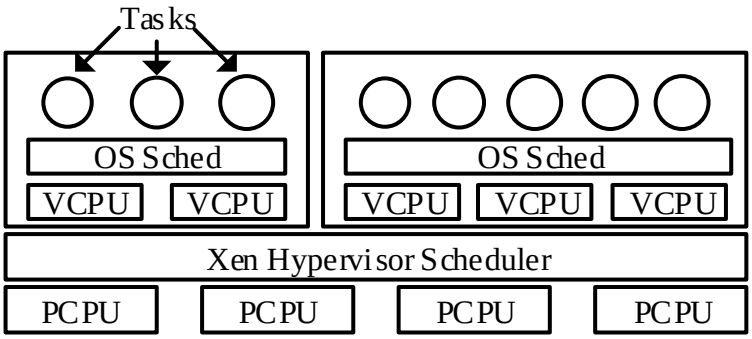


Figure 1.1: Scheduling Architecture of Xen

Virtualization introduces another layer of resource abstraction. A two-level hierarchical scheduling framework is used in most virtualization systems. A VM runs on virtual CPUs (VCPUs), which are scheduled on physical CPUs (PCPUs) by the hypervisor. Tasks in a VM are scheduled on VCPUs by the guest OS of the VM. Fig. 1.1 shows the hierarchical scheduling architecture of Xen hypervisor. In order to maintain the real-time performance of the service on edge, real-time resource provisioning involving VCPU scheduling should be addressed. Here are some challenges:

Dynamic Resource Provisioning. Edge clouds need to be reconfigured for agile manufacturing. Traditionally, computational resources for each VM are statically allocated through the hypervisor, in a configuration that is based on the worst-case CPU requirement of the VM. However, this static configuration approach may under-utilize hardware resources when VMs may operate in different modes each involving a different task set. Supporting multi-mode VMs becomes increasingly important as real-time systems operate in dynamic settings.

Achieving Virtualization Agnostic Latency. Another challenge faced by edge cloud operators is to manage numerous edge clouds in a scalable fashion. This challenge is compounded by the need to meet the service level objective of time-sensitive applications deployed on different edge clouds. Traditionally, significant effort is required to test, tune, and configure an edge service for each edge cloud. This is a labor-intensive process that cannot scale effectively for a large number of edge sites. It is thus necessary to provide a simple and effective system management solution for time-sensitive workloads on edge clouds. To simplify edge cloud management, we advocate *virtualization-agnostic latency (VAL)* as a desirable property to maintain consistent latency across different edge clouds. VAL requires that an application experiences similar latency distributions on a shared virtualized host as on a dedicated one. With VAL the same time-sensitive application can be deployed on multiple edge clouds with similar latency distribution, thereby greatly reducing the effort needed for performance testing and tuning on individual edge cloud.

Response Time Analysis for Aperiodic Workload. The confluence of stochastic latency requirements, aperiodic arrivals, and performance isolation mechanisms makes it highly challenging to analyze the response time of a time-critical service. The stochastic arrivals and the resultant queueing delays make response time analysis non-trivial, even for the highest-priority task in fixed-priority scheduling. The enforcement of CPU budgets for performance isolation further aggravates the complexity of the analysis.

Real-Time and Fault-tolerant Coordination. Consolidating services into edge hosts introduces a crucial challenge, especially for distributed time-sensitive applications that need to be highly reliable. The application to be migrated to edge cloud, need to be designed as a distributed application with fault tolerance feature, in case of an occasional physical edge host failure. It is complicated to design and implement a distributed application. Moreover,

the recovery procedure of such a service needs to meet certain real-time requirements. To achieve the timeliness of distribution applications on edge cloud, we propose to develop a fault-tolerant and real-time coordination service on edge clouds.

1.2 Contributions

Our major contributions and proposed works address dynamic resource provisioning architecture, resource configuration for periodic and stochastic time-sensitive tasks, and recovery-time aware real-time data store service:

- We developed *Multi-Mode-Xen*, a real-time virtualization platform that adapt to mode changes of VMs, maintaining real-time performance of dynamic systems.
- We introduced *virtualization-agnostic scheduling (VAS)*, a practical CPU scheduling framework for time-sensitive applications on shared virtualization hosts. For periodic and sporadic tasks, we established theoretical guarantees that time-sensitive tasks within a partial CPU can achieve the same task schedules as those on a full CPU, and thereby achieve the same latency distribution. Moreover, we also prove that the minimal (optimal) CPU bandwidth can be achieved while maintaining VAL.
- We established and analyse a novel queueing model, M/D(DS)/1, for an aperiodic (more generally, stochastic) task in a real-time virtualization system. We develop a numerical method for computing the stationary response time distribution of a stochastic Poisson arrival process scheduled by a deferrable server.
- We developed a real-time fault-tolerant coordination service – *RT-ZooKeeper* – for simplifying distributed real-time application design on edge cloud. RT-ZooKeeper

features a predictable and fast service recovery by incorporating several innovated protocols.

1.3 State-of-the-Art

1.3.1 Support Dynamic Resource Configuration

In many real-time virtualization systems, such as RT-Xen [138], vMPCP [65] in KVM [66], Quest-V[81], a two-level hierarchical scheduling model has been adopt. VCPUs are scheduled on PCPUs by the hypervisor. Tasks are scheduled on VCPUs by the guest OS of the VM. A VCPU's *resource interface* is represented as $V_i = (B_i, P_i)$, where P_i is the *period* and B_i is the *budget*, indicating that the VCPU is guaranteed to run for B_i time units in every interval P_i when schedulable.

To meet the desired real-time performance of the given tasks in a VM, we may calculate its resource interface based on *Compositional Scheduling Analysis (CSA)* [34, 12, 118, 140]. Given (1) the periods and the *worst-case execution times (WCET)* of a set of periodic tasks and (2) the real-time scheduling policy of the guest OS, we can compute the resource interface of each VCPU and how many PCPUs we need to satisfy the CPU resource requirement of the given system. If a host have enough CPU capacity to accommodate the resource interfaces, the task system is schedulable, otherwise we cannot guarantee the schedulability.

Traditionally, each VCPU within a VM has a *static* resource interface, which must be configured when the system is initialized. If the VM may change its workloads or real-time requirements at run time, the designer has to calculate the resource interfaces based on the

worst-case scenarios. Such a static configuration can result in significant resource under-utilization at run time except when all VMs experience their worst-case scenarios at the same time.

A dynamic mechanism for re-configuring resource interface at run-time is needed to mitigate the pessimistic configuration methodology and to avoid severe resource under-utilization in circumstances where dynamic workload applies.

1.3.2 Virtualization Agnostic Scheduling

To simplify edge cloud management, we advocate *virtualization-agnostic latency (VAL)* as a desirable property to maintain consistent latency across different edge clouds. VAL requires that an application experiences similar latency distributions on a shared virtualized host as on a dedicated one. With VAL a same time-sensitive application can be deployed on multiple edge clouds with similar latency distribution, thereby greatly reducing the effort needed for performance testing and tuning on individual edge cloud.

The state of the art in real-time scheduling provides two alternative approaches to support time-sensitive workloads on a virtualized platform. Current cloud infrastructure typically achieve VAL by dedicating physical CPUs (PCPUs) to time-sensitive services, i.e., they allow time-sensitive services to monopolize *full CPUs*. For example, Heracles [87] isolates cores by using `cgroup` features to achieve a specific latency Service Level Objective (SLO). VMware also supports time-sensitive applications by dedicating PCPUs to them [86], as does RedHat in the real-time KVM project [53]. For time-sensitive cloud services like Redis, it has been suggested [85, 89] to set the CPU affinity and use a dedicated core to maximize

performance. While delivering VAL, dedicating PCPUs to VCPUs can potentially incur resource overprovisioning, which is undesirable for resource-constrained edge platforms.

In contrast to the full CPU approach, the real-time systems community developed compositional scheduling frameworks and real-time virtualization technologies that allow multiple real-time VMs to share a PCPU, thereby achieving real-time performance on *partial CPUs* [65, 60, 145, 138, 81]. However, existing compositional scheduling approaches are geared toward meeting deadlines for hard real-time systems, and cannot be directly applied to cases that require SLO in term of tail latency. Furthermore, allocating CPU resources based on existing compositional scheduling analyses involving different scheduling server policies [121, 34, 12, 118, 16, 15] may lead to low resource utilization due to pessimism in providing real-time guarantees in a hierarchical manner. Finally, configuring the scheduling parameters of a VM based on compositional scheduling analyses may involve significant computation and complexity, which complicates edge cloud management.

To achieve VAL in a resource-efficient manner, we propose *virtualization-agnostic scheduling (VAS)* that allow time-sensitive applications to maintain its latency distribution on a shared CPU, while allowing general-purpose applications to reclaim unused CPU cycles in a virtualized host. As a first step towards enforcing virtualization-agnostic latency, we focus on CPU scheduling first. This work does not address performance interference caused by other shared resources such as cache, memory, and I/O subsystems. To develop a complete system solution that is fully agnostic to virtualization, future work is needed to manage other resources as well.

Despite the considerable progress in real-time scheduling for virtualized platforms, none of the prior work was geared toward achieving virtualization-agnostic latency. VAS is, to our

knowledge, the first scheduling framework to provide virtualization-agnostic latency for time-sensitive applications on partial CPUs. While traditional real-time virtualization is designed to meet task deadlines, VAS can preserve the same task schedules as those on dedicated host. Besides, in comparison to traditional scheduling approaches adopted by earlier real-time virtualization, VAS is tailored for more restrictive (but common) use cases. While traditional real-time scheduling supports any number of real-time VCPUs in a host (subject to schedulability), VAS can support only one time-sensitive VCPU per PCPU, and up to M VCPUs on an M-core processor. Nevertheless, VAS is suitable for edge platforms on which the workload is often dominated by non-real-time VMs, but it is important to meet the latency requirements of a small number of latency-sensitive VMs. In addition, while compositional scheduling analysis (CSA) cannot always fully utilize the PCPUs due to the pessimism in providing schedulability guarantees in a hierarchical manner, VAS allows a VMs to fully utilize a PCPU when needed. Finally, CSA is usually complex and time-consuming, but VAS allows a VCPU's resource interface to be computed efficiently, which simplifies edge cloud management.

1.3.3 Response Time Analysis for Aperiodic Tasks

Time-critical services are increasingly hosted in the cloud to take advantage of the flexibility and scalability of cloud computing platforms. Examples include streaming analytics [143], interactive services [80], and in-memory data stores [111]. Cloud providers face a tremendous challenge to meet the latency requirements of time-critical services. In contrast to traditional real-time systems, the service-level objective (SLO) of a time-critical service is usually in terms of a target *tail latency* instead of hard deadlines. Moreover, requests for cloud services may arrive *aperiodically* following stochastic arrival processes that depart from periodic or

sporadic task models. Finally, it is essential to enforce *performance isolation* between cloud services in a shared, multi-tenant environment. A service (or micro-service [99]) may be instantiated at different granularities, ranging from a process, to a container, to a virtual machine. For CPU resources, which are the focus, a common approach to performance isolation is to schedule a service as a *scheduling server* that is allowed to execute for a specified *budget* within each *period*. As examples, Quest-V [81] uses the PIBS server for micro-service processes, whereas a Linux process, when scheduled by `SCHED_DEADLINE` [77], is governed by a constant bandwidth server (CBS), and a virtual CPU (VCPU) is scheduled as a deferrable server (DS) when scheduled by the Xen RTDS [138, 139] scheduler.

The confluence of stochastic latency requirements, aperiodic arrivals, and performance isolation mechanisms makes it highly challenging to analyze the response time of a time-critical service. The stochastic arrivals and the resultant queueing delays make response time analysis non-trivial, even for the highest-priority task in fixed-priority scheduling. The enforcement of CPU budgets for performance isolation further aggravates the complexity of the analysis.

Traditionally, response time analysis in real-time systems focuses on bounding the worst-case response time. This approach does not work for time-critical services with aperiodic stochastic arrivals and tail latency requirements. Due to the stochastic arrivals of service requests, the worst-case response time cannot be bounded. While soft real-time scheduling approaches are geared towards achieving bounded tardiness [96, 24, 37], they focus on periodic tasks. Similarly, earlier efforts on stochastic analysis have been proposed for periodic tasks [126, 5, 32, 88, 94, 105]. Moreover, existing aperiodic scheduling approaches based on scheduling servers [124, 125, 83, 58] and aperiodic utilization bounds[1] are not designed to provide offline guarantees for aperiodic tasks. Hierarchical and compositional scheduling

analysis [120, 34, 12, 118] leverages scheduling servers to achieve performance isolation, but the analysis approach is generally designed for periodic tasks and deadlines instead of tail latency.

Queueing theory provides tools to derive the response time distribution subject to stochastic arrivals. Real-time calculus [128, 107] and real-time queueing theory [75, 76] extend probabilistic queueing theory to derive hard upper and lower performance bounds instead of tail latency. Furthermore, while traditional queueing theory usually models an always-on server, a scheduling server algorithm (e.g., deferrable server) dictates that the server is no longer always active, i.e., the server is suspended when it runs out of its budget. It is therefore necessary to extend queueing theory to model and analyze scheduling servers.

We a first step towards response time analysis for time-critical services. We consider a time-critical service scheduled as a deferrable server for performance isolation, and develop a numerical method for computing the stationary[1] response time distribution of a stochastic Poisson arrival process. From a queueing theory perspective, we denote the system as M/D(DS)/1. This notation extends the established M/D/1 queue (in Kendall's notation [63][2]), to highlight that the server model is changed from an always-active server to a deferrable server (DS).

We first observe that a scheduling server in real-time scheduling theory can be modeled as a periodic service queueing model that has been studied in the queueing theory literature [35, 103, 104, 115, 100]. Combining scheduling analysis and virtual waiting time analysis, we then establish a transformation between periodic and deferrable servers. Finally, we derive

[1] A stationary distribution of a Markov chain is a probabilistic distribution that remains unchanged in the Markov chain as time progresses [19].

[2] In Kendall's Notation, "M" is for Poisson arrival, indicating the feature of "Memory-less"; "D" is for constant service time, i.e., "Deterministic".

an efficient method to compute the stationary response time distributions of M/D(DS)/1 models. We demonstrate how the response time distribution of an M/D(DS)/1 system enables us to configure a deferrable server to achieve the desired tail latency for a time-critical service. We implement our approach in a virtualization platform based on Xen 4.10 and evaluate two case studies, one involving a synthetic service and another involving Redis, a common in-memory data storage service. The results of these studies demonstrate the validity and efficacy of our numerical approach in supporting time-critical services in a practical setting.

Kaczynski, Lo Bello, and Nolte [58] extended the SAF model [32] by allowing aperiodic tasks to run within polling servers. The objectives of our work and their work are different. On one hand, the extended SAF model was not designed for deriving the exact response time distribution of aperiodic tasks, which is the main objective of our work. On the other hand, another aspect of the extended SAF model is more general: It allows arbitrary arriving and arbitrary execution for aperiodic task, by using *Arrival Profile* and *Execution Time Profile (ETP)*, and running them within a polling server with any priority, while our system allows the aperiodic task to be a Poisson arrival with deterministic execution, running within a highest priority deferrable server. Unfortunately, given the difference between the deferrable server and the polling server, the ETP extraction cannot be directly adopted on a deferrable server.

Queueing systems with periodic service have been studied since 1956 [100]. Researchers encountered mathematical difficulties when trying purely analytical techniques to study virtual waiting time and response time distributions on continuous time domain models [115, 104, 103]. Eenige [35] focused on discrete time queueing systems with periodic services. He

observed the mathematical difficulties in deriving a pure analytical method even on a simplistic B/D(PS)/1 queue whose service time equals one time slot, i.e., $d = 1$. As a result, Eenige employed numerical methods for calculating the virtual waiting time distribution for a discrete queueing system with periodic service. Inspired by the queueing model with periodic service and virtual waiting time analysis, here we first observe the virtual waiting time equivalence between the periodic server and the deferrable server. Combining scheduling analysis and virtual waiting time analysis, we derive an efficient method to compute the stationary response time distributions of M/D(DS)/1 models.

1.3.4 Real-time Fault-Tolerant Coordination Service

Distributed applications require different forms of coordination services (e.g., leader election, naming, global configuration, group membership, and synchronization). However, it is tedious and error-prone to design and implement such coordination features from scratch for a distributed application, and so reusable fault-tolerant coordination services, like ZooKeeper [51], have been widely used to develop distributed applications and services in cloud environments [113]: e.g., Kafka [68] leverages ZooKeeper for for topic creation, electing a broker leader and mapping topic partition pairs, monitoring topology changes; and Hadoop [122], Hive [129], Flink [25], and Storm [131] use ZooKeeper for group management, naming, and global configuration.

Real-time distributed applications are often deployed in edge computing environments [117, 59, 49], which introduces both challenges and opportunities to coordination services: coordination services must recover from failures in a timely manner [46, 39, 144], while edge

computing platforms provide bounded communication delays that can be exploited to reduce recovery latency [43, 84, 74, 2].

Apache Zookeeper, as the prevailing open-source coordination service, comprises a group of replicated servers to provide fault-tolerate coordination services. It relies on ZooKeeper Atomic Broadcast protocol (ZAB) [57], a state of the art distributed protocol for synchronizing replicas against the leader, managing database update transactions, and recovering from a crashed state to a valid state. Though ZooKeeper offers fault-tolerance, its service recovery time on a leader failure can be excessive for many time-sensitive distributed applications, hindering the real-time performance of applications that rely on ZooKeeper. Moreover, there is a lack of recovery time analysis to characterize the duration of the unavailability of the ZooKeeper service when a leader failure occurs.

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we present Multi-Mode-Xen, which focuses on dynamic resource provisioning. Chapter 3 is for VAS, while Chapter 4 leverages the similar scheduling framework for establishing a novel queueing model, M/D(DS)/1. Chapter 5 presents RT-ZooKeeper, the real-time fault-tolerance coordination service features fast recovery. Finally, Chapter 6 summarizes our results, raises open questions, and discusses future work.

Chapter 2

Multi-Mode-Xen: A Dynamic Resource Provisioning System

In this chapter, we present Multi-Mode-Xen (M2-Xen), a real-time virtualization platform that adapt to mode changes of VMs, maintaining real-time performance of dynamic systems. M2-Xen is built on top of RT-Xen [137, 138], a real-time scheduling framework for the Xen hypervisor [11]. In Section 2.1, we review the Xen virtualization architecture, addressing the limitations of its resource provisioning framework. In Section 2.2, we formalize the problem and present the system model. Thereafter, in Section 2.3, we propose our the design and implementation of M2-Xen to support dynamic resource reallocation among system modes. To speed up the mode switch procedure, we explore several strategies including pure use-level solutions and a hypervisor-assist solution, which are stated in Section 2.4. In Section 2.5, we demonstrate the performance of M2-Xen.

2.1 Background

2.1.1 Xen

Xen [11] is a widely used open-source hypervisor. A Xen-based virtualization system includes a privileged administration VM (i.e., Domain 0) and multiple unprivileged VMs (i.e., guest domains). The privileged VM is used by system operators to manage the unprivileged VMs. Each VM has its own operating system that schedules its tasks.

A two-level hierarchical scheduling framework is used in a virtualized host based on Xen, as in most virtualization systems. A VM runs on multiple *virtual CPUs (VCPUs)*, which are scheduled on multiple *physical CPUs (PCPUs)* by the hypervisor. Tasks in a VM are scheduled on VCPUs by the guest OS of the VM.

2.1.2 RT-Xen and Its Limitations

The RT-Xen project [137, 138] developed a suite of real-time schedulers for the Xen hypervisor, including the RTDS scheduler [138] that was adopted in Xen in 2015. When the RTDS scheduler is used in the Xen hypervisor, the user must specify the *resource interface* of each VM. A resource interface $\omega = (N, B, P)$ specifies that the VM has N VCPUs, and that each VCPU should be executed no more than $B\mu s$ (budget) every $P\mu s$ (period).

To guarantee the budget and period assigned to each VCPU, RTDS treats each VCPU as a deferrable server, where the budget of the VCPU is consumed only when a task is running on the VCPU. RTDS schedules all VCPUs based on the global Earliest Deadline First (gEDF) policy.

To meet the desired real-time performance of the tasks in a VM, we may calculate its resource interface ω based on *Compositional Scheduling Analysis (CSA)* [34, 12]. Given (1) the periods and the *worst-case execution times (WCET)* of a set of periodic tasks and (2) the real-time scheduling policy of the guest OS, we can compute the resource interface ω of the VM using CSA (e.g., the Multi-processor Periodic Model [118]). If the underlying hypervisor scheduler (RTDS) satisfies the resource interface of the VM at run time, the tasks in this VM are expected to meet their deadlines [140]. If a host does not have enough CPU capacity to accommodate the resource interfaces of all VMs, it cannot guarantee the schedulability of the tasks based on CSA.

It is important to note that RT-Xen supports a *static* resource interface per VM, which must be configured when the VM is initialized. If the VM may change its workloads or real-time requirements at run time, the designer has to calculate the resource interfaces based on the *worst-case* scenarios. Such a static configuration can result in significant resource underutilization at run time except when all VMs experience their worst-case scenarios at the same time. In contrast, M2-Xen supports dynamic VMs through on-line reallocation of CPU resources based on *multi-mode* resource interfaces.

2.2 Problem Statement

A *multi-mode* virtualization system may operate in different *modes* at run-time. Each VM can have different sets of soft real-time tasks in different modes. The task set of V_k in mode m is $S_{k,m} = \{\tau_{k,m,i}\}$. A dynamic real-time system may switch its mode (e.g., triggered by a user command or external events) at run time. When the system changes from mode m to mode m', each VM will switch from its task set in mode m to that in mode m'. Each

periodic task $\tau_{k,m,i}$ is characterized by $(C_{k,m,i}, T_{k,m,i})$, where C and T are WCET and period, respectively. The *utilization* of a task set $S_{k,m}$ is defined as $U_{k,m} = \sum_i \frac{C_{k,m,i}}{T_{k,m,i}}$.

For each VM, M2-Xen allows a user to specify multiple resource interfaces corresponding to its different modes. Specifically, V_k runs on N_k VCPUs; each VCPU of V_k has a budget $B_{k,m}$ and a period $P_{k,m}$ in mode m. The resource interfaces of V_k in mode m is specified as $\omega_{k,m} = (N_k, B_{k,m}, P_{k,m})$. The *bandwidth* of resource interface $\omega_{k,m}$ is defined as $W_{k,m} = N_k \times \frac{B_{k,m}}{P_{k,m}}$. Given a task set $S_{k,m}$ in mode m, VM V_k, and the real-time scheduling policy of the guest OS, we can use CSA to calculate the resource interface $\omega_{k,m}$.

A multi-mode virtualization system must meet the following requirements in response to a mode change.

- **Dynamic resource reallocation:** The system should dynamically change the resource interface and task set of each VM. When the system's mode changes from m to m', for each VM V_k, the system should change the resource interface and task set, from $\omega_{k,m}$ and $S_{k,m}$, to $\omega_{k,m'}$ and $S_{k,m'}$, respectively. The change in the resource interface consequently causes the underlying RTDS scheduler to reallocate CPU resources among the VMs.

- **Overload avoidance in VMs:** The system must avoid overloading a VM, i.e., the utilization of the task set and the mode management system in a VM cannot exceed the bandwidth of its resource interface. An overload in a VM can lead to deadline misses and even a kernel panic in the guest OS, when it runs out of CPU cycles allocated by the hypervisor.

- **Overload avoidance in the physical host:** The total bandwidth of all the VMs should not exceed the total number of PCPUs. If the total bandwidth exceeds the

CPU capacity of a host, some VMs will not receive their required CPU bandwidth, which may lead to deadline misses and even a kernel panic.

- **Fast mode switching:** The system needs to complete the mode switch quickly so that applications can start operating in the new mode without significant delays [3].

2.3 M2-Xen Architecture

In this section we first describe the main components of M2-Xen. We then explain the mode switching process, followed by our design and implementation choices.

2.3.1 System Architecture

Fig. 2.1 illustrates the system architecture of M2-Xen. M2-Xen comprises multiple VMs running on the Xen hypervisor in a multi-core host. The administrative VM, Domain 0, is pinned on a dedicated PCPU, PCPU 0. Each VM comprises soft real-time tasks running on a guest OS and a set of VCPUs.

The system administrator specifies a multi-mode system in a *Global Mode Table*. A *global mode* specifies a *local mode* for each VM, where each local mode is associated with a task set and the corresponding resource interface for the VM. Given a global mode m, we can find the task set $S_{k,m}$ and the resource interface $\omega_{k,m}$ of the VM k.

[3] While M2-Xen can meet the real-time performance requirements in different modes in steady state, it does not guarantee system schedulability during a mode change. M2-Xen is suitable for applications that can tolerate potential deadline misses during the transient mode switching process. However, for hard real-time systems, we plan to explore using mode switching protocols [73, 29, 22, 45, 114, 50, 110] in the future to avoid deadline misses during mode switching.

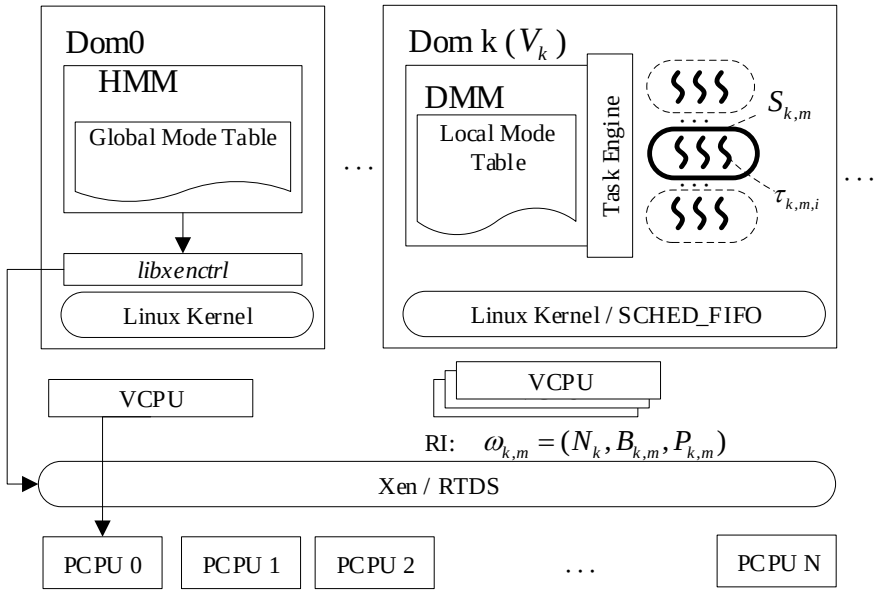


Figure 2.1: Architecture of M2-XEN

M2-Xen manages mode switching through a centralized *Host Mode Manager (HMM)* in Domain 0 and a *Domain Mode Manager (DMM)* in each VM. The HMM schedules and coordinates the local mode changes of the VMs based on the Global Mode Table. During mode switch, the HMM sends commands to the DMMs of VMs, requesting them to change the task sets to those of the new mode. The HMM is also responsible for changing the resource interfaces of VMs (via the *libxenctrl* library provided by the Xen hypervisor).

The DMM in each VM has a local mode table, which maps each local mode to a set of tasks. The DMM uses a task engine to change the task set in response to a mode change.

The HMM and DMMs communicate through TCP sockets to coordinate the mode switching process. The HMM contains a TCP server to which DMMs connect. When initialized, each DMM establishes a long-lived TCP connection to the HMM.

As DMM consumes CPU cycles to change task sets and to communicate with HMM, we need to account for the overhead when we compute the resource interface of a VM. Specifically, we can model the DMM as a periodic task whose period is the minimum inter-arrival time of mode switch requests. The WCET of a local mode switch is 100 μs in our current implementation. In practice, the mode switch frequency is usually not high and a VM can account for its DMM overhead in its resource interface, which allows M2-Xen to avoid CPU overload at both the VM and host levels despite the DMM overhead.

M2-Xen currently provide the mechanisms to change the modes of individual VMs at run-time. To support dynamic mode changes of individual VMs, M2-Xen may employ an online admission controller to perform compositional schedulability analysis and schedule VM mode changes on demand. A potential challenge in online admission control is that its schedulability analysis needs to be efficient in comparison to the required mode switching latency. The current M2-Xen avoided this issue by supporting a known set of global modes that are schedulable based on offline compositional analysis. This approach is applicable if the set of (global) system modes are well defined at design time or when the number of VMs and modes are limited.

2.3.2 Mode Switching Procedure

The mode switching procedure works as follows:

Step 1. Mode changes may be triggered by a predefined set of events from an unprivileged VM. The DMM of the VM then sends a predefined request to the HMM to trigger a mode change. This request contains a new global mode identifier. Based on the Global Mode Table, the HMM identifies the VMs whose local modes need to be changed.

Step 2. The HMM schedules the mode changes of the VMs . The mode changes of different VMs may be scheduled to occur sequentially or in parallel. Importantly, to prevent *PCPU overload* the HMM must enforce the PCPU capacity constraint when scheduling the mode changes. That is, the total bandwidth of all the VCPUs should always remain below the total PCPU capacity throughout the mode switching process.

Step 3. To change the mode of a VM, if the bandwidth of the VM increases in the new mode, the HMM first changes the VM's resource interface before sending the mode change command to the VM. Otherwise, the HMM sends the mode change command to the DMM, waits for the completion acknowledgement from the VM, and changes the resource interface of the VM. Note that this policy prevents *VM overload* by ensuring that the bandwidth of the VM remains larger than the total utilization of its tasks during the mode change process.

Step 4. When a DMM receives a mode change command, the DMM uses the task engine to switch the old task set to that in the new mode. The DMM sends a completion acknowledgement to the HMM once it finishes changing the task set.

Step 5. The HMM continues to schedule mode changes of the remaining VMs until all the VMs have completed their mode changes.

In the current implementation, M2-Xen accepts a new mode change request only after finishing the current mode change. In the future, we can explore other policies, e.g., (1) queuing mode change requests and serve them in order, or (2) allowing a new mode change request to preempt the ongoing mode change.

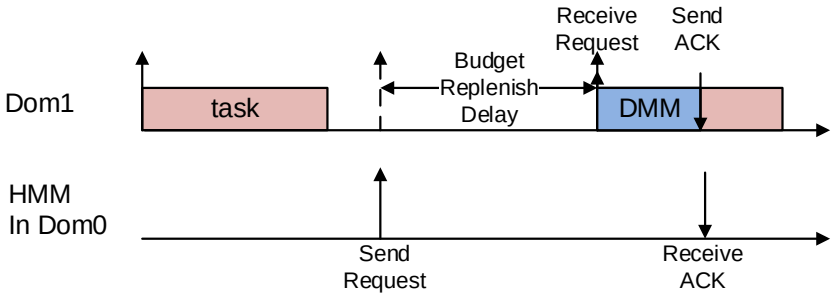
2.3.3 Implementation Issues

Configuration of Domain 0. In current implementation, M2-Xen allocates a full-capacity VCPU to Domain 0 and pins the VCPU onto a dedicated PCPU0. Domain 0 hosts both the HMM, which manages the mode changes for all VMs, and the inter-VM virtual network. Dedicating a PCPU to Domain 0 mitigates interference between Domain 0 and the applications running in the guest VMs.

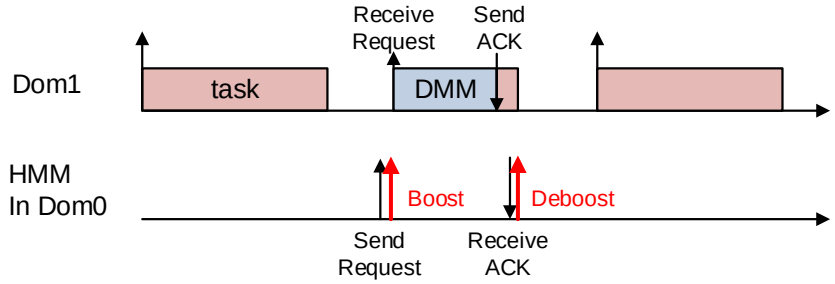
Changing the task set. Each DMM uses a task engine to change its local task sets. Our current implementation employs the signal mechanism in Linux. Given a task set identifier, the task engine sends SIGSTOP signals and SIGCONT signals to the tasks to be stopped and started, respectively. However, M2-Xen is not limited to a specific implementation of the task engine, and different VMs may adopt different approaches to change their task sets.

Inter-VM communication. We choose to use TCP connections for message exchanges between the HMM and the DMMs. Using TCP makes our implementation portable accross different operating systems and extensible to support mode switching in distributed systems in the future. As discussed in the next section, a challenge of TCP-based inter-VM communication is that its latency depends on VCPU scheduling. We present solutions to reduce inter-VM communication latency in the next section.

2.4 Reducing Mode Switching Latency



(a) Budget Replenishment



(b) Elimiate Budget Replenishment by Boosting VCPU

Figure 2.2: Typical Timelines of Sending Mode Switch Request

A key challenge to M2-Xen is to reduce the mode switch latency. Fast mode switch not only allows the system to enter the new node quickly, but also shortens the transient states in which tasks may potentially miss deadlines: the mode switching latency is largely attributed to communication latency between Domain 0 and the VMs, which can be heavily influenced by VCPU scheduling. When the VCPU of a VM runs out of budget and when a mode switch request arrives, the DMM received the request will not run until the VCPU's budget is replenished, a delay referred to as the *budget replenishment delay*, as shown in Fig. 2.2(a).

In this section, we frist analyse the delay experienced in a mode switching process, and then introduce approaches to reduce the delay through (1) user-level mode switch schedulings or (2) VM scheduling in the hypervisor.

The latency of changing a VM's mode comprises four parts: (1) the time required to send a local mode change request from the HMM to a DMM in a VM, (2) the overhead of changing the resource interface of a VM, (3) the latency of changing the task set of a VM, and (4) the latency of receiving a completion acknowledgement from the DMM. Note the above steps are ordered for the case when the bandwidth of a VM needs to be increased in order to avoid overload a VM. When the bandwidth of a VM needs to be reduced, the task set needs to be changed before the resource interface is reduced, as discussed in Section 2.3.

(1) The HMM sending a mode change request to a DMM. When a mode change starts, the HMM sends a mode change request message to each affected VM. The message includes a new resource interface and the local mode identifier. The latency of an inter-VM socket communication can be as small as several microseconds if the VCPU hosting the DMM, is running on a PCPU when the request message arrives. However, as shown in Fig. 2.2(a), if the DMM-VCPU has exhausted its budget and hence is not eligible to run, the DMM will not be able to receive a message from the HMM immediately. As a result, the message has to wait until the the DMM-VCPU's budget is replenished, as shown in Fig. 2.2(a). The budget replenishment delay significantly increase the latency of a mode change [4].

(2) Changing resource interface of the VM. The HMM uses *xl*, an administration tool, to change resource interfaces. Internally, the xl invokes a hypercall *Hypercall_DOMCTL*. The hypercall is wrapped into the *IOCTL* system call in Domain 0's kernel. Invoking such

[4]The communication delay induced by budget replenishment was also observed in Quest-V [81].

a hypercall will cause a context switch in the Linux Kernel and a VM-exit in Xen, which introduces overheads at the order of tens of microseconds.

(3) Changing task sets in VMs. Changing the task set to the one corresponding to the new mode can be implemented a domain-specific way. Our current implementation based on Linux *SIGNAL* takes no more than 100 μs to change a task set, as shown in Section 2.5.

(4) Receiving an acknowledgement from the DMM. As a PCPU is dedicated to Domain 0, there is no budget replenishment delay before the HMM receives the acknowledgement from the DMM.

As analyzed above, the mode switch latency can be dominated by the budget replenish delay of the VM. In the rest of this section, we first introduce three user-level mode switch scheduling methods to mitigate the mode switching delay and then present a technique to significantly reduce the budget replenish delay by modifying the hypervisor scheduler.

2.4.1 User-level Mode Switch Scheduling

To mitigate the impact of inter-VM communication delays, we propose three policies to schedule the mode change requests to different VMs.

Sequential. To avoid overloading the host during a mode transition, the HMM first sorts the mode change requests in increasing order of the difference between the new bandwidth and the old bandwidth of the VM. Then the HMM will change the mode of each VM one by one, starting from the VM that will incur the largest bandwidth decrease. The HMM does not send the next local mode change request until it has received the acknowledgement message from the previous VM.

Batch. The batch approach shortens the mode transition by allowing the HMM to issue mode switch requests to multiple VMs in parallel through non-blocking send over the TCP sockets. After collecting all the completion acknowledgements from those VMs, we then issue another batch of messages to VMs on the "increase" sublist. To avoid overloading in the physical host, the VMs in first batch that the HMM sends mode change requests are the ones whose bandwidthes are to be reduced.

Greedy. Like the batch approach, the Greedy approach also exploits concurrent requests to speed up the mode transition, but in a more aggresive manner. Rather than wait for *all* the acknowledgements from the "decrease" sublist, we can issue mode change requests to some of VMs on the "increase" sublist as soon as it can be performed without overloading the host. In order to avoid overloading the host, the HMM performs a bandwidth check every time we receive an completion acknowledgement from any VM whose bandwidth is decreased, and issue an increase message as soon as enough bandwidth becomes available.

2.4.2 Reduce Mode Switching Delay through VCPU Boost

While the user-level approaches described above can mitigate the impact of inter-VM communication delay through concurrent requests, they do not fundamentally reduce the inter-VM communication delay caused by the underlying VM scheduler. To this end, we introduce a *VCPU-boost* mechanism in the hpyervisor scheduler. The key idea of achieving fast message delivery is to temporarily boost the priority of the VCPU hosting the DMM of the receiving VM, when the HMM sends a mode switch request; and then return the DMM to its normal priority after the HMM receives the acknowledgement from the DMM. Following this

insight, we propose and implement VCPU Boost feature based on the RTDS scheduler in Xen hypervisor.

RTDS architecture. The RTDS scheduler in Xen 4.8 is event-driven. The scheduler is triggered by the following events: (1) the VCPU budget replenishment event, when a VCPU replenishes its budget; (2) the VCPU budget exhaustion event, when a VCPU runs out of its current budget; (3) the VCPU sleep event, when a VCPU has no task running; and (4) the VCPU wake-up event, when a task starts to run on a VCPU.

Whenever the RTDS scheduler is triggered, it applies the global EDF algorithm to schedule VCPUs: A VCPU with an earlier deadline has higher priority; at any scheduling point, the scheduler picks the highest priority VCPU to run on a feasible PCPU. A PCPU is feasible for a VCPU if the PCPU is idle or has a lower-priority VCPU running on it.

The RTDS scheduler uses two global queues to keep track of all VCPUs' runtime information. The first is a *run queue*, which has all runnable VCPUs sorted in increasing order of their priorities. A VCPU is runnable if the VCPU still has budget in the current period. The second is a *depleted queue*, which keeps all VCPUs that run out of budget in their current periods. The VCPUs in the *depleted queue* are sorted based on their release time, i.e., the next budget replenish time.

Design of the VCPU boost feature. We introduce the *boost* priority level to the RTDS scheduler. A VCPU with boost priority always has a higher priority than non-boosted VCPUs. All boosted VCPUs have the same priority and are scheduled based on the First Come First Served (FCFS) policy. The number of boosted VCPUs is no larger than the number of PCPUs, so that the boosted VCPUs can always be scheduled immediately.

We introduce two hypercalls for the administrative VM (Domain 0), to change a VCPU's boost status: (1) a *boost hypercall*, which promotes a VCPU's priority to boost priority, and (2) a *de-boost hypercall*, which degrades a boosted VCPU's priority back to non-boost priority. We allow only Domain 0 to issue these two commands by installing rules in the Xen Security Model (XSM). Note that it is important to restrict the boost hypercall to Domain 0 to prevent VMs from abusing the boost feature. As the boost mechanism effectively grants the boosted VCPU additional CPU time beyond its resource interface, it is important for Domain 0 to minimize the boost duration. As shown in our experimental results, the time needed for a mode change of a single VM is limited to within 200 μs. The impact of boost on the resource allocation is therefore negligible for many applications. Nevertheless, the boost feature limits M2-Xen to soft real-time applications because it affects the accuracy of CPU allocation to VCPUs.

The RTDS scheduler with the VCPU boost feature has two new scheduling events: (1) a *VCPU boost event*, when a VCPU's priority is promoted to the *boost* priority by Domain 0; (2) a *VCPU de-boost event*, when a boosted VCPU's priority is degraded to that of a non-boosted VCPU.

The scheduler is triggered by these two boost-related events: (1) At the *VCPU boost event*, the scheduler will find the PCPU the lowest priority, remove the boosted VCPU from the global queue (which can be the run queue or the depleted queue), and schedule the boosted VCPU on the PCPU. A PCPU's priority is equal to the current running VCPU's priority on the PCPU. A PCPU without any VCPU running on it always has lower priority than a PCPU with a VCPU running on it; (2) At the *VCPU de-boost event*, the scheduler will

update the de-boosted VCPU's deadline and its budget. The scheduler will insert the de-boosted VCPU back into the run queue if the de-boosted VCPU still has budget in the new period, or into the depleted queue otherwise.

Boost. We leverage the proposed VCPU boost feature to further reduce mode change latency. The HMM first sorts the mode change requests in ascending order of the difference between the new bandwidth and the old bandwidth of the VMs. Then the HMM changes the mode of each VM one by one, based on the sorted list of the mode change requests. After sending the mode switch request to a VM's DMM, the HMM immediately boosts the DMM-VCPU. Once the HMM receives the acknowledgement from the DMM, the HMM de-boosts the DMM-VCPU as illustrated in Fig. 2.2(b).

As a boosted VCPU receives extra CPU cycles, it is therefore necessary to reserve CPU bandwidth to accommondate the worst-case boost time in order to avoid host overload, i.e., the total bandwidth of all VCPUs should be no larger than the total number of PCPUs (as defined in Section 2.2). We can account for the additional bandwidth consumed by a boosted VCPU as a hypothetical VCPU. Based on the minimum inter-arrival time of mode switch request (the period) and the worst-case mode switch times of the VMs (the budget). To avoid host overload, we reserve the total bandwidth of the hypothetical VCPU of all the VMs. Furthermore, the HMM can use a timeout mechanism, to avoid a malfunctioning DMM from occupying a PCPU for an excessive amount of time. If the DMM fails to send an acknowledgement before the timeout, and hence enforcies an upper bound of the boost time of each VM.

2.4.3 Comparison of Different Approaches

As an example, Fig. 2.3 presents the mode switch timelines under different policies in real experiment. We used 6 VMs (exclude Dom 0) to share a 15-PCPU host. Each VM got 3 VCPUs. The timelines are plotted based on real experiment trace files. In this mode switch, Dom 1, 3, 4, and 5 decrease their bandwidth, while Dom 2 and 6 increase theirs. When

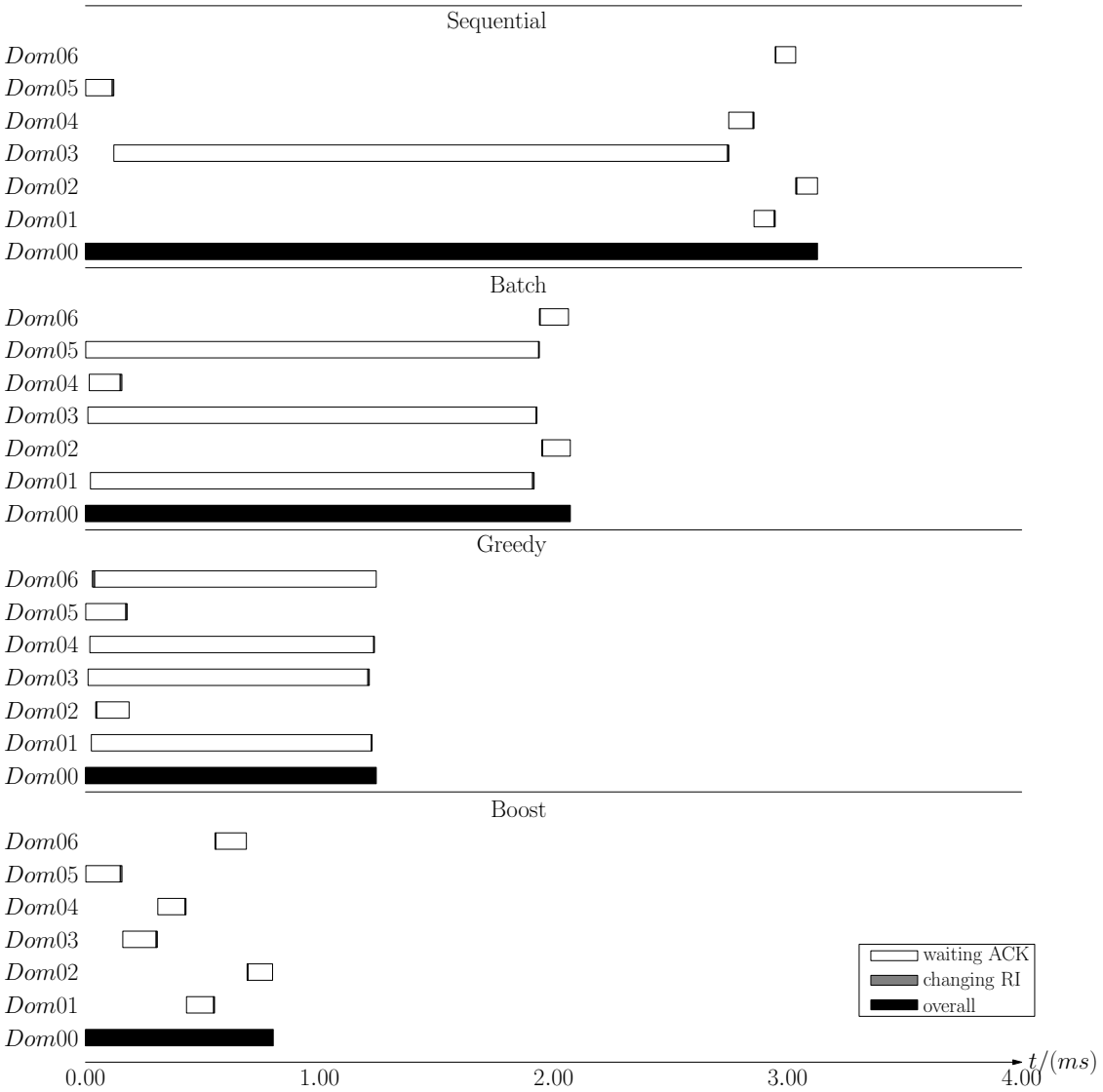


Figure 2.3: Example Mode Change Timelines: 6 3-VCPU VMs

decreasing a VM's bandwidth, the HMM sends a mode switch request to each DMM, receives the acknowledgement, then changes the resource interface. When increasing bandwidth, the resource interface should be changed before sending the request to the DMM. The Sequential approach changes VMs modes one by one, which introduces extra delay and is also vulnerable to budget replenishment delays. The Batch approach exploits parallelism to mitigate the budget replenish issue: "decreasing" requests are sent in a batch, followed by the "increasing" sublist. The Greedy approach opportunistically increases resource interfaces of VMs as soon as possible, and thus yields a better result than Sequential or Batch. The Boost approach avoids budget replenishment delays by boosting VCPUs on demand. Boost consistently outperforms the others by providing a lower overall mode switch latency. Boost also reduces the variability of mode switch delays.

Compared to the user-level approaches, VCPU boost can drastically reduce mode switch latency, but it requires patching the hypervisor. The choice between the user-level approaches and the Boost approach therefore involve a trade-off between performance and hypervisor modifications.

2.5 Evaluation

To evaluate M2-Xen, we conducted an extensive set of experiments, using randomly generated real-time workloads and micro-benchmarks. We had three main objectives: (1) evaluate the real-time performance improvement (in terms of missed deadlines) of M2-Xen over vanilla Xen, which is Xen with static configuration, for multi-mode systems; (2) evaluate the latency overhead incurred by mode-switch operations with micro-benchmarks; and (3) compare the overall mode-switch latencies of four fast mode switch policies.

2.5.1 Experimental Setup

Hardware. We conducted the experiment on a machine with an Intel E5-2683v4 16-core chip and 64GB memory. We disabled hyper-threading and power saving features and fixed the CPU frequency at 2.1 GHz to reduce the system's unpredictability as in [64, 141, 138].

Hypervisor and VMs. We used Xen 4.8.0 with the RTDS scheduler as the baseline hypervisor. We used our modified Xen with the VCPU boost feature for the overall latency evaluation. We used Linux 4.4.19 for all VMs. We configured Domain 0 with one full-capacity VCPU pinned to one dedicated core, i.e., PCPU 0. For each VM, We set the irq-affinity of the network interface (*eth0-q0*) to the VM's VCPU 0 and disabled *irqbalance*. In the deadline miss experiment, we used 12 VMs, each of which had 3 VCPUs. In other experiments, we used 6 - 12 VMs, each of which had multiple VCPUs. In each experiment, we randomly generated test cases, each of which had different numbers of VCPUs and different resource interfaces for the VMs.

Mode change manager. The HMM and the mode request generator were deployed in Domain 0. The HMM and the DMM processes were scheduled by the Linux SCHED_FIFO scheduler with a priority of 98. The mode request generator randomly generated mode switch requests that were sent to the HMM through a socket. The DMM in each VM was pinned onto the VCPU 0 of the VM.

Real-time workload. Each real-time workload was an independent process. We set real-time workloads' priorities to 1 - 95 under the Linux *SCHED_FIFO* scheduler so that the scheduler becomes the Rate Monotonic scheduler to schedule the real-time workloads. The WCET and the period of each task were determined by test cases.

Test case generation. We randomly generated test cases each comprising a set of VMs hosted by a multicore host. Each test case has two system modes. Each VM has different task sets and resource interfaces in different modes. The task sets in both modes are schedulable based on CSA performed using the CARTS tool [109].

The test case generator guaranteed that each generated test case had the following properties: (1) the total bandwidth of both modes are the same; (2) the number of tasks generated for a VM V_k with N_k VCPUs was no smaller than the number of VCPUs N_k; (3) the system was claimed schedulable in each mode by CARTS [109]; (4) the resource interfaces and task sets of a VM are different in different modes. In case a randomly generated test case violated these properties, the test case generator discarded the test case and regenerated one.

2.5.2 Real-Time Performance in Multiple Modes

We compared M2-Xen against Xen in term of real-time performance of multi-mode systems. As baselines for comparison, we considered two representative approaches to configure resource interfaces of VMs on Xen: (1) based on the *initial mode*, or (2) based on the *worst-case mode*. When the resource interfaces were configured based on the initial mode, the system may suffer deadline misses after a mode switch if a VM requires larger bandwidth in the new mode. On the other hand, Xen can avoid deadline miss in any mode if the resource interface of each VM is configured based on the largest bandwidth in any mode. However, the system would be forced to over-provision resources unless all VMs experience their respective worst-case workloads in the same mode.

We first compared the empirical performance of M2-Xen against Xen when the latter was configured based on its initial mode. In this set of experiments, we randomly generated 40

test cases each has two modes. In each mode, each test case has 12 3-VCPU VMs running on 15 PCPUs, and each VM contained a set of randomly generated tasks whose periods was uniformly distributed in $[100, 1000]ms$. The total VCPU bandwidth of all VMs was 8.0 in each mode, but the required bandwidth of each individual varied between modes. The period of each VM's resource interface is $5ms$. Under M2-Xen each VM had a resource interface for each mode. In contrast, under Xen each VM had a fixed resource interface computed based on the initial system mode (Mode 0).

We ran each test case for $200s$. The system started in Mode 0 and switched to Mode 1 at $100s$. The system recorded deadline misses during the experiments. As expected M2-Xen consistently met deadlines in both system modes, while Xen suffered deadline misses after mode changes. Specifically, the system experienced deadline misses in 6 out of 40 test cases under Xen. In contrast, no test case witnessed any deadline miss under M2-Xen because it dynamically changed the resource interfaces in response to the mode switches.

We then assessed the cost of over-provisioning resources in Xen based on each VM's worst-case bandwidth need in comparison to the multi-mode interfaces supported by M2-Xen. In this analysis, each VM was assigned the resource interface with the largest bandwidth in both system modes. As a result, tasks in each VM are schedulable in all modes, but this approach led to resource over-provisioning assuming all VMs experience their respective worst-case workloads at the same time, a rare situation in practice. To guarantee the schedulability of the entire system, we used CARTS [109] to compute the required number of PCPUs for these resource interfaces. If the required number of PCPUs was no larger than the available number of PCPUs (i.e., 15 PCPUs in our setting), the system was schedulable. In contrast, M2-Xen provisions different resource interfaces for each VM in different modes. The system

is schedulable as long as the required number of PCPUs are smaller than the number of available PCPUs in each mode.

For the same 40 test cases used in the last set of experiments, we computed the required number of PCPUs for each test case (denoted *Static Worst-case* in Fig. 2.4). Notably, none of the 40 test cases was schedulable on the 15 PCPUs available on our experimental platform. In contrast, all 40 test cases were schedulable in both Mode 0 and 1, which means that they were schedulable under M2-Xen that can dynamically switch to the resource interfaces corresponding to the current mode.

In summary, this set of experiments and analyses have shown that M2-Xen can significantly maintain real-time performance by adapting to mode changes while avoiding resource over-provisioning.

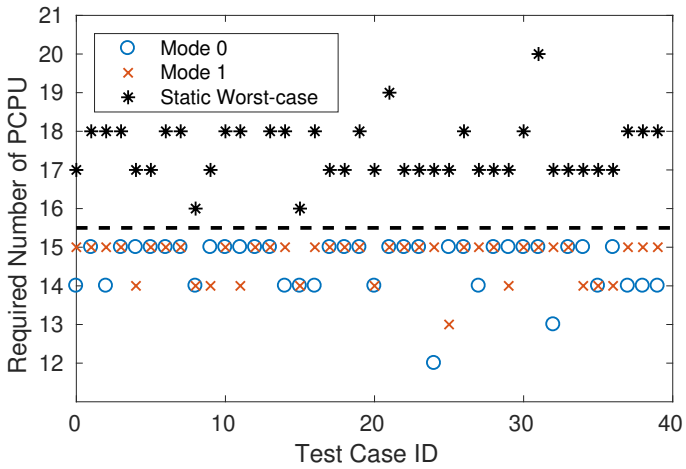


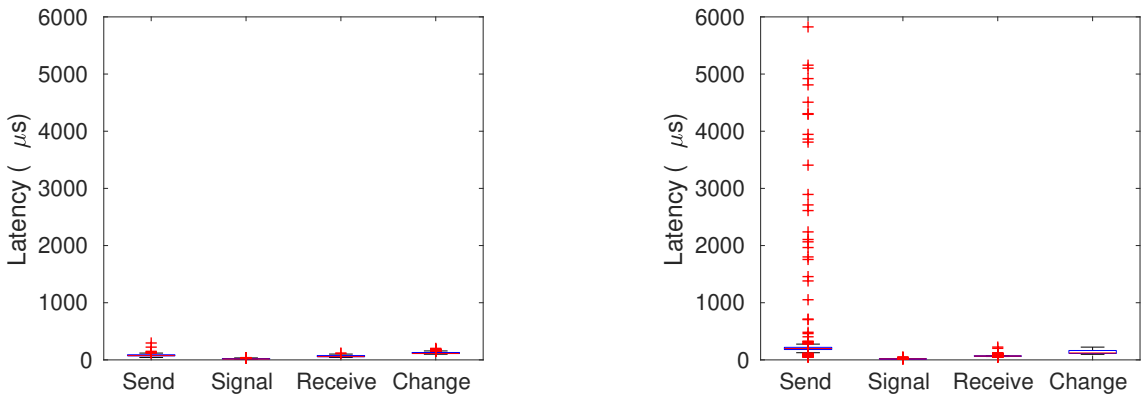
Figure 2.4: Number of PCPU required for different test cases

Table 2.1: Global Mode Setting for Micro Benchmark

Global Mode	**Dom 0** (B, P)	**Dom 1** (B, P)
0	(10ms, 10ms)	(10ms, 10ms)
1	(10ms, 10ms)	(4ms, 10ms)

2.5.3 Latency Breakdown among Mode-Switch Operations

We measured the latency introduced by the mode-switch procedures with micro benchmarks. We created a VM with one VCPU and two modes. Each mode had 4 randomly generated tasks. We had a Domain 0 to run the HMM and the mode switch generator. The resource interface for the VM and the Domain 0 in both modes are shown as Table 2.1. We toggled the system mode between Mode 0 and Mode 1 500 times with the mode-switch interval randomly picked between 200 ms and 400 ms. We measured the delay introduced by each mode-switch procedure at each mode switch event. We classified the delay into two cases: (1) case 1 when the system changed from Mode 0 to Mode 1; and (2) case 2 when the system changed from Mode 1 to Mode 0.



(a) Change from Mode 0 to Mode 1

(b) Change from Mode 1 to Mode 0

Figure 2.5: Boxplot of Mode Change Latencies

Fig. 2.5 shows the latency of the four procedures that contribute to the overall mode-switch latency: (1) Send, which sends a mode change request from the HMM to a DMM; (2) Signal, which uses the task engine to change the task set; (3) Receive, which receives an acknowledgement from the VM; and (4) Change, which uses the HMM to change the VM's resource interface.

We observed in Fig. 2.5 that the latencies of the Signal, Receive, and Change procedures are relatively short and stable. In both cases, the latencies stay within $100\mu s$ most of the time. We also observed that the Send procedure did not suffer a significant delay when system changed from Mode 0 to 1. This is because Domain 1 had a full-capacity VCPU in Mode 0 at the mode switch, which eliminated the budget replenishment overhead in the Send procedure and allowed the DMM to immediately process an inter-VM message from the HMM.

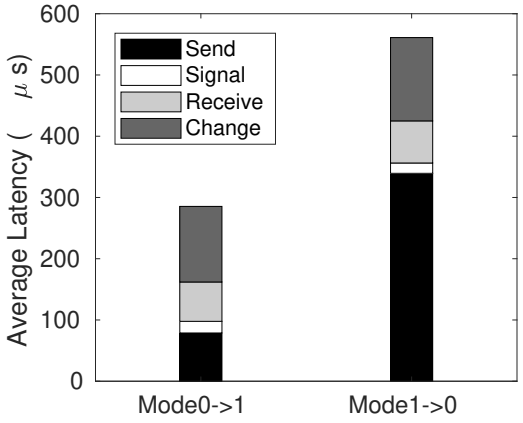
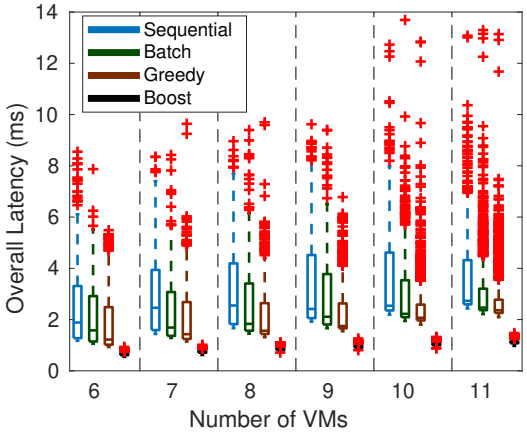
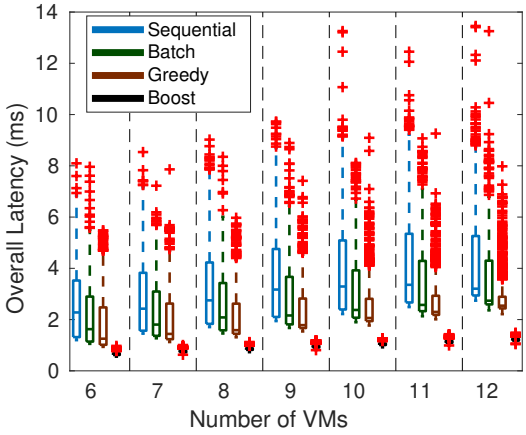


Figure 2.6: Overall Mode Change Latency

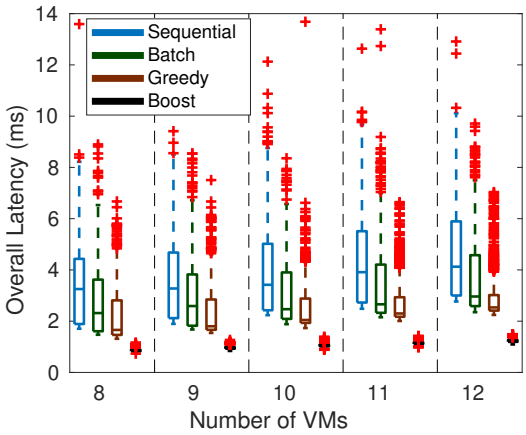
However, when the system changed from Mode 1 to 0, Domain 1 had a partial-capacity VCPU at the mode switch. The VCPU running the DMM may exhaust its budget and be de-scheduled, resulting in budget replenishment delay of up to 6 ms, which was consistent



(a) $W_{total} = 7.0$



(b) $W_{total} = 8.0$



(c) $W_total = 9.0$

Figure 2.7: Boxplot of Mode Change Latencies

with the upper bound of $P - B = 10ms - 4ms = 6ms$. This latency dominated the overall mode change latency. The long tail of the Send latency increased not only the worst-case latency but also the average latency, as shown in Fig. 2.6.

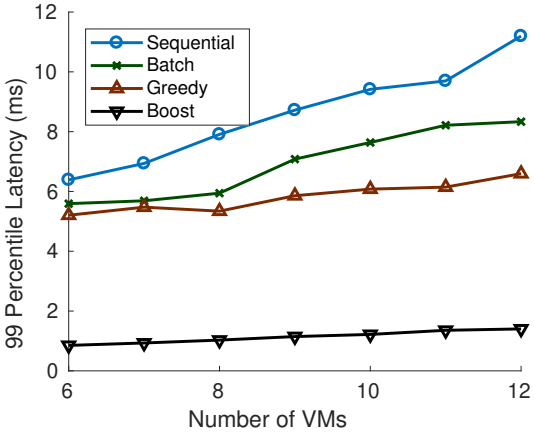
2.5.4 Overall Mode Change Latency

This experiment aims to evaluate the budget replenishment delay of the Send procedure for four different mode switch policies: Sequential, Batch, Greedy, and Boost.

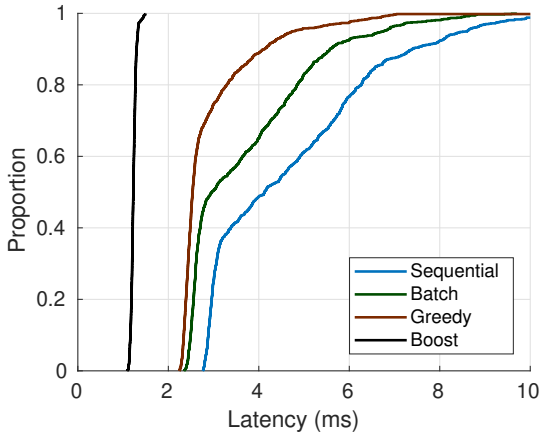
We had 15 PCPUs for guest VMs. Each VM had 3 VCPUs. We generated test cases by varying the total bandwidth W_{total} and number of VMs. We had 10 test cases for each combination of total bandwidth and number of VMs. We had 10 test cases for each $\{W_{total}, \#VMs\}$ setting. For each test case, we switched the system mode between Mode 0 and 1 50 times. The interval between two mode switch requests was randomly choosed [2000, 3000] ms. We ran each test case with different mode switch policies and measured 100 overall mode change latencies for each test case. We had 1000 latency results in total for each test case.

Latency distribution. Fig. 2.7 showed the overall latency in boxplot for the four mode switch policies in different bandwidth settings. We observed that the median latency increased monotonically with the number of VMs. This is reasonable because the HMM was implemented as a single thread server, and all communication and resource interface management were handled on PCPU 0 serially. The more VMs to manage, the longer median overall latency was expected. We observed that the Greedy approach outperformed the other two user-level approaches. In general, the Greedy approach had smaller median latency than the other two approaches, because it leveraged parallelism in an opportunistic manner. However, all three user-level approaches witnessed large variance of latencies.

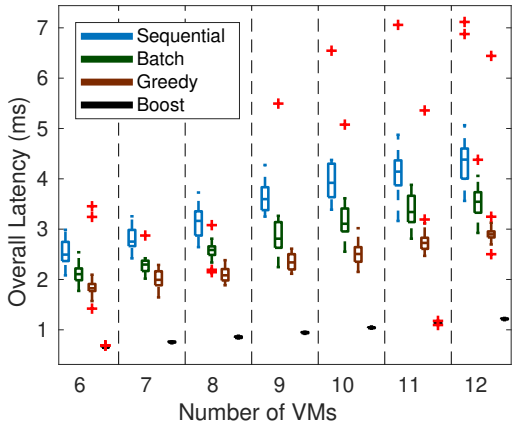
We also observed that the Boost approach significantly outperformed all three user-level approaches across all system bandwidths. To be specific, the Boost approach's overall latencies were consistently smaller than 1.5 ms, while the best user-level approach, i..e, the Greedy approach, had its largest overall latency larger than 10 ms (see the results for 11 VMs in



(a) 99th Percentile Latency, $W_{total} = 8.0$



(b) Empirical CDF, $W_{total} = 9.0$, 12 VMs



(c) Boxplot of Average Latency Among Testcases

Figure 2.8: Latency Distributions for different settings

Fig. 2.7 (a). This is reasonable because the Boost approach avoided the extra delay caused by the budget replenishment.

Fig. 2.8(a) showed the 99th percentile latencies of 1000 samples across different number of VMs. We observed that the 99th percentile latency under all three user-level approaches was always larger than 5 ms, while it was less than 1.5 ms under the Boost approach. This

demonstrated that user-level approaches suffered from long budget replenishment delays which were significantly reduced by the Boost approach.

In order to show the distribution of the latencies of the four mode switch policies, we picked the results for the setting that had 12 VMs and 9.0 total bandwidth. We plotted the CDF of the 1000 latency samples for all four policies in Fig. 2.8(b). We observed that all the latency samples of the Boost approach were distributed in a narrow range $[1.09, 1.49]$ ms. This demonstrated that the Boost approach could provide very stable and small mode switch latencies. In comparison, the Greedy approach, the best user-level approach, had the minimal latency 2.25 ms, and had more than 10% latency samples greater than 4 ms, which is $4/1.49 = 2.68\times$ larger than that of the Boost approach. This again demonstrated the effectiveness of the Boost approach in eliminating the budget replenishment delay.

Performance among different mode settings. Since the mode setting affects the distribution of budget replenishment delay during mode switch, a VCPU with higher bandwidth suffers from less budget replenishment penalty. In this experiment, we evaluated the stability of our mode change policies: the mode switch latency of a stable mode switch policy should not be seriously affected by different mode settings. We randomly generated test cases that had different resource interface settings for this experiment.

We created a boxplot to evaluate the impact of VCPU bandwidth on the average latency among different policies. Each data point was the average latency of 50 delay samples at mode switches of a test case. Fig. 2.8(c) showed the boxplot of four mode switch policies across different number of VMs.

We observed that the Batch and the Greedy approaches were less sensitive to mode settings when compared to the Sequential. This is because these two approaches leveraged the parallelism mitigate the impact of the variable budget replenishment delay.

We also observed that the Boost approach was the most robust one: Fig. 2.8(c), showed that the latencies of the Boost approach were aggregated in a very small range that is less than 100 μs. This is because the Boost approach boosted the VCPU hosting the DMM and scheduled it immediately whenever the HMM sent a mode switch request, making the overall mode switch latency insensitive to resource interfaces.

When the budget replenishment delay was removed, the mode switch latency was dominated by the cost of invoking hypercall to change resource interfaces in Domain 0. Since changing a VM's resource interface usually took constant time and the HMM changed resource interfaces sequentially, the overall latency was expected to have linear relation with the number of VMs, which was confirmed in our empirical results of the Boost approach.

2.6 Related Work

2.6.1 Real-Time Virtualzation Approaches

Real-time virtualization has received significant attention as a promising approach for embedded system integration and latency-sensitive cloud computing [40]. KVM [66], as type-II hypervisor, incorporates itself within the host OS. VCPUs are treated as processes and are scheduled together with other tasks in the host. In KVM, it is possible to apply a real-time scheduler [14, 44] for VCPUs, thus achieving hierarchical real-time scheduling. The

deadline-based real-time scheduler [26] in the IRMOS project [28] is an instance of hierarchical real-time scheduling based on KVM. Another KVM-based scheduler, ExVM [33], adopts flattened scheduling design [69]. It exposes scheduling information within VMs to the host scheduler. While Flattening may potentially improve the performance and flexibility of multi-mode scheduling, an advantage of hierarchical scheduling approach as in M2-Xen is to avoid exposing task-specific information within a VM, which can be appealing to a subsystem vendor who may not want to disclose their proprietary design and intellectual property.

Quest-V [82, 97], a separation kernel, divides physical resources, such as cores and I/O devices, into separated sandboxes. Each sandbox runs its own kernel and schedules tasks directly on the cores in this sandbox. With this architecture, Quest-V establishes a predictable model and addresses the communication delay due to scheduling [81].

The MARACAS scheduling framework [142] addresses memory-aware scheduling, shared cache, and memory bus contention, for multicore scheduling. MARACAS throttles the execution of threads running on specific cores when memory contention exceeds a certain threshold.

While the aforementioned work focused on developing novel architectures and scheduling policies for real-time virtualization, they largely assumed static configuration for real-time VMs. In contrast, M2-Xen incorporates dynamic mode switching and run-time resource allocation within a conventional two-level hierarchical scheduling framework provided by RT-Xen [137, 138].

2.6.2 Real-Time Mode Change Protocols

Burns [21] gave an overview of mode change in real-time systems. Switching between two modes requires eliminating some tasks in the old mode and establishing other tasks in the new mode. For hard real-time systems, the greatest challenge is to guarantee hard real-time requirements during the transition, when the tasks of the old and the new modes may be scheduled simultaneously. For uniprocessor, the *Idle Time Protocol* [130], the *Maximum-Period-Offset Protocol* [9], and the *Minimum Single Offset Protocol* [110] are existing synchronization protocols. Research has also focused on multi-core real-time mode change [98]. Although multi-core mode switch protocols with task-level mode switch have been proposed [73, 29, 22, 45, 114, 50, 110], no relevant study has focused on solution that leverage mode change protocols for virtualized systems, which can change the resource interface and task set. In M2-Xen, we do not guarantee deadline misses are avoided entirely during the transition, although reducing the mode switching latency mitigates the potential for deadline misses during the mode switch. Our current system is therefore designed for soft real-time systems.

Timeliness guarantees for M2-Xen during mode transitions is an interesting yet challenging future direction. To ensure the schedulability of tasks during mode transitions, (1) the resource interface that each VM exposes to the hypervisor must be sufficient to account for the effects of task-level mode changes on the tasks' resource demands; and (2) the analysis at the hypervisor level must consider the impact of VM-level mode changes on the timing behaviors of the VCPUs of the VMs; neither of these is supported by current interface models and analysis. Prior work on multi-mode interfaces and compositional analysis such as [108] can serve as a starting point towards this direction; however, this line of work targets uniprocessors only and assumes the same mode change semantics at both VM and task levels,

and thus it must be substantially extended to work with M2-Xen. We intend to build on this experimental work to develop more rigorous mode switching protocols for Xen in the future.

2.7 Conclusions

We have designed and implemented M2-Xen, a virtualization platform for dynamic real-time systems. In contrast to existing real-time virtualization platforms supporting static resource allocations, M2-Xen can adapt resource allocations among real-time virtual machines in response to system mode changes. As a result, M2-Xen can maintain real-time performance for multi-mode real-time systems without over-provisioning resources. Furthermore, M2-Xen avoids transient system overloads during mode switches and employs user-space and hypervisor scheduling techniques to reduce mode switching latency. M2-Xen has been implemented and evaluated in Xen 4.8 with the RTDS scheduler. Experiments on a 16-core host demonstrated the efficacy, efficiency, and scalability of M2-Xen in comparison to existing static real-time scheduling approaches in Xen.

Chapter 3

Virtualization Agnostic Scheduling: A Scheduling Framework for Achieving Virtualization Agnostic Latency

In this chapter, we present a practical CPU scheduling framework *virtualization-agnostic scheduling (VAS)*, for time-sensitive applications on shared virtualization hosts. In Section 3.1, we specified the task model as well as partitioned hierarchical scheduling model for VAS within an edge host. In Section 3.2, we claim some useful theoretical properties for the scheduling model we specified and then prove the correctness of those properties. In Section 3.3, we show how to implement the VAS on Xen. In Section 3.4, we demonstrate the correctness and usefulness of our VAS system with both synthetic workload and two real case-studies involving Redis and Spark Streaming.

3.1 Scheduling Approach

3.1.1 Task Model

An edge server is a multi-tenant system comprising time-sensitive tasks and general-purpose tasks. A *time-sensitive* task has an SLO in terms of expected tail latency. We assume that a time-sensitive task maintains the same execution time when deployed in different edge clouds. In practice, an edge cloud operator can maintain consistent execution times in different edge clouds by adopting machines from the same vendor according to uniform specifications. While the edge servers share the same hardware profile, they run different mixes of time-sensitive and general-purpose tasks, which may make it challenging to meet the tail latency requirements in different edge clouds. For example, while a manufacturer may deploy edge clouds using the same hardware platform in different factories, the same time-sensitive task may be co-located with a different set of general-purpose applications locally.

A *general-purpose* task has no latency requirements, but may have throughput requirements and demand a certain share of CPU in a multi-tenant system. To improve resource efficiency, we allow a VCPU hosting time-sensitive tasks to share a PCPU with VCPUs hosting general-purpose tasks.

VAS aims to provide (1) *virtualization-agnostic latency (VAL)* for time-sensitive tasks and (2) *resource isolation* for time-sensitive and general-purpose tasks. VAL requires that an application experiences similar latency distributions on a shared host as on a dedicated one. With VAL the same time-sensitive application can be deployed on different edge clouds with similar latency distribution, thereby greatly reducing the effort needed for performance

tuning on individual edge cloud to deliver desired tail latency. In addition, as edge clouds are multi-tenant systems, we need to provide resource isolation between VMs running time-sensitive or general-purpose tasks, so that an aggressive or faulty VM will not cause starvation in other VMs on the same server.

3.1.2 Scheduling Framework

VAS provides a practical scheduling framework to provide VAL and resource isolation for VMs sharing virtualized hosts. A VM has one or more VCPUs. A VCPU may be *time-sensitive* if it runs time-sensitive tasks, or *general-purpose* if it runs general-purpose tasks. A virtualized system employs two-level scheduling. The hypervisor schedules VCPUs on PCPUs, and the VM scheduler schedules tasks on VCPUs.

VM Scheduler. Each time-sensitive VM runs a partitioned scheduler in which each VCPU is scheduled by the same **stable** and **work-conserving** scheduling policy on that VCPU. A stable scheduling policy forms the same schedule for the same input. A work-conserving policy never lets the VCPU idle when there are pending tasks on the VCPU. Note that we do not require the partitioned scheduler to be work-conserving on the multicore VM. Instead, because each VCPU is scheduled independently in partitioned scheduling, we only require the scheduling policy on the VCPU to be work-conserving locally. For example, common scheduling policies such as EDF, RM, and FIFO are work-conserving.

Hypervisor Scheduler. The hypervisor employs partitioned fixed-priority scheduling, i.e., a set of VCPUs is pinned to and scheduled on a PCPU based on a preemptive fixed-priority policy. A VCPU is scheduled as a deferrable server with a *resource interface* (B, P, R), where the *budget* $B > 0$, the *period* $P \geq B$, and R is the *priority*. Accordingly, the VCPU will be

scheduled to run for B time units every P time units, at priority R. The deferrable server provides a mechanism to meet the latency requirement of time-sensitive VCPUs and provide resource isolation for both time-sensitive and general-purpose VCPUs.

Only one time-sensitive VCPU can be allocated on a PCPU and runs at the highest priority on the PCPU, which can be shared with multiple general-purpose VCPUs. Therefore, a host with m cores may host up to m time-sensitive VCPUs. Though restricting the number of time-sensitive VCPUs on a host, this solution is suitable for edge servers on which workloads are dominated by general-purpose services[5]. While restrictive, this scheduling approach provides a practical and efficient way to achieve VAL for time-sensitive tasks. A contribution of this work is to establish both theoretically and experimentally that VAS can achieve its design goals (as is detailed in Sections 3.2 and 3.4).

3.2 Theoretical Properties

Our first design goal is to achieve virtualization-agnostic latency for all the tasks within a time-sensitive VM. In this section we will prove that, theoretically, VAS leads to the same task schedule on a partial CPU as on a full CPU. While it is not the only way to achieve virtualization-agnostic latency, this theoretical property further enhances the predictability of time-sensitive services deployed on partial CPUs.

The intuition originates from a well-known property of fixed-priority scheduling: the highest-priority task's schedule is not affected by lower-priority tasks (assuming there is no other resource sharing dependency). We formalize this property in the case of VAS in Theorem 1.

[5]A similar but simpler system model was adopted for predicting latency distributions of aperiodic services on virtualized platforms, which assumed a single time-sensitive service on each PCPU [79]. VAS allows multiple time-sensitive services to share the time-sensitive VCPU on each PCPU.

However, in a hierarchical scheduling scenario, simply assigning the highest priority to the time-sensitive VM (VCPU) is not sufficient to maintain the same task schedule within the VM, because the budget enforcement mechanism may alter the schedule of the task when a server is suspended. The key theoretical contribution, therefore, is to establish the conditions under which the budget enforcement will not be activated for a deferrable server (if the time-sensitive application conforms to its workload specification). Further, we derive the minimal VCPU bandwidth and server configuration that avoids budget enforcement (Theorem 2): hence, we can achieve virtualization-agnostic latency with an optimal configuration in terms of resource consumption.

3.2.1 System Model

We now formalize the system model as a basis for our theoretical analysis. Recall that VAS employs a partitioned scheduler in the hypervisor on which VCPUs are partitioned among the underlying PCPUs. Because each PCPU is scheduled independently, henceforth, we focus on a single PCPU that is running a single time-sensitive VCPU and a set of general-purpose VCPUs.

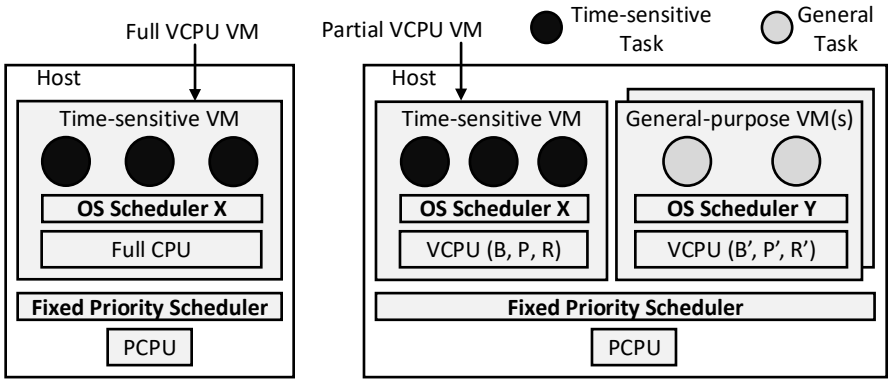


Figure 3.1: VAS: System Model

A free offset periodic (or sporadic) *task system* Γ is defined as $\Gamma = \{\tau_i = (A_i, C_i, T_i) | i = 1..N\}$, where the *offset* $A_i \geq 0$, and the *worst case execution time (WCET)* $C_i > 0$. The *period* of a task (or *minimal inter-arrival time* for a sporadic task) $T_i > C_i, T_i \in \mathbb{N}^+$. The *utilization*, U, of task system Γ, is $U = \sum_i \frac{C_i}{T_i}$. The *hyperperiod* of the task system is $LCM\{T_i\}$, the least-common-multiple of the period of all the tasks. A VCPU is a deferrable server whose *resource interface* is (B, P, R). The *bandwidth* W of a VCPU is $W = \frac{B}{P}$. On a PCPU the task system of interest resides in a *time-sensitive VCPU*, while other VCPUs on the same PCPU are denoted as *general-purpose VCPUs* (Fig. 3.1). The time-sensitive task system is scheduled hierarchically by a scheduler X on the VCPU; and the VCPU (deferrable server) is scheduled by a preemptive *Fixed Priority (FP)* scheduler in the hypervisor, on the PCPU. A *full CPU VCPU* is a time-sensitive VCPU whose bandwidth $W = 1$. Hence, it monopolizes the single PCPU of the host (Full VCPU VM in Fig. 3.1). A *partial CPU VCPU* is a time-sensitive VCPU whose bandwidth $W < 1$. Thus, it can share the PCPU with VCPUs from other general-purpose VMs (Partial VCPU VM in Fig. 3.1).

For simplicity of presentation, in the rest of the paper, we will refer to full CPU VCPU as "full VCPU", and to partial CPU VCPU as "partial VCPU". Let the task system of interest be Γ, whose $U < 1$. Γ is scheduled in either a full VCPU or a partial VCPU. We denote $G(\Gamma, t)$ as the scheduled job of a task system Γ at time t.

For the same task system Γ and same stable and work-conserving scheduler X, we want to answer two questions:

1. Is it possible to make the schedule, $G'(\Gamma, t)$, in a partial VCPU, the same as the schedule, $G(\Gamma, t)$, in a full VCPU for all time t, by setting the resource interface of the partial VCPU?

2. If so, what is the optimal resource interface setting, in terms of achieving the minimal bandwidth W?

3.2.2 Theorems and Proofs

Theorem 1. *The schedule of* Γ *in a partial VCPU,* $G'(\Gamma, t)$*, can be identical to the schedule* $G(\Gamma, t)$ *in a full VCPU, as long as (1) the VCPU of the partial VCPU has the highest priority among other VCPUs and (2) the budget of the VCPU is never exhausted within any period of the VCPU.*

From Theorem 1, if enough budget is provided for each period of a partial VCPU with highest priority, we can ensure that Γ achieves the same schedule as it would on a full VCPU. Then, Theorem 2 indicates an optimal resource interface can be achieved:

Theorem 2. *To make the same schedule* $G(\Gamma, t)$ *in a full CPU, the minimal bandwidth of a partial VCPU should not be less than the utilization of* Γ*, i.e.,* $W \geq U$*. Specifically, if the period of the partial VCPU is the hyperperiod of* Γ*, i.e.,* $P = LCM\{T_i\}$*, then* $W = U$*. Thus, the resource interface of the partial CPU is* $(P \times U, P, Highest)$*;*

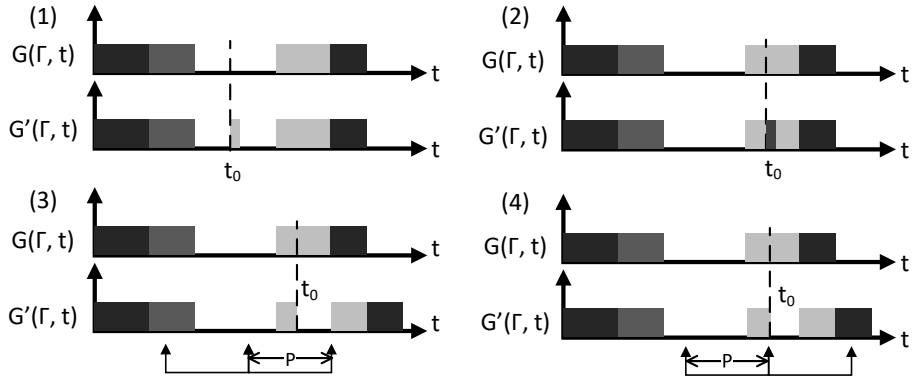


Figure 3.2: Proof of Theorem 1

Proof of Theorem 1. We prove Theorem 1 by contradiction: We assume that we can find the earliest time t_0 where the partial CPU's schedule differs from that of a full one, even though the partial CPU has highest priority and the budget is never exhausted within a period. That is, $\exists t_0 \geq 0$, where,

$$\begin{cases} G(\Gamma, t) = G'(\Gamma, t) & t \leq t_0 \\ G(\Gamma, t_{0+}) \neq G'(\Gamma, t_{0+}). \end{cases} \tag{3.1}$$

For time t_0, there are four possible cases (Fig. 3.2):

(1) $G(\Gamma, t_{0+})$ has no job scheduled; $G'(\Gamma, t_{0+})$ has a job.

(2) $G(\Gamma, t_{0+})$ and $G'(\Gamma, t_{0+})$ have different jobs scheduled.

(3) $G(\Gamma, t_{0+})$ has a job scheduled; $G'(\Gamma, t_{0+})$ has no job; and t_{0+} falls in a deferrable server's period.

(4) $G(\Gamma, t_{0+})$ has a job scheduled; $G'(\Gamma, t_{0+})$ has no job; and t_{0+} is at the end of one period of a deferrable server.

Case (1) implies that a new job is released at t_0. However, the full VCPU does not schedule this pending job, which contradicts the "work-conserving" assumption. For case (2), the schedulers in the full and partial VCPU should have the same scheduling state (and pending queue), but they make different decisions, which contradicts the "stable" assumption. Because scheduler X is a work-conserving scheduler, it can refuse to schedule a job for $G'(\Gamma, t_{0+})$, only if the VCPU budget is exhausted. Thus, Case (3) contradicts the "budget never exhausted within a period" assumption. For case (4), even if the budget is exhausted

at the end of the previous period (t_{0-}), the budget will replenish at the start of a new VCPU period, thus also creating a contradiction. We find a contradiction for each case, thus proving Theorem 1. □

Before proving Theorem 2, we introduce some auxiliary functions and lemmas. Let the *budget demand function* , $D(t)$, of task system Γ, be

$$D(t) = \sum_{i=1}^{N} \sum_{j=0}^{\infty} C_i \times u(t - A_i - jT_i), \tag{3.2}$$

where $u(t)$ is the *Heaviside step function*:

$$u(t) = \begin{cases} 0 & t < 0 \\ 1 & t \geq 0. \end{cases} \tag{3.3}$$

The budget demand function $D(t)$ denotes the total execution time that all jobs released before t request from the task system Γ (where a job of τ_i would request C_i execution time units as soon as it is released). $D(t)$ is a stepwise function.

We define $\mathbb{Y} \subset \mathbb{R}$ to indicate where $D(t)$ is continuous:

$$\mathbb{Y} = \mathbb{R} \setminus \{t = kT_i + A_i | \forall k \in \mathbb{N}, i = 1, 2..N\}. \tag{3.4}$$

Let the continuous *budget supply function*, $S(t)$, be

$$\begin{cases} S(t) = 0 & t \leq 0 \\ \frac{d}{dt}S(t) = 0 & S(t) = D(t), \quad t \in \mathbb{Y} \\ \frac{d}{dt}S(t) = 1 & S(t) < D(t), \quad t \in \mathbb{Y} \\ \lim_{x \to t} S(x) = S(t) & \forall t \in \mathbb{R}. \end{cases} \tag{3.5}$$

The budget supply function $S(t)$ denotes **the total execution time provided by a full VCPU before time** t. The third equation in (3.5) indicates that when there are pending (unfinished) jobs, the VCPU is active and providing service to the jobs in the queue. The second equation in (3.5) indicates that when there are no pending jobs and hence the queue is empty, the VCPU is idle. Hence, $S(t)$ reflects the budget supply behavior of a *work-conserving* scheduler. The fourth equation denotes the continuity of $S(t)$ [6].

Fig. 3.3a shows the demand and supply values vs. time, i.e., $D(t)$ and $S(t)$, of a typical periodic task system $\Gamma_1 = \{\tau_i = (A_i, C_i, T_i) | i = 0, 1, 2, 3\}$, where the parameters (in ms) of four tasks are (150, 40, 250), (100, 200, 500), (50, 100, 1000), and (0, 200, 2000), respectively.

Lemma 3. *For the same task system with any work-conserving scheduler X, $S(t)$ is unique.*

Proof. $D(t)$ is determined by task system Γ. $D(t)$ is a non-decreasing stepwise function, so $S(t)$ can be determined by $D(t)$ uniquely. According to the definition of $S(t)$ in Eq. 3.5, $S(t)$ reflects a generic work-conserving scheduler's behavior, so any work-conserving scheduler should have the same $S(t)$. □

[6] To clarify, $D(t)$ and $S(t)$ are different from the well-known *demand bound function*, $dbf(t)$ [13] and *supply bound function*, $sbf(t)$ [119], respectively. While demand bound function and supply bound function are typically used for schedulability analysis, we introduce $D(t)$ and $S(t)$ to find the resource interface that avoids budget exhaustion for deferrable server.

Lemma 4. *If $P = LCM\{T_i | i = 1, 2..N\}$, then $S(t)$ satisfies:*

$$S(t+P) - S(t) = PU, \qquad t \geq \max_i \{A_i\}, \tag{3.6}$$

$$S(t+P) - S(t) \leq PU, \qquad t < \max_i \{A_i\}. \tag{3.7}$$

Proof. Let scheduler X be an EDF scheduler, considering task system Γ with *implicit deadlines*. Then, using Theorem 1 in [78], $\exists t_0 = \max\{A_i\} \geq 0$, where the schedule repeats itself after t_0, with a hyperperiod $P = LCM\{T_i\}$.

In each hyperperiod, the schedule is the same. So $S(t_0 + (n+1)P) = S(t_0 + nP) + M$, where M is a constant value. For time $[0, t_0)$, the accumulated budget supply is $S(t_0)$. By definition, the utilization, U, is

$$U = \lim_{n\to\infty} U(n) = \lim_{n\to\infty} \frac{S(t_0) + nM}{t_0 + nP} = \frac{M}{P} \Rightarrow M = PU.$$

Using Lemma 3, if $S(t+P) - S(t) = PU$ is true for the EDF scheduler, the theorem holds for any work-conserving scheduler. Inequality (3.7) holds because not all tasks start releasing jobs before $\max_i \{A_i\}$. Since the load of the system is less, the budget supply in a hyperperiod is less. □

Proof of Theorem 2. The first part of Theorem 2 is trivial and can be proved by contradiction: if $W < U$ holds, the partial CPU is overloaded due to a lack of resources and hence cannot achieve the same schedule as a full one.

The second part of Theorem 2 is $P = LCM\{T_i\} \Rightarrow W = U$, which we prove as follows: From Lemma 4, we have $P = LCM\{T_i\} \Rightarrow S(t+P) - S(t) \leq PU$. Using Theorem 1, a

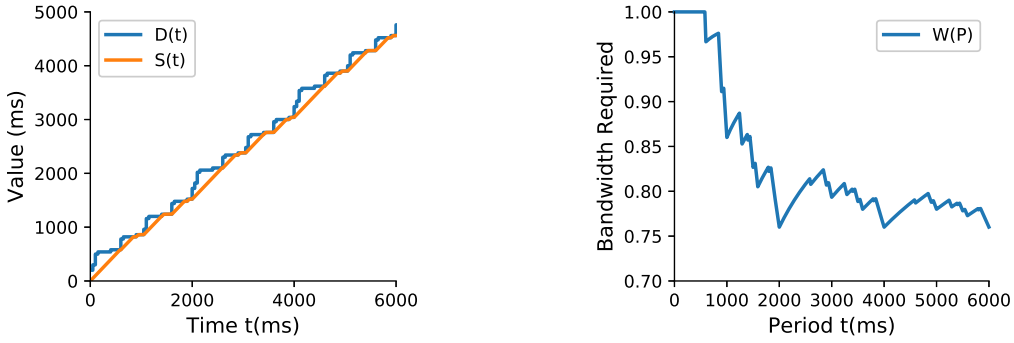
sufficient condition to achieve the same schedule is to keep the budget supply of any period of a partial CPU not less than the budget supply of the corresponding interval in a full CPU, i.e., $B \geq S(t+P) - S(t)$. So, if $B = PU$, we can guarantee $G(\Gamma, t) = G'(\Gamma, t)$. Thus, we can achieve the minimal bandwidth which equals the task utilization, $W = \frac{B}{P} = U$. The proof for sporadic tasks shares a similar procedure, with equations (3.2) and (3.6) being changed to inequalities. □

Theorem 2 explicitly indicates how we can configure a partial CPU to both maintain the scheduling of a full CPU and achieve minimal VCPU bandwidth:

Given a taskset $\Gamma = \{\tau_i = (A_i, C_i, T_i) | i = 1, ..N\}$,

1. Compute the hyperperiod $LCM\{T_i\}$;
2. Compute the utilization $U = \sum_i \frac{C_i}{T_i}$;
3. For the partial VCPU with a resource interface (B, P, R),

   let $P \leftarrow LCM\{T_i\}$, $B \leftarrow PU$, $R \leftarrow Highest$. □



(a) D(t) and S(t)

(b) W(P)

Figure 3.3: Important Curves Related to Γ_1

Theorems 1 and 2 can be explained from the another point of view. Theorem 1 indicates that, given a certain periodic task system within a time-sensitive partial VCPU, we can always avoid budget exhaustion in any period and hence achieve VAL. All we need to do is allocate enough bandwidth for the VCPU. The minimal bandwidth is a function of the period of the VCPU, i.e., $W(P) = \max_t\{S(t+P) - S(t)\}$. Theorem 2 states that $W(P)$ achieves its minimum when $P = LCM\{T_i\}$, i.e., $W = \min_P\{W(P)\} = U$. For example, computing $W(P)$ for task system Γ_1, we illustrate $W(P)$ in Fig. 3.3b. Any point on this figure refers to a certain resource interface configuration. Using Theorem 1, if the point sits on or above the solid line, we can keep the virtualization-agnostic property. Using Theorem 2, if $P = 2000ms$, which equals the hyperperiod of Γ_1, we can achieve the minimum of $W(P)$, $W = U = 76\%$.

A potential limitation of the resource interface configuration for the time-sensitive VCPU is that the general-purpose VCPUs sharing the PCPU may be deprived of CPU time for an extended period of time when the hyperperiod or utilization of time-sensitive tasks is large. Hence, VAS assumes the general-purpose tasks to be tolerant of delays. In addition, time-sensitive tasks with harmonic periods help avoid a large hyperperiod.

3.3 System Implementation

We have implemented VAS in the Xen hypervisor (4.10.0) on a multi-core host. In the hypervisor, VAS employs a partitioned fixed-priority scheduling policy, and a VCPU needs to be scheduled as a deferrable server. The existing RTDS scheduler in Xen already schedules a VCPU as a deferrable server, but employs a global EDF scheduling policy. Hence we implemented a new partitioned fixed-priority (FP) scheduler in the RTDS scheduler, which

was similar to the original partitioned fixed-priority scheduler implemented in RT-Xen [138]. Specifically, we extended `struct rt_vcpu` by adding the field `uint32_t prio`, and modified the function `compare_vcpu_priority()` to compare the `prio` values rather than their absolute deadlines. To support partitioned scheduling, we set a VCPU's PCPU-affinity via the VM configuration file to bind the VCPU to a PCPU at run time. Finally, we modified *libxl* and *libxc* to support an "-r" option for the command "*xl sched-rtds*" so that any VCPU's priority is configurable via Domain 0.

To configure the resource interface of a time-sensitive VCPU in a real system, we need to take into account several system issues that are not modeled in the theoretical results. First, VAS and the VM schedulers incur scheduling and context switching overhead at both the hypervisor and the VM levels. Moreover, interrupt handlers and other kernel services in the VMs may consume additional CPU time. Although those tasks may be modeled as sporadic tasks and thus can be handled by our theoretical analysis, this solution may force us to choose a very large period for the VCPU, because the minimal inter-arrival time of those tasks could be large. We therefore take a pragmatic approach to handle the additional system activities and overhead through moderate overprovisioning. In our experiments with Linux-based VMs, we found that adding 5% more CPU bandwidth to the theoretical bound was sufficient to deliver virtualization-agnostic latency[7].

[7]While the amount of overprovisioning is heuristically derived, the same margin is used in Linux [27]. As future work it will be interesting to leverage earlier research on characterizing virtualization overhead [4] to configure the overprovisioning systematically.

3.4 Evaluation

In this section, we present three different sets of experiments for testing the VAS system: (1) synthetic taskset experiments, (2) a case study on Redis, and (3) a case study on Spark Streaming. The synthetic taskset experiments examined how VAS performs in periodic/sporadic task systems with different work conserving schedulers. The two case studies were designed to assess the effectiveness of our approach for real-world applications.

We conducted experiments on a machine with one Intel E5-2683v4 16-core CPU and 64 GB memory. We disabled hyper threading and power saving features and fixed the CPU frequency at 2.1 GHz to reduce unpredictability, as in [64, 141, 138]. We used Xen 4.10.0 with our VAS implementation as the hypervisor scheduler. We used Linux 4.4.19 for all VMs. We configured Domain 0 with one full CPU pinned to one dedicated core, i.e., PCPU 0. We used Redis version 4.0.9, Kafka version 2.0.0, and Spark version 2.3.1.

We mainly concerned with whether a task set within a partial VCPU VM in the VAS system can achieve VAL, i.e., the same latency distributions as those in a full VCPU. We plotted cumulative distribution functions (CDF) to illustrate the difference between the two distributions, and used the *Wasserstein Distance* [132] to quantify differences between two distributions.

3.4.1 Synthetic Server Evaluation

System Architecture. The synthetic task system we used for evaluation is comprised of one server and several clients. Clients dispatch jobs periodically by sending requests to the server within the time-sensitive VM. The server queues incoming jobs, schedules

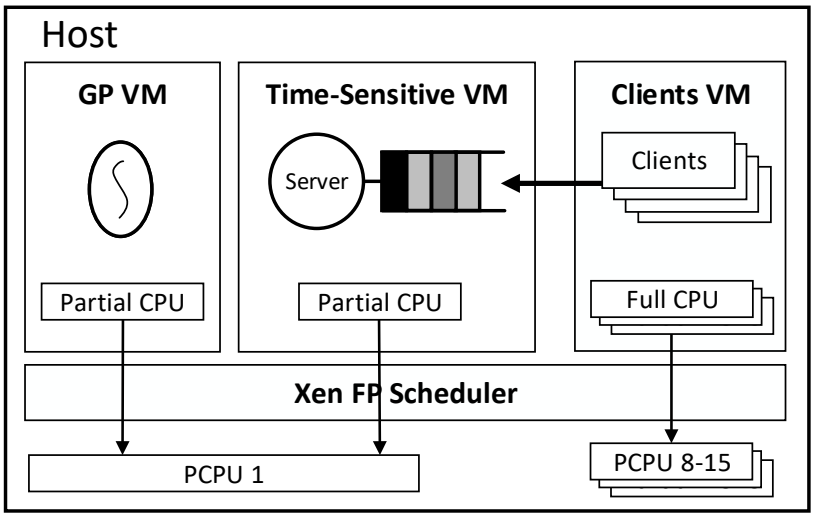


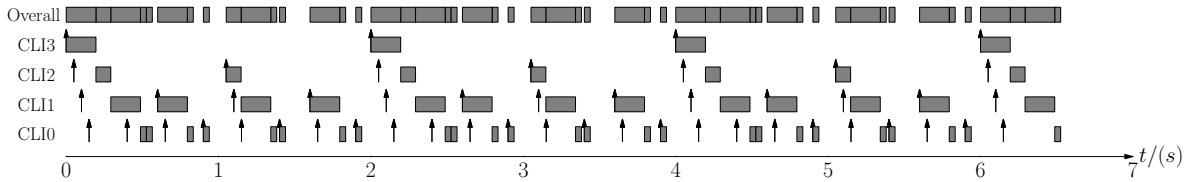
Figure 3.4: VAS System Architecture for a Synthetic Application

them according to a work-conserving scheduling policy, and provides services to jobs. This architecture pertains to many cloud applications, such as Redis, Spark and FTP services. We used three VMs for this experiment, as Fig. 3.4 shows: (1) a 1-VCPU general-purpose VM (GP VM), with the VCPU pinned to PCPU 1. (2) a 1-VCPU time-sensitive VM for running the synthetic server, which shares PCPU 1 with the GP VM in a partial CPU configuration. and (3) an 8-VCPU VM for running clients, with each VCPU pinned to a distinct PCPU (from PCPU 8 to PCPU 15). In the VM for the clients, to make the clients as independent as possible, we assigned clients to VCPUs in a round-robin fashion. The VM for the clients does not share PCPU 8-15 with the other VMs.

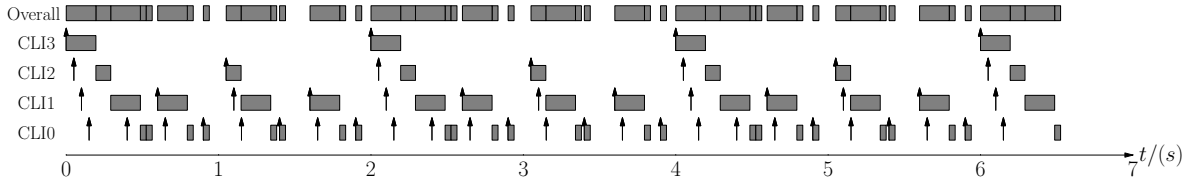
For the partial CPU VM configuration, we configured the resource interface of the VCPU in the time-sensitive VM based on Theorem 2. The GP VM, which was created to share the PCPU with the time-sensitive VM, ran a purely CPU intensive workload to consume CPU cycles when possible. For the full CPU VM configuration, we did not run the GP VM: setting the VCPU in the time-sensitive VM to have full bandwidth, we let the time-sensitive VM occupy PCPU 1 exclusively.

We measured the response time of each job for different combinations of settings: periodic / sporadic tasks, harmonic / non-harmonic tasks, and FIFO / FP schedulers. We then also tested our system extensively by using randomly generated test cases.

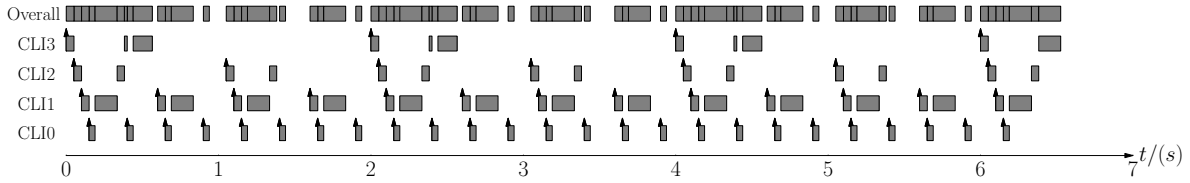
Periodic Tasks. We used a harmonic periodic task setting with a free offset, Γ_1 (defined in section 3.2), whose hyperperiod is 2000 ms. According to Theorem 2, the minimal VCPU bandwidth equals the utilization (76%) of the taskset: the interface (B, P) of the partial CPU is (1520 ms, 2000 ms).



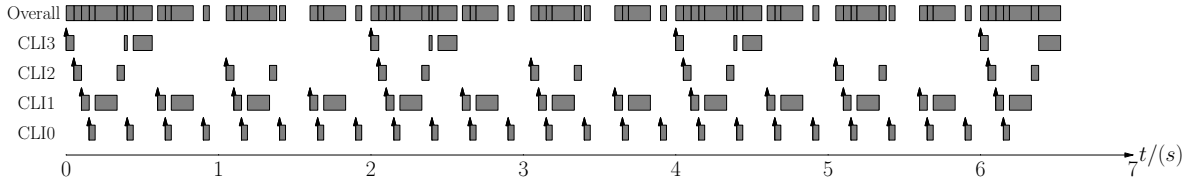
(a) Full CPU (FIFO)



(b) Partial CPU (FIFO)

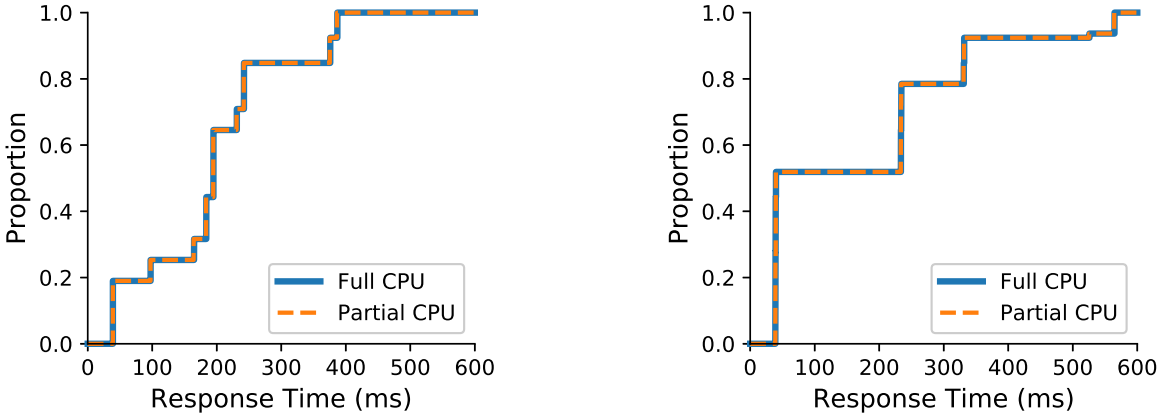


(c) Full CPU (FP/RM)



(d) Partial CPU (FP/RM)

Figure 3.5: Periodic Tasks: Schedule Comparison



(a) FIFO

(b) FP/RM

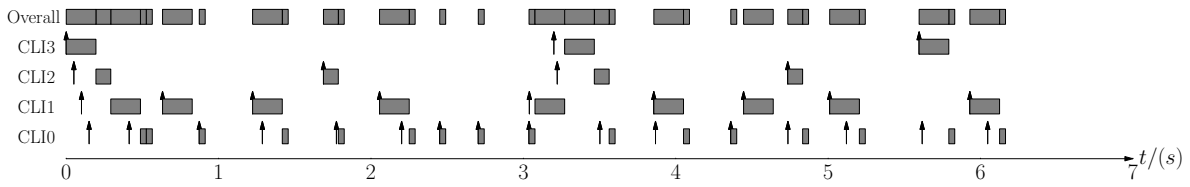
Figure 3.6: Periodic Tasks: Empirical CDF Comparison

We ran the experiment for 100 seconds for both the full and the partial CPU setting on both the FIFO and FP scheduler. For the FP scheduler, we used a rate monotonic priority assignment. We show the first 6 seconds in Fig. 3.5, and make three observations: (1) The schedules of the partial CPU approximate the schedules of the full CPU well, in either a FIFO job scheduler (Fig. 3.5a and 3.5b) or a FP scheduler (Fig. 3.5c and 3.5d). (2) Each schedule repeats itself every hyperperiod. (3) The server's "active states" are identical between the two work-conserving schedulers ("Overall" traces of Fig. 3.5b and 3.5d), which supports the claim in Lemma 3: any work-conserving scheduler can yield the same $S(t)$.

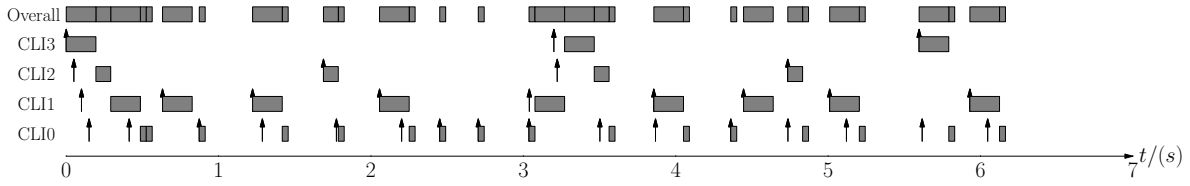
Fig. 3.6 shows the latency distribution of two different VCPU settings in either the FIFO or FP scheduler. We observe that (1) The "stepwise" feature of the CDFs indicates the hyperperiodicity of the system. (2) In either Fig. 3.6a or 3.6b, the partial CPU VM can appropriately achieve VAL: the CDF curves can hardly be differentiated visually, so we used Wasserstein distance to distinguish them, as report later under *Practical Overprovisioning*.

Sporadic Tasks. We modified task system Γ_1 for the sporadic test, with the minimal job arrival interval of each sporadic task being the same as the period of the corresponding

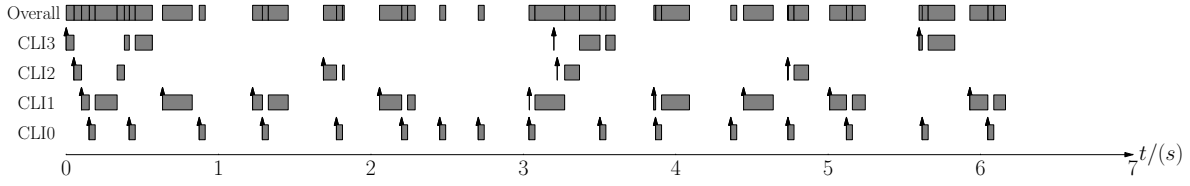
periodic task. Each job arrival interval is generated from a random variable with a uniform distribution over $[P, 2P]$. The random inter-arrival times are generated offline, and thus we can use the same "input" for different runs and make comparisons among them. From the first 6 seconds of the schedule shown in Fig. 3.7, we observe that: (1) The schedules of the partial CPU approximate the schedules of the full CPU well. (2) Because of the randomized inter-arrival times, each schedule no longer repeats itself. (3) The servers follow the same activity patterns for two different work-conserving schedulers (FIFO and FP).



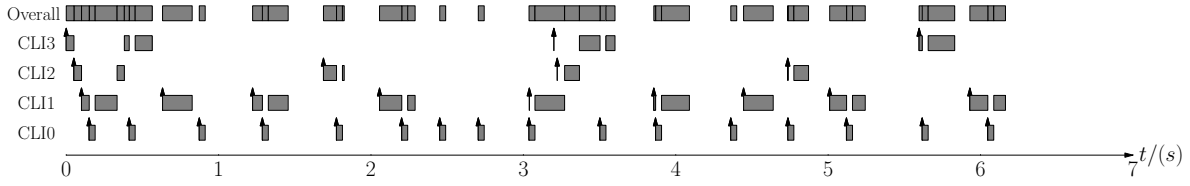
(a) Full CPU (FIFO)



(b) Partial CPU (FIFO)

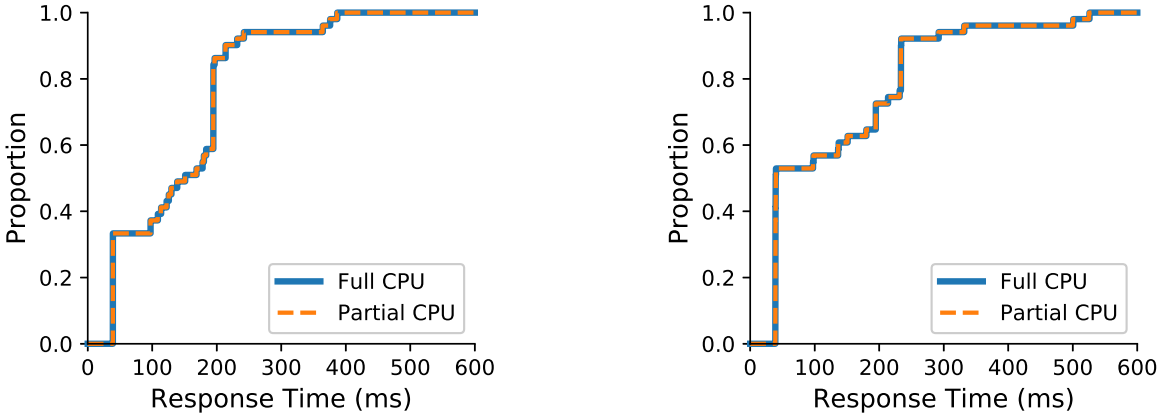


(c) Full CPU (FP/RM)



(d) Partial CPU (FP/RM)

Figure 3.7: Sporadic Tasks: Schedule Comparison



(a) FIFO

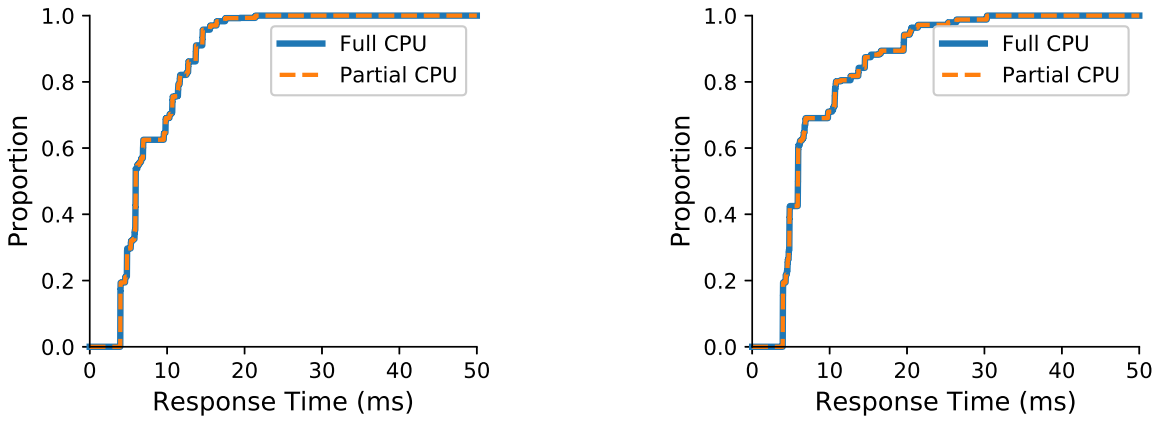
(b) FP/RM

Figure 3.8: Sporadic Tasks: Empirical CDF Comparison

In Fig. 3.8, we again observe overlapping CDF curves: thus, we can appropriately achieve VAL in a **sporadic** setting as well as in a periodic one.

Non-Harmonic Periodic Tasks. We also tested non-harmonic period settings: $\Gamma_2 = \{\tau_i = (A_i, C_i, T_i) | i = 0, 1, 2, 3\}$, where the parameters (in ms) of four tasks are (15, 4, 20), (10, 6, 30), (5, 10, 50), and (0, 7, 70), respectively. The hyperperiod is $2100ms$, which is much larger than each task's period. From the overlapping curves shown in Fig. 3.9, we can conclude that our system is also effective for a non-harmonic setting. A non-harmonic task system can potentially incur a large hyperperiod (with a proportionally large budget), especially for co-prime intervals. As a result, it may prevent a low-priority VCPU from running for a relatively long time. Hence, VAS is more suitable when time-sensitive VMs with harmonic tasks or when general-purpose tasks are tolerant of delays.

Practical Overprovisioning. Using a FP scheduler with RM priority assignment for synthetic server, we tested the periodic task system Γ_1 with different VCPU bandwidth configurations. The task set utilization was 76%. We repeated the periodic task experiment for 100 seconds each run, under three bandwidth settings: 71% (overload), 76% (theoretical



(a) FIFO

(b) FP/RM

Figure 3.9: Non-Harmonic Periodic Tasks, Empirical CDF

minimum), and 81% (overprovisioned). We plotted the CDF curves in Fig. 3.10. The Wasserstein distances of each setting were 67.169 ms, 1.3491 ms, and 65.038 μs.

These results support three main observations: (1) The overload setting (-5% BW) deviates significantly from the full CPU's CDF because the system is overloaded. If the system were run indefinitely, the pending jobs would accumulate and the queue would never be emptied. Even running the system for a finite time (100 seconds), the latency distribution shows a very long tail, with a max latency up to 1457.2 ms. (2) The overprovisioned setting (+5% BW) outperforms the others by giving the best approximation, with a Wasserstein distance of 65.038 μs. (3) The theoretical minimal bandwidth setting approximates the full CPU well. The Wasserstein distance to the full CPU distribution was 1.3491 ms. The system overhead can potentially induce temporary overload of a VCPU and thus deteriorate the latency performance of the system. Hence, we suggest a mild VCPU bandwidth overprovision (by 5%) when adopting our configuration in real-world systems.

Multi-Testcase Evaluation. We generated randomized testcases for our system. Given a desired utilization: (1) We uniformly picked a period P_i, in milliseconds, from the harmonic

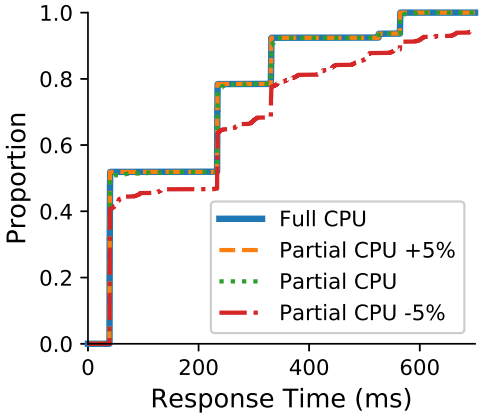


Figure 3.10: CDF, Different RI

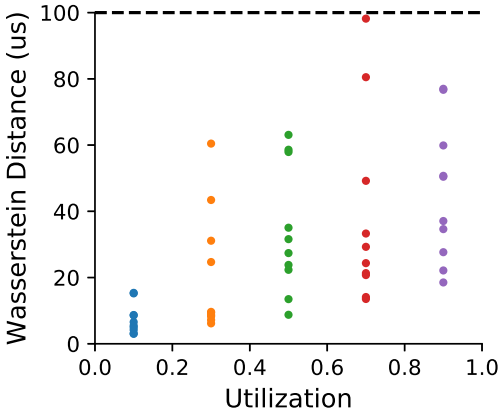


Figure 3.11: Multi-Testcase

set {10, 20, 40, 80, 160, 320, 640, 1280}. (2) We uniformly generated an offset, A_i, over $[0, P_i]$. (3) We also generated a utilization, U_i, for this task, following a medium bimodal distribution, which distributed uniformly over $[0.0001, 0.5)$ with probability of $2/3$, or $[0.5, 0.9]$ with probability of $1/3$ as was done in prior work [138]. We then calculated the WCET, $C_i = P_i U_i$. (4) We repeated steps (1) through (3) to generate more tasks as long as the total utilization was less than desired, then trimmed down the last task to fit the desired total utilization (if necessary), and terminated the procedure.

We generated ten testcases for each of the following utilization settings: 0.1, 0.3, 0.5, 0.7, and 0.9. We ran each testcase for 100s, for both partial and full CPU settings. Fig. 3.11 shows that the Wasserstein distances of all testcases are lower than 100 μs, indicating that the partial CPU configuration approximates the full CPU configuration well for these randomized task sets. We also observed that high utilization settings tended to yield greater distances: the higher the desired utilization, the higher the average task numbers, and the greater the likelihood of long period tasks. Those long period tasks generated fewer samples with the same duration in a single run. As a result, we have a higher chance to find a large Wasserstein distance, since fewer samples can deviate significantly (higher p-value). Longer

period tasks had lower priorities (under RM priority assignment), and thus were more likely to be preempted by higher priority tasks. The execution time jitter of short period tasks also may accumulate, and affect the response time of a long period task.

3.4.2 Case Study: Redis

Redis is a widely used single-threaded [111] in-memory data storage server. Redis is typically used within a virtualized host such as AWS ElastiCache [6]. In the following experiments, we employed Redis as a real-world time-sensitive server to test our VAS system design.

Single Redis VM. The testbed for this experiment was similar to the synthetic evaluation architecture shown in Fig. 3.4, except the synthetic server was replaced with a Redis server. Clients sent "HSET" queries (which are used in the Structured Yahoo Streaming Benchmark [7]) to the Redis server. We leveraged `libhiredis` to implement the querying clients. The WCET of the "HSET" query was 0.12 ms in our system. We used six periodic clients that each sent one "HSET" query with a random key-value every 1ms. Thus, the utilization was 72%. We overprovisioned the bandwidth by 5%, which meant the GP VM could consume at least 23% of the CPU cycles per CPU. We ran the system for 100s. From the CDF curves in Fig. 3.12 and the Wasserstein distance of 9.499μs, we can conclude that the partial CPU configuration in VAS also effectively approximated the full CPU configuration in the Redis test.

Multi-Redis-Tenant on PCPUs. Our system can be applied to a multi-core host by using a partitioned approach. On the remaining 15 cores in our host, we created seven single VCPU Redis VMs, with each VCPU pinned to a PCPU from among PCPUs 1-8. Another 7-VCPU GP VM, running seven CPU-intensive processes, shared PCPUs 1-8 with

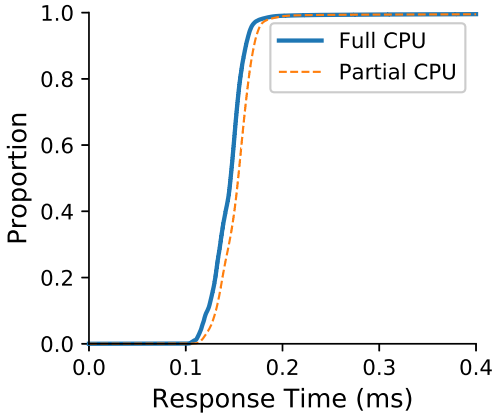


Figure 3.12: Single Redis CDF

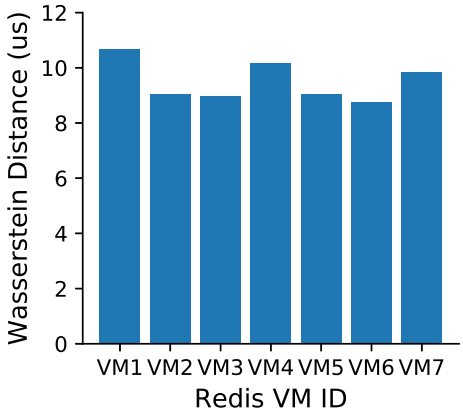


Figure 3.13: Multi-Tenant

the Redis VMs. We used the other eight PCPUs for a client VM and ran seven groups of client processes, with six clients in each group. Each client sent a HSET query (WCET = 0.12ms) to one Redis server instance every 1 ms. The utilization of each Redis server was 72%. We ran the system for 100s, collecting the response times corresponding to each Redis VM. We then computed the Wasserstein distance between the distributions for the full and partial CPU settings.

As Fig. 3.13 illustrates, all seven Redis VMs show similar performance, with a consistent Wasserstein distance around 10μs. We can conclude that our design remained effective in a multi-tenant scenario. This experiment yielded a relatively better result (10μs distance) than that shown in Fig. 3.11. This phenomenon is due to our use of the same execution time and period settings as in the single Redis evaluation, while varying the period and execution time in the synthetic experiments.

3.4.3 Case Study: Spark Streaming

Spark Streaming is a popular streaming and data analytics engine, often run on virtualized hosts. For example, the *Structured Yahoo Streaming Benchmark* [7], an open-source real-time advertisement campaign application, is deployed on the DataBricks' platform running on AWS. In this experiment, we evaluated whether our system model can be easily extended and adopted for a Spark application.

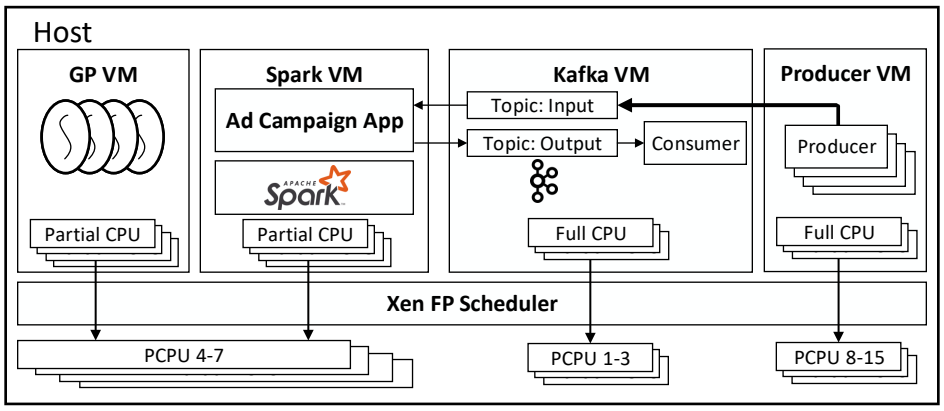


Figure 3.14: Applying VAS to a Spark Streaming Application

System Architecture. We created four VMs for this experiment, as Fig. 3.14 shows: (1) a 4-VCPU VM for Spark, with each VCPU pinned to a PCPU from among PCPUs 4-7, (2) a 4-VCPU GP VM, which shares PCPUs 4-7 with the Spark VM, (3) a 3-VCPU VM for a Kafka message broker, consumer, and Zookeeper, with each VCPU pinned to a PCPU from PCPU 1-3, and (4) a 8-VCPU VM for producers, with each VCPU pinned to a PCPU from PCPUs 8-15.

The Spark VM ran an advertisement campaign application. We modified DataBricks' *Structured Yahoo Streaming Benchmark* [7] under realistic conditions. We set Spark to operate in a local mode with four workers. Each worker ran on one VCPU in a Spark VM. We set the shuffle partitions to 4. The Spark scheduler worked in FIFO mode. We used an 8-VCPU

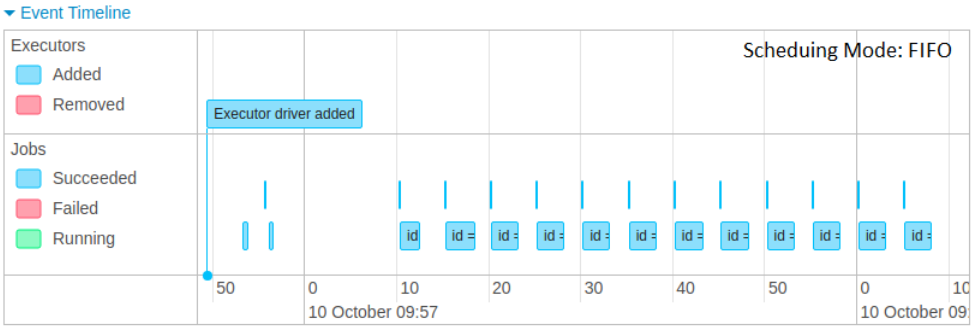


Figure 3.15: Periodic Processing Pattern of Spark

VM for producers, which produced and published advertisement-events to a Kafka "input" topic, at a rate of 28,000 events/s. Those events were then consumed by Spark.

Periodic Processing. Spark features micro-batch processing: The arriving events will not be processed until a periodic micro-batch ends. As a result, Spark processes the incoming events periodically, regardless of the events' arrival pattern. We set the micro-batch window size to 5 seconds via the `writeStream.trigger()` method. Fig. 3.15 is a run-time screenshot of the Spark Web UI when we ran Spark with a full CPU setting. We observed periodic behavior: Spark processed the incoming events every 5 seconds. Effectively, each VCPU within the Spark VM ran a periodic task with a period of 5 seconds. Moreover, this observation was true for every worker in Spark, since the Spark Driver triggered the processing of each worker every 5 seconds. Effectively, each VCPU within Spark VM ran a periodic task with a period of 5 seconds.

Execution Time. Configuring the Spark VM with full-VCPUs, we measured the Spark jobs' elapsed times, which provide a safe estimation of the execution time for each Spark worker. With a total input rate of 28,000 event/s, a 5-second batch-window, four workers (i.e., four periodic tasks) on four PCPUs, Fig. 3.16 shows the distribution of elapsed time. The maximum elapsed time was 3522 ms.

Partial CPU Setting. We used 3522 ms and 5000 ms as the WCET and period for each Spark task, respectively. The utilization for each worker on each VCPU was 70.44%. We set 75% bandwidth for each VCPU in the Spark VM (overprovisioning by 4.56%). Each VCPU in the GP VM took the remaining 24%. The final resource interface for each VCPU in Spark VM was (3750 ms, 5000 ms, Highest).

Note that the Spark experiment was more realistic and yet introduced more sources of variation: (1) The advertisement events were randomly generated for different runs independently. (2) We cannot control the Spark internal or guarantee it handles tasks exactly the same in different runs. (3) Compared to the single-threaded Redis and synthetic server platforms, the Spark platform needs to manage additional threads (e.g., for Spark Web Service, and Spark Driver).

Hence, the absolute value of the Wasserstein distance was much larger than in either the synthetic experiments or the Redis case study, making it hard to tell whether results violate our claims. Thus, we need two full CPU runs and compare the Wasserstein distance between them to indicate the impact of those variations, as a baseline. Then, we made another run in partial CPU setting, to observer whether the Wasserstein distance between partial and full CPU runs significantly exceeds the baseline of two full CPU runs.

We made three runs, each for 26 minutes, two with the full CPU setting, and the other with the partial CPU setting. We compared two runs with **the same full CPU setting**, where the Wasserstein distance was 9.190 ms. Ironically, the partial CPU setting can yield an even better Wasserstein distance of 8.378 ms. Thus, we can still maintain that the partial CPUs in VAS system achieve comparable VAL, as the results in Fig. 3.17 indicate that the latency distribution of the partial CPU again closely approximates that of the full CPU.

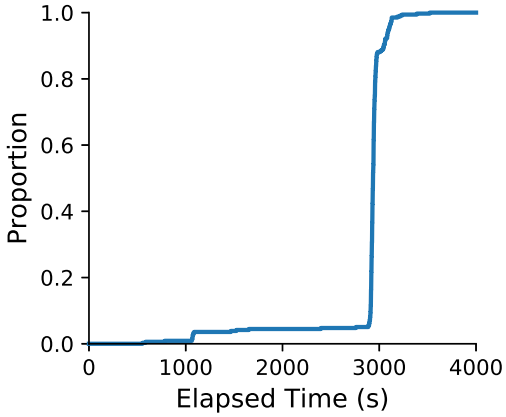


Figure 3.16: Elapsed Time

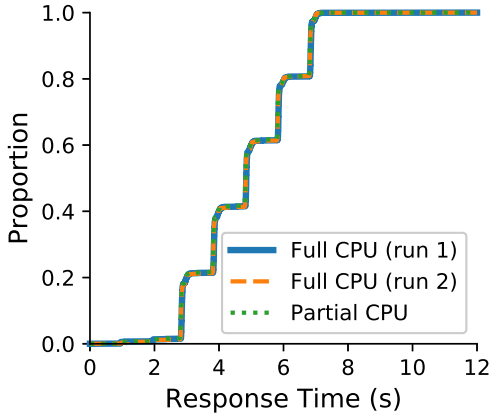


Figure 3.17: Spark CDF

3.5 Related Work

Recent years have witnessed significant research on VCPU scheduling and resource allocation for real-time virtualization systems. A multitude of scheduling approaches have been explored for real-time VCPU scheduling. The Quest-V separation kernel [81] schedules each process in a sandbox as a deferrable server, and takes budget replenishment delay into consideration for predictable communications among sandboxes. vMPCP [65] provides a partitioned hierarchical scheduling framework based on deferrable servers and a synchronization protocol based on the Resource Kernel [102]. RT-Xen [138] developed real-time schedulers in the Xen hypervisor (including the RTDS scheduler adopted in the hypervisor). A predictable VM scheduling framework has also been developed based on standard qemu/KVM, and Linux SCHED_DEADLINE scheduler [3]. RTVirt [145] introduced cross-layer scheduling by sharing scheduling metadata between the hypervisor and the OS scheduler within a VM. These systems leveraged real-time scheduling analysis to provide guarantees in terms of meeting task deadlines. They are not designed to meet probabilistic tail latency target or predictable latency distributions. In contrast, VAS is, to our knowledge, the first scheduling framework to provide VAL for time-sensitive applications on partial CPUs. Based on the

predictable latency distribution, it is hence straightforward to achieve tail latency guarantees with VAS on multiple edge clouds. A limitation of VAS is that it can support only one time-sensitive VCPU per PCPU (shared with multiple general-purpose VCPUs), and hence up to m VCPUs on an m-core processor. VAS is therefore suitable for managing edge clouds each co-hosting a mix of numerous general-purpose workloads and a small set of time-sensitive services with stringent tail latency SLO.

Tableau [134] provides a scalable scheduling framework based on dispatching tables that can be generated on-demand. With low and predictable scheduling overhead, Tableau helps reduce tail latency, but it is not designed to achieve tail latency guarantees or VAL. Tableau is specifically designed for efficient scheduling of high-density workloads. In contrast, VAS focuses on providing VAL for a small number of time-sensitive VCPU sharing CPUs with (potentially larger numbers of) general-purpose VCPUs. Tableau and VAS therefore complement each other, and it will be interesting explore approaches to combine their advantages.

While VAS achieves predictable latency distributions for periodic and sporadic workloads on partial CPUs, it cannot provide the same guarantee for aperiodic services. The problem of predicting latency distributions of aperiodic services on virtualized platforms has been studied theoretically [79]. This work adopted a similar system model as VAS but assumes a single time-sensitive service on each PCPU, while VAS can accommodate multiple time-sensitive services on a single time-sensitive VCPU for each PCPU. For a time-sensitive service with Poisson arrivals and scheduled as a deferrable server, a queueing model is proposed to predict the latency distribution of the time-sensitive service. The scheduling framework proposed in [58] extended the SAF model [32] by allowing aperiodic tasks to run within polling servers. An important research direction is to extend the VAS framework to support aperiodic time-sensitive services based on the theoretical models.

3.6 Conclusions

We introduce virtualization-agnostic latency (VAL) as a desirable property for deploying and managing time-sensitive applications on different edge clouds. In a system providing VAL, a time-sensitive application can maintain similar latency distributions on a partial CPU as on a dedicated CPU, thereby alleviate the significant effort of testing, tuning, and configuring a time-sensitive service for targeted tail latency on numerous edge clouds. We present virtualization-agnostic scheduling (VAS), a simple and effective approach for achieving VAL on a host shared by time-sensitive and general workloads. Two case studies involving commonly used time-sensitive cloud services, Redis and Spark Streaming, demonstrated the efficacy of VAS in achieving VAL in virtualized environments. While this work has taken the first step in supporting VAL, there are several promising directions for future work: On one thing, in addition to CPU scheduling, VAL will also require scheduling support for non-CPU resources. On another, to generalize the VAS approach to heterogeneous platforms, VAS can be extended to tailor the resource interfaces based on the execution times on different platforms and avoid throttling and preemption of time-sensitive VMs, thereby achieving predictable latency distribution on heterogeneous platforms. In next chapter, we will show how to leverage the same VAS model and to conduct the analysis for aperiodic time-sensitive tasks.

Chapter 4

Response Time Analysis for Aperiodic Tasks: Predicting Latency Distributions of Aperiodic Time-Critical Services

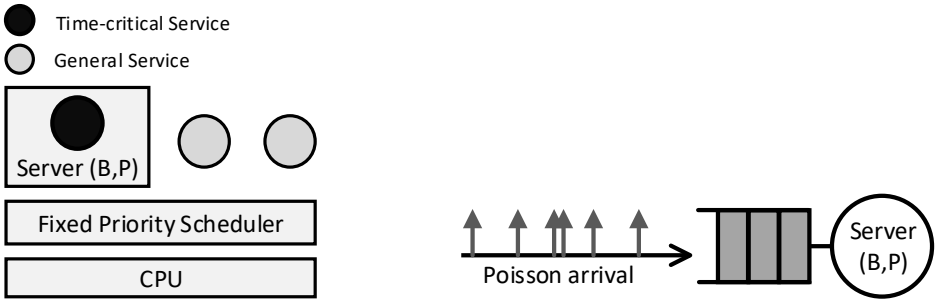
In last chapter, we exploit the handy properties for the VAS system. The VAS model can be simple and effective when dealing with periodic and sporadic tasks. However, aperiodic task are beyond the assumption in VAS model: the budget enforcement of the deferrable server is unavoidable for aperiodic tasks and we are curious about the latency distribution impact from the budget replenishment penalty.

In this chapter, we introduced a novel queueing model, M/D(DS)/1, for an aperiodic task in a real-time virtualization system. We develop a numerical method for computing the stationary response time distribution of a stochastic Poisson process scheduled by a deferrable server.

The system model is similar to what we've see in Section 3.1, while the task is no longer periodic. In Section 4.1, we will introduce the system from a different perspective – as a queueing system. This helps us to compare our system to other relatively well-studied queueing models: M/D/1 and M/D(PS)/1. In Section 4.2, we claim several useful theoretical properties for then prove the correctness of those properties, which are critical to speed up the algorithm for solving the stationary latency distribution of M/D(DS)/1 queue. In Section 4.3, we demonstrate how our algorithm and queueing model can be used to taming the tail latency and achieve the service SLO in an edge environment. In Section 4.4, we evaluates our system in Xen with both synthetic workload and real Redis case study.

4.1 System Model

Our goal is to predict the stationary response distribution of a time-critical service with a Poisson arrival process and a deterministic service time, when the service is governed by a deferrable server. As the first step towards establishing a scheduling theory for time-critical services, in this paper we investigate a relatively simple, but still practical, system model. Specifically, the system has M CPUs. There is one aperiodic, time-critical service assigned on each CPU. There may be other general services sharing the CPU with the time-critical service. The requests for the same time-critical service arrive following a Poisson process and are scheduled in a FIFO fashion. The system employs partitioned fixed-priority scheduling. The time-critical service is scheduled as a deferrable server assigned a higher priority than the other services on the same CPU. Fig. 4.1a shows a single CPU as an example. Despite its simplicity, this architecture is suitable for cloud or edge computing scenarios that provide a mix of time-critical and general services. We focus on CPU resources and do not consider



(a) Scheduling Model (b) Queueing Model

Figure 4.1: M/D(DS)/1 and M/D(PS)/1 Queueing Model

inter-service interference arising from other shared resources, such as cache, memory, and I/O. Handling other resources will be part of future work.

Finding the stationary response time distribution of such a time-critical service can help us to validate whether the service level objective (e.g., a 90^{th} percentile latency of less than 10 ms) can be met, in order to configure practical server parameter settings and thus to provide desired performance.

Due to the stochastic nature of the arriving jobs, the system is amenable to analysis based on a probabilistic queueing model. We transfer the scheduling model (Fig. 4.1a) to the queueing model (Fig. 4.1b) for the purpose of analyzing the stochastic response time of the time-critical service. In the following section, we further introduce our queueing model, starting with the canonical M/D/1 queue.

M/D/1 Queue. An M/D/1 queue presents a single always-active server with Poisson arrivals and deterministic service times. This model has two parameters (λ, d). The *arrival rate* λ means that jobs' inter-arrival times follow an exponential distribution of parameter λ; the constant *service time* is d.

Periodic Server: M/D(PS)/1 Queue. Unlike the M/D/1 queue, a single periodic server is not always active. We use two additional parameters, *budget* and *period*, noted as (B, P), where $B \leq P$, to represent the periodic server model. Since the server has the highest priority, the server's schedule is fixed and switches between the on-state and off-state. When scheduling with other general tasks, as long as the periodic server is in the on-state, its budget linearly decreases, regardless whether it is processing a request. The server stays in the off-state (not providing service) during time $t \in [kP, kP + (P - B))$, and provides service in the on-state during time $t \in [kP + (P - B), (k + 1)P)$. The server runs jobs only when it is in the on-state.

Deferrable Server: M/D(DS)/1 Queue. A server in an M/D(DS)/1 queue is another not-always-active server: a deferrable server. Although M/D(DS)/1 can be also modeled by four parameters (λ, d, B, P) (like M/D(PS)/1, in Fig. 4.1b), the server policy is different. At the beginning of each period, the server replenishes its remaining budget to B. Whenever a job arrives and the server has remaining budget, it can run the job by consuming its budget. The server retains its remaining budget if it is not running, and if the budget is exhausted (reaches 0), the server stops providing any service until the next period, when the budget gets replenished.

System Stability. For a deferrable or periodic server with parameters (B, P), we define the *bandwidth* $W = \frac{B}{P}$. For an M/D/ arrival with parameters (λ, d), the system *utilization* is $U = \lambda d$. In order to make a system stable, say, to achieve statistical equilibrium, the system utilization U should be less than the server bandwidth B. Specifically, for M/D(PS)/1 and M/D(DS)/1 systems with parameters (λ, d, B, P), the stability condition is

$$U = \lambda d < \frac{B}{P} = W. \tag{4.1}$$

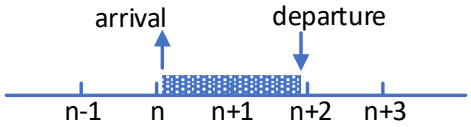


Figure 4.2: Job Arrival and Departure

We are interested in the response time (sojourn time) distribution of an M/D(DS)/1 queue. In this paper, we present a numerical algorithm to derive the stationary response time distribution of a Poisson arrival process with a deterministic service time, subject to the deferrable server policy.

Challenge. The M/D(PS)/1 model has been studied in queueing theory (e.g., traffic flow under a periodic traffic signal) since the mid 1900s. However, it remains difficult to derive the explicit analytical result for the stationary response time distribution of M/D(PS)/1 queueing models. Because the M/D(DS)/1 model is much more complicated than the M/D(PS)/1 model, and since budget management for M/D(DS)/1 is correlated with stochastic arrivals, it is extremely challenging to derive an explicit analytical result for the stationary response time distribution of an M/D(DS)/1 model.

Thus, instead of studying the M/D(DS)/1 model in continuous time using mathematical tools like the *Laplace-Stieltjes Transform*, we use a discrete time analysis to approximate the continuous time M/D(DS)/1 model, as detailed in the next section. Thanks to the *Poisson Limit Theorem* [62], we can first study Bernoulli arrivals with deterministic service times, subject to the deferrable server policy (B/D(DS)/1 queue). Then, by choosing a small enough time quantum, we can closely approximate the M/D(DS)/1 model.

B/D(PS)/1 and B/D(DS)/1. Consider a single queue system where we divide the time axis into intervals of equal length. Each interval is a *slot*. Jobs arrive according to a Bernoulli process with parameter η: in each slot, either we have exactly one job arrival, with probability

η, or we have no job arrival, with probability $1-\eta$. The inter-arrival time distribution of the Bernoulli process is a geometric distribution with parameter η. The service times of jobs are deterministic, denoted as d. A B/D(DS)/1 or B/D(PS)/1 queue can be modeled by the parameters (η, d, B, P).

Job arrival, service starting, and job departure (service completion) occur only at slot boundaries. For convenience, we assume that an job arrival ori the start of a service always occurs *just after* the slot boundary, and a job departs only *just before* the slot boundary. For example, in Fig. 4.2, a job arrives at n_+ and departs at $(n+2)_-$, starting its service just after the beginning of n-th slot, running for 2 time units, and departing just before the $(n+2)$-th slot.

The server renders service according to the budget management policy, i.e., deferrable or periodic server policy. The stability condition is $U = \eta d < \frac{B}{P} = W$.

4.2 Theoretical Properties and Algorithms

In this section, we compute the stationary response time distribution of an M/D(DS)/1 queue with the parameters (λ, d', B', P'). This M/D(DS)/1 queue can be approximated by a discrete B/D(DS)/1 queue with the parameters (η, d, B, P). We first illustrate how to quantize the system and map the parameters. We then study two queues, B/D/(PS)/1 and B/D(DS)/1, and focus on their *virtual waiting time process*. If a new job happens to arrive at the queueing system at time n, the server may still have pending jobs to render; and the server has to stay in active-state for $v[n]$ time units before handling this new job. This time $v[n]$ is defined as the *virtual waiting time*.

Assuming their virtual waiting time processes (discrete stochastic processes) are $\tilde{V}_{DS}[n]$ and $\tilde{V}_{PS}[n]$, we study the virtual waiting time at the start of each period, i.e., $V_{DS}[n] = \tilde{V}_{DS}[nP]$ and $V_{PS}[n] = \tilde{V}_{PS}[nP]$. $V_{DS}[n]$ and $V_{PS}[n]$ are Markov chains. We prove an equivalence in Theorem 5 and Corollary 9: if the system is stable, the two queues always have the same virtual waiting time distribution at the start of each server period, i.e., $V_{DS}[n] = V_{PS}[n]$.

Theorem 5 indicates that if we can compute the stationary virtual waiting time distribution at the start of the server period, then for B/D(PS)/1, the result can be applied directly to the comparable B/D(DS)/1 model. Algorithm 1 leverages this feature and uses the B/D(PS)/1 model to compute the stationary virtual waiting time distribution at the start of the server period. Then, the stationary virtual waiting time distribution of B/D(DS)/1 at all slots within a server period can be computed recursively via Algorithm 2.

Finally, if the virtual waiting time of the queueing system upon one job's arrival is known, then the response time is deterministic, due to the FIFO policy and deterministic service time. Using the discrete version of the PASTA Theorem [19][8], in which "upon arrival at a station, a job observes the system as if in steady state at an arbitrary instant for the system without that job" [133], we can determine the stationary response time distribution, F_{DS}, for a B/D(DS)/1 queue.

4.2.1 Approximating the continuous M/D(DS)/1 model

As we discussed previously, we can employ a B/D(DS)/1 queue, using the Poisson Limit Theorem [62], to approximate the stationary response time distribution of an M/D(DS)/1 queue. Given an M/D(DS)/1 queue with parameters of (λ, d', P', B'), we can quantify the

[8]The discrete version is also called the BASTA Theorem – Bernoulli Arrivals See Time Average.

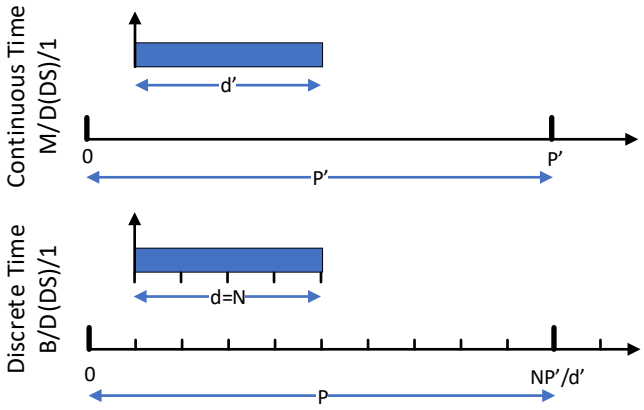


Figure 4.3: Quantization: From Continuous Time to Discrete Time

Parameter	M/D(DS)/1	B/D(DS)/1
Arrival/Succeed Rate	λ	$\eta = \lambda d'/N$
Service Duration	d'	$d = N$
Period	P'	$P = NP'/d'$
Budget	B'	$B = NB'/d'$

Table 4.1: Parameter Mapping

continuous time model by choosing a quantum (slot width) of d'/N and get a discrete time version, as shown in Fig. 4.3. Then, we can represent a correlated B/D(DS)/1 queue with the parameters (η, d, P, B) in this discrete time model. Using the invariance of utilization (i.e., $\lambda d' = \eta N$) and bandwidth (i.e., $\frac{B'}{P'} = \frac{B}{P}$), we can easily produce the mapping shown in Table 4.1. Using the Poisson Limit Theorem [62], we can approximate a Poisson process by a Bernoulli process, given large enough N. As a result, we can approximate the M/D(DS)/1 system by mapping it to a B/D(DS)/1 system with an appropriate value of N.

4.2.2 Virtual Waiting Time Equivalence at Start of Server Period

In this section, we show the equivalence of the virtual waiting times at the start of each server period for the periodic and deferrable server models, with the same parameters (B, P). Theorem 5 below can be adopted to any arbitrary arrival pattern and any service time distribution (usually noted as G/G/ arrival in Kendall's notation, where "G" stands for "General") in both continuous and discrete time domains. We denote the general deferrable and periodic server queueing systems as G/G(DS)/1 and G/G(PS)/1, respectively. Without losing generality, suppose the continuous virtual waiting time processes of the deferrable and periodic queueing systems are $\tilde{V}_{DS}(t)$ and $\tilde{V}_{PS}(t)$, respectively. The virtual waiting time sequences at each start of a server period are Markov chains, $V_{DS}[n]$ and $V_{PS}[n]$, where

$$V_{DS}[n] = \tilde{V}_{DS}(nP), \quad V_{PS}[n] = \tilde{V}_{PS}(nP), \quad n \in \mathbb{N}.$$

We denote a realization of the stochastic sequences $V_{DS}[n]$ and $V_{PS}[n]$ as $v_{DS}[n]$ and $v_{PS}[n]$, respectively.

Theorem 5. *Let the same G/G/ process arrive in both the deferrable and periodic queueing servers with the same server parameters (B, P). The virtual waiting time realizations of the two systems at the start of each period are always equal to each other, i.e.,*

$$v_{DS}[n] = v_{PS}[n].$$

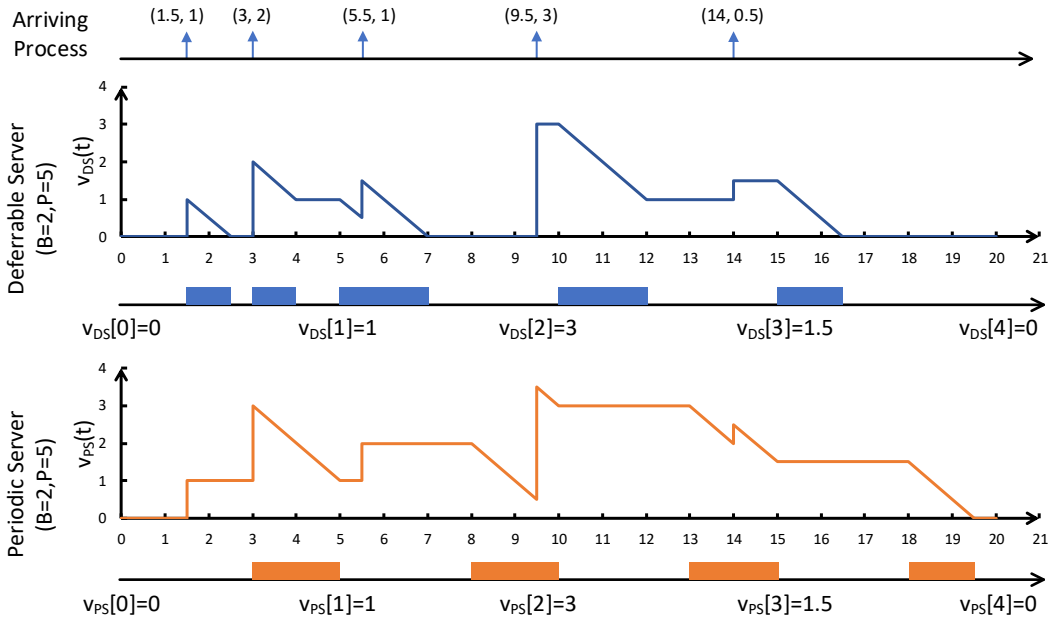


Figure 4.4: Example: How $v_{DS}[n] = v_{PS}[n]$ in a possible realization

Fig. 4.4 illustrates an example of Theorem 5: We randomly generate an arriving job sequence $\{(a_i, c_i)\}$. The i-th job releases at time a_i, with execution time c_i. We use the same process to stimulate both the deferrable server and the periodic server, whose server parameters are the same $(B = 2, P = 5)$. Then, $v_{DS}(t)$ is one possible realization of the stochastic process $V_{DS}(t)$, and $v_{PS}(t)$ is a realization of $V_{PS}(t)$. By observing the virtual waiting time at the start of each period, that is, $t = 0, 5, 10, 15, 20, ...)$, we find the sequence $v_{DS}[n] = v_{PS}[n]$, as Theorem 5 indicates.

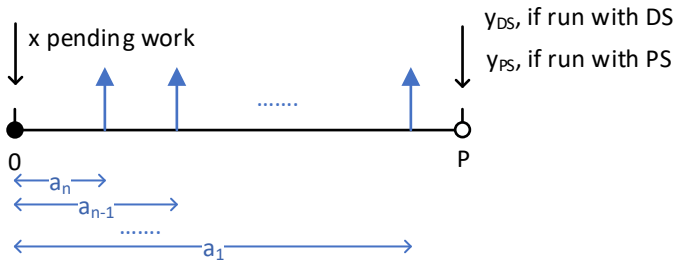


Figure 4.5: n jobs fall in a server period

To prove Theorem 5, we consider a certain server period, denoted as $[0, P)$, in Fig. 4.5. Suppose that we have x time units of work pending at the start of the period (time 0), and that exactly n jobs arrive within this period. The jobs are indexed in reverse: Their arrival times are $a_n, a_{n-1}, ..., a_1$, where $0 \leq a_n < a_{n-1} < ... < a_1 < P$. The execution times of the n jobs are $c_n, c_{n-1}, ..., c_1$, respectively.

We analyse this period with both the deferrable and periodic policies. We denote the work remaining at the end of this period (time P) as y_{DS} and y_{PS}, for deferrable server and periodic server, respectively.

Hypothesis H(n). In the described period, $\forall x \geq 0, \forall P > 0, \forall B \in [0, P]$, if exactly n jobs arrive within the period, then $y_{DS} = y_{PS}$. We can prove that $\forall n \in \mathbb{N}$, $H(n)$ is true.

Lemma 6. *H(0) is true.*

Proof of Lemma 6. $H(0)$ means no new job arrives within this period. We can consider only the x units of pending work.

If $x \geq B$, then in this period all of the budget will be consumed, whether DS or PS is used: i.e., $y_{DS} = y_{PS} = x - B$. If $x < B$, then $y_{DS} = y_{PS} = 0$. For either case, $y_{DS} = y_{PS} = max\{0, x - B\}$, so $H(0)$ is true. □

Lemma 7. *$H(n)$ is either true, or has the same logic value (a.k.a. truth value) as $H(n-1)$.*

Proof of Lemma 7. Considering the first job, a_n, we can enumerate the possible x and a_n values and evaluate each case. We will see that the system can be reduced to a system with $n - 1$ job arrivals within the new period.

Case 1. If $x \geq B$, then $H(n)$ is true.

Whether the server is deferrable or periodic, it will exhaust exactly B time units of budget for the pending job, and all incoming jobs will accumulate. As a result,

$$y_{DS} = y_{PS} = x + \sum_{i=1}^{n} c_i - B,$$

so $H(n)$ is true.

Case 2. If $x < B$, then we must consider a_n.

Case 2.1 If $x < B$ and $a_n < x$, then we evaluate c_n.

Case 2.1.1 If $x < B$, $a_n < x$, and $x + c_n \geq B$, then $H(n)$ is true. In this case, whether the server is deferrable or periodic, it will exhaust all of its budget for x and job a_n. Effectively,

$$y_{DS} = y_{PS} = x + c_n + \sum_{i=1}^{n-1} c_i - B = x + \sum_{i=1}^{n} c_i - B,$$

so $H(n)$ is true.

Case 2.1.2 If $x < B$, $a_n < x$, and $x + c_n < B$, then we first consider the deferrable server, as shown in Fig. 4.6. If $a_n < x$, then the schedule of the original system (with n arrivals) is equivalent to a system starting with a pending job of $x + c_n$, and with only $n - 1$ arriving jobs. Those two systems will yield the same busy period for the deferrable server, so the pending work at the end of the period, y_{DS}, will remain unchanged.

Similarly, we can also reduce the periodic server system to a system with $n - 1$ arrivals. Since the busy period stays the same, y_{PS} will remain unchanged. According to Case 2.1.1 and Case 2.1.2, for Case 2.1, $H(n)$ will have the same logic value as $H(n - 1)$.

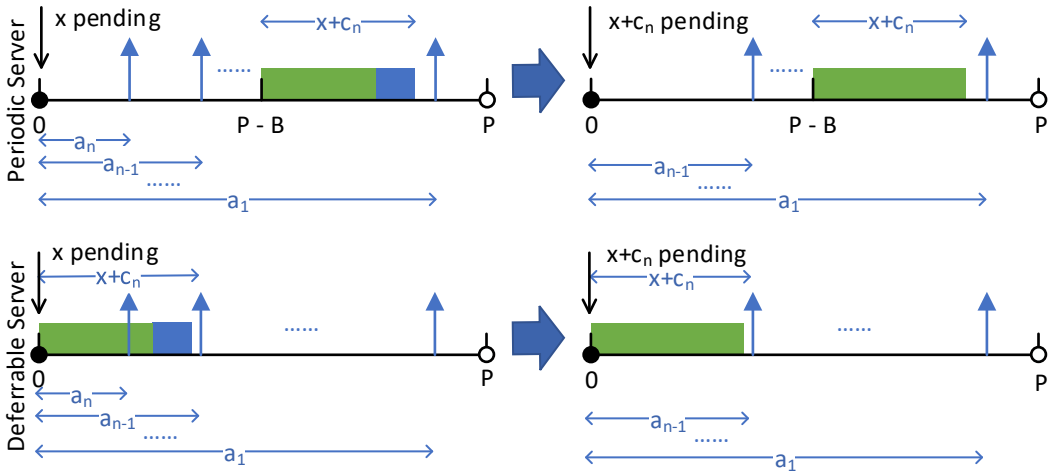
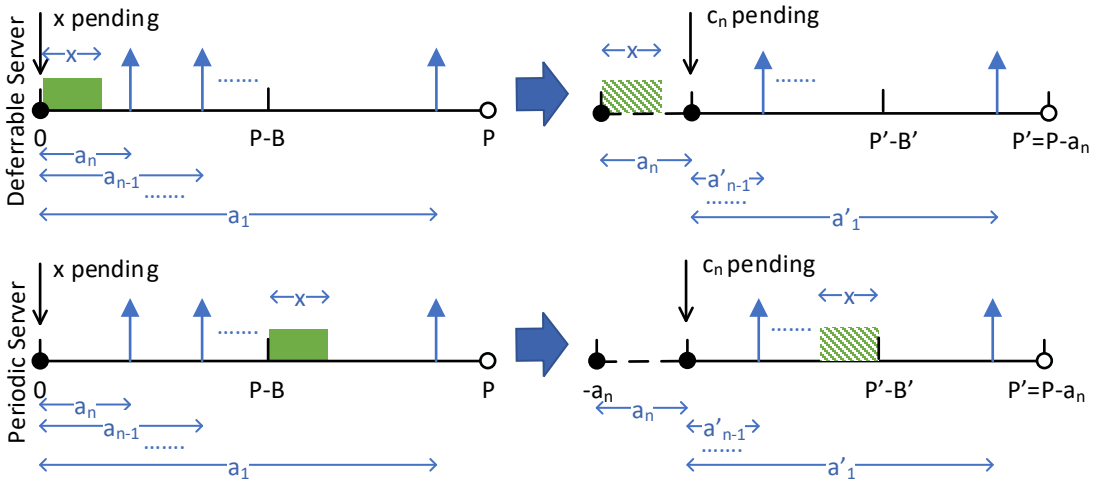


Figure 4.6: Case 2.1.2: reduce the system to $n-1$ arrivals

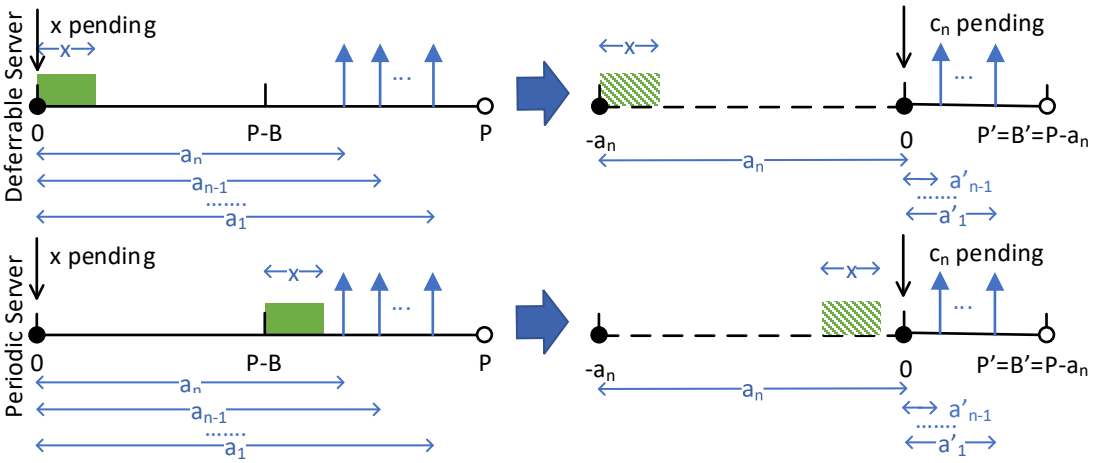
Case 2.2 If $x < B$ and $a_n \geq x$, then we consider the deferrable server first. In this case, since a_n will arrive after the pending job queue is emptied, we can reduce the system to a new system with $n-1$ arrivals with different parameters: $P' = P - a_n$, $B' = min\{B-x, P-a_n\}$, $a'_i = a_i - a_n$, and $x' = c_n$. Note that $P - a_n$ may be less than $B - x$, which means that a_n arrives so late that at least some budget has been wasted and can never be reclaimed.

Next, we consider the periodic server: As before, since a_n will arrive after the pending job queue is emptied, we can reduce the budget of the new system by x units and consider the new system that starts when a_n arrives. The original schedule of x must fall within the off-period of the new system. Hence, we reduce the original system to a new system with $n-1$ arrivals with different parameters: $P' = P-a_n$, $B' = min\{B-x, P-a_n\}$, $a'_i = a_i - a_n$, and $x' = c_n$.

As shown in Fig. 4.7, the new systems for both the deferrable server and the periodic server still satisfy the condition for $H(n-1)$: $P' \geq B$, $x' = c_n \geq 0$, and $0 \leq a_{n-1} < a_{n-1} < ... < a_1 < P$. Thus for Case 2.2, $H(n)$ has the same logic value as $H(n-1)$.



(a) $B - x \leq P - a_n$



(b) $B - x > P - a_n$

Figure 4.7: Case 2.2: reduce the system to $n - 1$ arrivals

Considering Case 2.1 and Case 2.2 as a whole, for Case 2, $H(n)$ is either true or has the same logic value as $H(n-1)$.

Considering Case 1 and Case 2, $H(n)$ has the same logic value as $H(n-1)$. □

Lemma 8. *$\forall n \in \mathbb{N}$, $H(n)$ is true.*

Proof of Lemma 8. For $n = 0$, we have proved Lemma 6. For $n > 0$, we can recursively use Lemma 7 to reduce $H(n)$ to $H(0)$, and finally to find $H(n)$ is true. □

Proof of Theorem 5. Let the system start at $t = 0$, with no initial pending job, i.e., $v_{DS}[0] = v_{PS}[0] = 0$. Using Lemma 8 recursively for each period $[kP, (k+1)P)$, we can conclude that $v_{DS}[n] = v_{PS}[n]$. □

The stationary virtual waiting times of the two Markov chains are $V_{DS}[n] \xrightarrow{p} V_{DS}$, and $V_{PS}[n] \xrightarrow{p} V_{PS}$, respectively.

Corollary 9. *Let a G/G/ process arrive in both the deferrable and periodic servers with the same server parameters (B, P). If the stability condition holds, the stationary distributions of the virtual waiting times of the two systems at the start of each period exhibit* ***almost sure equality***[9]*, i.e.:*

$$V_{DS} \overset{\text{a.s.}}{=} V_{PS}.$$

Proof of Corollary 9. Using Theorem 5, we have $v_{DS}[n] = v_{PS}[n]$. Thus, $Pr\{V_{DS}[n] = V_{PS}[n]\} = 1$. That is, $V_{DS}[n] \overset{\text{a.s.}}{=} V_{PS}[n]$. If the stability condition holds, statistical equilibrium can be achieved: $V_{DS} \overset{\text{a.s.}}{=} V_{PS}$. □

4.2.3 Conditional Stationary Virtual Waiting Time in a Period

Corollary 9 indicates that a comparable deferrable server and periodic server have the same stationary virtual waiting times at the start of a server period: $\overline{V}_{DS}|(T = 0)$ has the same

[9] In probability and statistics, *almost sure equality* is a stronger equivalence than *equality in distribution* [62].

distribution as $\overline{V}_{PS}|(T=0)$. We apply Corollary 9 on B/D/ arrival to simplify the computation of the stationary response time distribution of a B/D(DS)/1 queue.

Stationary Virtual Waiting Time of a Periodic Server. Supposing the B/D(PS)/1 queue reaches its statistical equilibrium, we denote the distribution of stationary virtual waiting times at any slot of the server period as $p_{l,n}$, where

$$p_{l,n} = Pr\{\overline{V}_{PS} = l | T = n\} \quad l, n \in \mathbb{N}. \tag{4.2}$$

Then, when the sever is in the off-state ($0 \leq n \leq P - B - 1$),

$$\begin{cases} p_{l,n+1} = (1-\eta)p_{l,n} & 0 \leq l < d, \\ p_{l,n+1} = (1-\eta)p_{l,n} + \eta p_{l-d,n} & l \geq d. \end{cases} \tag{4.3}$$

When the sever is in the on-state ($P - B \leq n \leq P$),

$$\begin{cases} p_{0,n+1} = (1-\eta)(p_{0,n} + p_{1,n}) & l = 0, \\ p_{l,n+1} = (1-\eta)p_{l+1,n} & 1 \leq l < d-1, \\ p_{l,n+1} = (1-\eta)p_{l+1,n} + \eta p_{l-d+1,n} & l \leq d-1. \end{cases} \tag{4.4}$$

Equations (4.3) and (4.4) govern the state transitions within a server period. Fig. 4.8 shows an example of the state transition within a server period of the B/D(PS)/1 queue when $d = 0$.

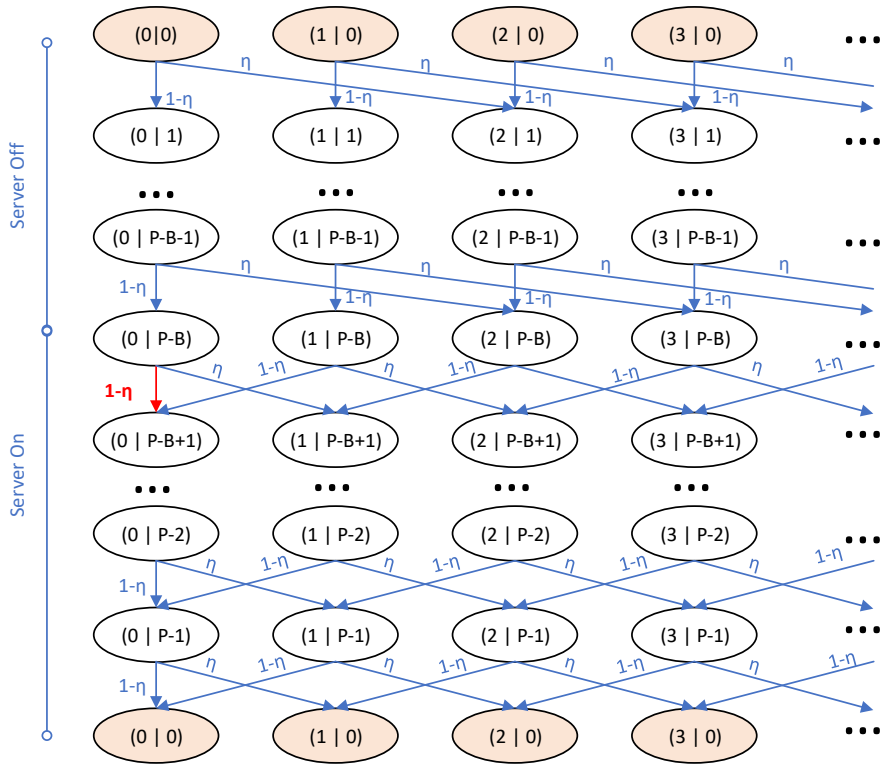


Figure 4.8: State Transition Diagram for a B/D(PS)/1 queue: $Pr\{\overline{V}_{PS} = l|T = n\}$, $d = 2$

If the statistical equilibrium has been achieved, due to the periodicity of the system, we have

$$p_{l,n+P} = p_{l,n}, \quad \forall l \in \mathbb{N}. \tag{4.5}$$

Consider the normalization condition of the conditional probability mass function:

$$\sum_{l=0}^{\infty} p_{l,n} = \sum_{l=0}^{\infty} Pr\{\overline{V}_{PS} = l|T = n\} = 1, \quad 0 \leq n \leq P. \tag{4.6}$$

Equations (4.3), (4.4), (4.5), and (4.6) form a differential equation system, and $p_{l,n}$ can be uniquely determined. However, instead of deriving a purely analytical solution for the differential system, we employ a positive convergent series, $\{p_{l,n}\}_{l=0}^{\infty}$, and use the mathematical nature of this series [72] to construct a practical algorithm.

Theorem 10. $\forall \epsilon > 0, n \in \mathbb{N}, \exists M \in \mathbb{N}, s.t. \sum_{l=M}^{\infty} p_{l,n} < \epsilon$.

Proof of Theorem 10. According to equation (4.6), the non-negative series $\{p_{l,n}\}$ is convergent. The partial summation series $\{S_m\}_{m=0}^{\infty} \to 1$, where $S_m = \sum_{l=0}^{m} p_{l,n}$. In other words, $\forall \epsilon > 0, n \in \mathbb{N}, \exists M \in \mathbb{N}, s.t.$

$$\sum_{l=0}^{M} p_{l,n} > 1 - \epsilon. \tag{4.7}$$

Using (4.6) minus (4.7), we get $\sum_{l=M}^{\infty} p_{l,n} < \epsilon$. □

Theorem 10 indicates that in practice, if we choose a large enough M, we can ignore the tail distribution of $p_{l,n}$. This feature helps us construct a practical algorithm: We focus on only the first M items, i.e., $\{p_{l,n}\}_{l=0}^{M-1}$, and treat the tails, $\{p_{l,n}\}_{l=M}^{\infty}$, as 0s.

Algorithm 1 demonstrates how we leverage the recursive structure in Fig. 4.8 to compute $\{p_{l,0}\}_{l=0}^{M-1}$: (i) we initialize only $p_{0,0} = 1$, while others are equal to 0, which represents layer $n = 0$ of Fig. 4.8; (ii) using equation (4.3), we can reach layer $n = P - B$. Using equation (4.4), we can reach layer $n = P$; (iii) because of equation (4.5), layer P is effectively layer 0. By normalizing the result of layer P, we are able to compare it to the original layer 0; (iv) we repeat the procedure until the 1-norm error is less than the threshold ($\delta < \Delta$).

Algorithm 1 also indicates how to compute $\{p_{l,n}\}_{l=0}^{M-1}, n = 1, 2, ..., P-1$: After $\{p_{l,0}\}_{l=0}^{M-1}$ converges, we can use equations (4.3) and (4.4) to compute and then normalize each layer.

Algorithm 1: Compute first M terms of $p_{l,0}$

Input: Vector Length: M, Expected Error: Δ

Output: $M \times 1$-Vector: $\overrightarrow{V} = (p_{0,0}, p_{1,0}, ..., p_{M-1,0})$

$M \times 1$-Vector: $\overrightarrow{V} \leftarrow (1, 0, 0, ...)$; $\quad \delta \leftarrow \infty$;

while $\delta \geq \Delta$ **do**

  $\overrightarrow{V'} \leftarrow \overrightarrow{V}$;

  for $i \leftarrow 0$ **to** $P - B - 1$ **do**

    $M \times 1$-Vector: $\overrightarrow{V_T} \leftarrow (0, 0, 0, ...)$;

    Compute $\overrightarrow{V_T} = (p_{0,i+1}, p_{1,i+1}, ..., p_{M-1,i+1})$ from $\overrightarrow{V} = (p_{0,i}, p_{1,i}, ..., p_{M-1,i})$, using Eq.(4.3);

    $\overrightarrow{V} \leftarrow \overrightarrow{V_T}$;

  end

  for $i \leftarrow P - B$ **to** $P - 1$ **do**

    $M \times 1$-Vector: $\overrightarrow{V_T} \leftarrow (0, 0, 0, ...)$;

    Compute $\overrightarrow{V_T} = (p_{0,i+1}, p_{1,i+1}, ..., p_{M-1,i+1})$ from $\overrightarrow{V} = (p_{0,i}, p_{1,i}, ..., p_{M-1,i})$, using Eq.(4.4);

    $\overrightarrow{V} \leftarrow \overrightarrow{V_T}$;

  end

  $\overrightarrow{V} \leftarrow \overrightarrow{V} / \| \overrightarrow{V} \|_1$; $\quad \delta \leftarrow \| \overrightarrow{V} - \overrightarrow{V'} \|_1$;

end

return $\overrightarrow{V}$;

Stationary Virtual Waiting Time of a Deferrable Server. In contrast to a periodic server, the active time slot of a deferrable server is no longer independent of the arrival process, and the remaining budget must be included in the state transition diagram. Thus, we express the conditional joint distribution of both the virtual waiting time $\tilde{V}_{DS}$ and the

remaining budget G,

$$q_{l,g,n} = Pr\{\tilde{V}_{DS} = l, G = g | T = n\}, \quad l, n \in \mathbb{N}, 0 \leq g \leq B. \tag{4.8}$$

A deferrable server replenishes its budget at the start of each period, which means the remaining budget must be B when $n = 0$. According to Theorem 5, we have

$$\begin{cases} q_{l,B,0} = p_{l,0}, & l \in \mathbb{N}, \\ q_{l,g,0} = 0, & l \in \mathbb{N}, g < B. \end{cases} \tag{4.9}$$

Now we compute the conditional joint distribution $q_{l,g,n}$ for layers $n \geq 1$. Because of the deferrable server policy and additional state variable G, the B/D(DS)/1 queue has B times the number of states as in B/D(PS)/1. Fortunately, many of those states are not reachable. Thus, instead of deriving equations corresponding to (4.3) and (4.4) for a B/D(DS)/1 system, we can use a forward recursion algorithm (Algorithm 2), which helps us determine both the effective states and the probability.

The algorithm returns a $M \times B \times P$-Matrix, A, each item of which represents $q_{l,g,n}$, and a list, `nz_list`, comprised of n sets, with each set recording the unique non-zero states of layer n.

Algorithm 2 iterates through the layers, and in each layer n, we iterate through each non-zero state. For each state, we have probability η of an incoming job and $1 - \eta$ for no incoming job. Each non-zero state can potentially transit to two different states in the next layer. We determine the state for the next layer by evaluating state parameters l and g with the deferrable policy, and finally we get the matrix and the `nz_list`.

Algorithm 2: Compute $q_{l,g,n}$

Input: $M \times 1$-Vector: $\overrightarrow{V} = (p_{0,0}, p_{1,0}, ..., p_{M-1,0})$
Output: $M \times B \times P$-Matrix: $A = \{q_{l,g,n}\}$, List: `nz_list`

```
A = zeros(M, B, P); nz_list ← [ ]; tset ← { };
for l ← 0 to M − 1 do
    A[l][B][0] ← p_{l,0};
    tset.union({(l, B)});
end
nz_list.append(tset);
for n ← 0 to P − 2 do
    iterset ← nz_list[n]; tset ← { };
    for (l, g) in iterset do
        if g > 0 then
            if l > 0 then
                A[l − 1][g − 1][n + 1] += (1 − η)A[l][g][n]; tset.union({(l − 1, g − 1)});
                if l + d − 1 ≤ M then
                    A[l + d − 1][g − 1][n + 1] += ηA[l][g][n];
                    tset.union({(l + d − 1, g − 1)});
                end
            else
                A[l][g][n + 1] += (1 − η)A[l][g][n];
                tset.union({(l, g)}); if l + d − 1 ≤ M then
                    A[l + d − 1][g − 1][n + 1] += ηA[l][g][n];
                    tset.union({(l + d − 1, g − 1)});
                end
            end
        else
            A[l][g][n + 1] += (1 − η)A[l][g][n]; tset.union({(l, g)});
            if l + d > M then
                A[l + d][g][n + 1] += ηA[l][g][n]; tset.union({(l + d, g)});
            end
        end
    end
    nz_list.append(tset);
end
return (A, nz_list);
```

How Theorem 5 Helps Reduce Complexity. One can derive a state transition diagram for B/D(DS)/1 and use a similar iteration method as in Algorithm 1 to compute the stationary distribution. However, such a method needs to iterate through each state of the deferrable server, which is $O(MP^2)$. If the outer "while" loop needs K iterations, then the overall complexity is $O(KMP^2)$. Thanks to Theorem 5, we can first compute $\{p_{l,n}\}_{l=0}^{M-1}$ by Algorithm 1 with $O(K \times M \times P)$, then explore the state space of B/D(DS)/1 only once by Algorithm 2 with $O(MP^2)$. Hence we get an overall complexity of $O(MP(K+P))$ rather than $O(KMP^2)$.

4.2.4 Determine the Stationary Response Time Distribution

Finally, we consider the stationary response time distribution, $F_{DS}(t) = Pr\{R_{DS} = t\}$, for a B/D(DS)/1 queue. Using the total probability law over the joint random variables $\overline{V}_{DS}, G, T$:

$$\begin{aligned} &F_{DS}(t) = Pr\{R_{DS} = t\} = \\ &\sum_{l=0}^{\infty}\sum_{g=0}^{B}\sum_{n=0}^{P-1} Pr\{R_{DS} = t|\overline{V}_{DS} = l, G = g, T = n\} Pr\{\overline{V}_{DS} = l, G = g, T = n\}. \end{aligned} \tag{4.10}$$

Given the values of (l, g, n), the response time can be determined directly by $f_{DS}(l, g, n)$, where

$$\begin{aligned} f_{DS}(l,g,n) &= \begin{cases} h, & h \le \gamma_1, \\ h + \gamma_2, & \gamma_1 < h \le \gamma_1 + B, \\ h + \gamma_2 + \lceil \dfrac{h - (\gamma_1 + B)}{B} \rceil (P - B), & h > \gamma_1 + B. \end{cases} \\ h &= l + d, \quad \gamma_1 = \min\{P - n, g\}, \quad \gamma_2 = P - n - \gamma_1. \end{aligned} \tag{4.11}$$

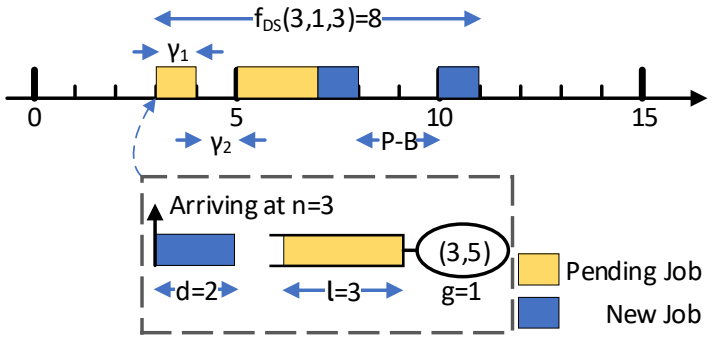


Figure 4.9: Determine the Response Time, An Example

We use an example to illustrate both the notation and the intuition of equation (4.11). Fig. 4.9 shows a B/D(DS)/1 system with $P = 5$, $B = 3$, and $d = 2$. At the third slot of a period ($n = 3$), a new incoming job arrives, and the sever has only one unit of budget remaining ($g = 1$) for current period, while it still has three-unit pending jobs ($l = 3$). We want to figure out the response time of the new incoming job, i.e., $f_{DS}(3, 1, 3)$.

The server needs to provide h units of service to finish the new incoming job. γ_1 indicates the available service units within the first period. Clearly, if the server can provide enough service time within the first period ($h \leq \gamma_1$), the response time should be h, as in the first case of equation (4.11).

Then, if the h units can be fulfilled before the end of the second period ($h < \gamma_1 + B$), we need to pay only an additional γ_2 budget replenishment penalty. Thus the response time is $h + \gamma_2$, as in the second case of equation (4.11).

If more periods are required to fulfill the h units of service time, we need to wait $P - B$ units each time, before consuming B units service from each new period.

In Fig. 4.9, the service time cannot be fulfilled within the first two periods ($h = 5 > \gamma_1 + B = 4$). We need one ($\lceil \frac{5-(1+3)}{3} \rceil = 1$) more period besides the first two, which we get by waiting for one additional budget replenishment ($P - B = 2$) . As a result, the response time is $5 + 1 + 1 \times 2 = 8$.

Since $f_{DS}(l, g, n)$ is determined, the first factor in equation (4.10), $Pr\{R_{DS} = t | \overline{V}_{DS} = l, G = g, T = n\}$, is either 1 or 0. The first factor equals 1, only if $f_{DS}(l, g, n) = t$.

To obtain the second factor in equation (4.10), using Bayes' theorem, the PASTA theorem [19], and equation (4.8), we have

$$Pr\{\overline{V}_{DS} = l, G = g, T = n\} = \frac{q_{l,g,n}}{P}. \tag{4.12}$$

Finally, the response time distribution of B/D(DS)/1 can be represented as

$$F_{DS}(t) = \frac{1}{P} \sum_{\substack{l \geq 0, 0 \leq g \leq B, 0 \leq n \leq P-1, \\ f_{DS}(l,g,n) = t}} q_{l,g,n}. \tag{4.13}$$

4.2.5 Summary of the Numerical Method

We now summarize how to compute the stationary response time distribution of an M/D(DS)/1 queue with the parameters (λ, d', B', P'), as follows:

Given an M/D(DS)/1 system: (λ, d', B', P'),

1. Choose N, using Table 4.1 for a B/D(DS)/1 system (η, d, B, P);

2. Use Algorithm 1 to compute the stationary virtual waiting time distribution at the start of a server period ($q_{l,B,0}$ of Equation (4.9));

3. Use Algorithm 2 to compute the conditional stationary virtual waiting time distribution at each slot within a period ($\{q_{l,g,n}\}$) recursively;

4. Compute the stationary response time distribution $F_{DS}(t)$ via Equation (4.13). □

4.3 Configuration to meet SLOs

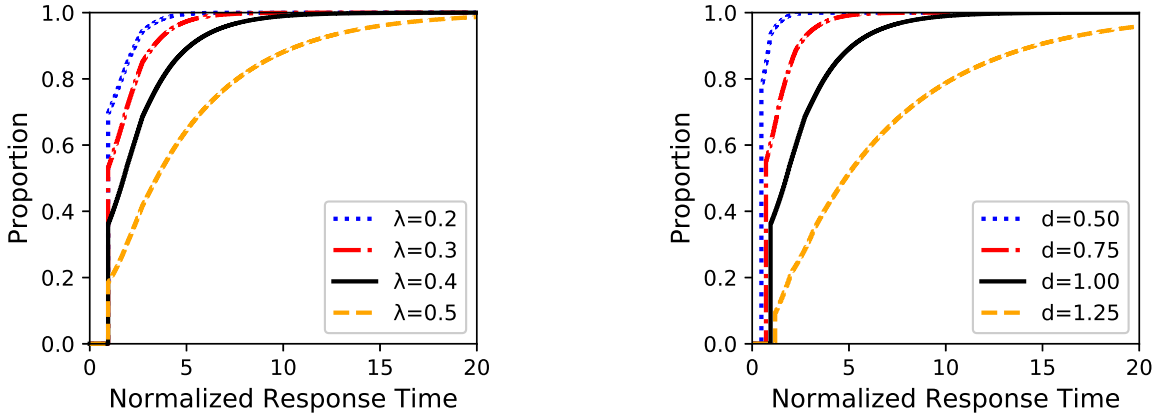
In this section, we apply our algorithm as a design tool, showing how the computed stationary response time distribution can be used to meet the service level objective (SLO) of a time-sensitive service.

Latency SLO Validation. A time-sensitive service may have a required tail latency performance: e.g., a 90^{th} percentile latency of less than $10ms$. This latency objective can be represented as a point $(d_0 = 10ms, p = 0.90)$ on the same coordinate system as a stationary response distribution whose x-axis is latency and y-axis is cumulative proportion. For an M/D(DS)/1 system with the parameters (λ, d, B, P), we can uniquely determine the CDF of its stationary response time. This capability makes SLO validation trivial: if the point (d_0, p) falls below the CDF curve, then the SLO requirement will be met; otherwise, the SLO cannot be met. The single-point latency objective can also be generalized to a multi-point latency objective and even to a lower-bound CDF curve objective.

Parameter Space Exploration. Computing the stationary response time distribution enables us to conduct both qualitative and quantitative analysis for an M/D(DS)/1 queue. As shown in Fig. 4.10, to illustrate the impact of each parameter, we change only one parameter while keeping the others unchanged. We plot the M/D/1 response time distribution [38] as a reference when sweeping W and P. We make three observations: (1) When λ and d increase, the utilization $U = \lambda d$ increases. As a result, the response time performance becomes worse (Fig. 4.10a and 4.10b). (2) As shown in Fig. 4.10c, when λ, d, and P are constant, the larger the budget (bandwidth), the better the response time. When $B = P$ and hence $W = 100\%$, the M/D(DS)/1 model degenerates to an M/D/1 model. Indeed, since the server of the M/D(DS)/1 queueing model is subject to a resource constrained policy (deferrable server with the parameters (B, P)), we can immediately conclude that the response time distribution of M/D(DS)/1 is upper bounded by that of the M/D/1 queue. (3) As shown in Fig. 4.10d, when λ, d and $W = \frac{B}{P}$ are constant, the larger the period (and the budget, proportionally), the more the deferrable server can tolerate the burstiness introduced by stochastic arrivals. As a result, we get better response time performance. If $P \to \infty$, the M/D(DS)/1 system degenerates to an M/D/1 model.

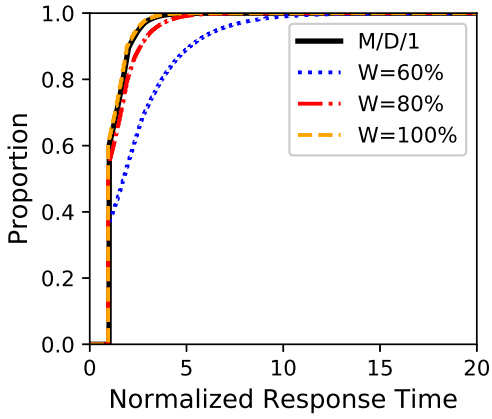
Parameter Selection. In practice, the job arrival rate (λ) and service duration (d) usually cannot be easily adjusted, while the server period (P) and budget (B) can be configured to meet a tail latency requirement, noted as (d_0, p). With the M/D(DS)/1 stationary response time distribution, parameter selection is straightforward, as follows:

(1) SLO feasibility: The response time distribution of the M/D/1 provide an upper bound of the M/D(DS)/1 response time distribution. Only if $B = P$ can the M/D(DS)/1 achieve the same performance of the M/D/1 queue. If the SLO (d_0, p) falls below the M/D/1 response

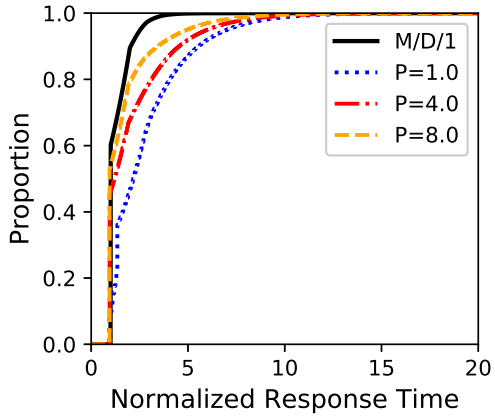


(a) Sweep λ

(b) Sweep d



(c) Sweep W

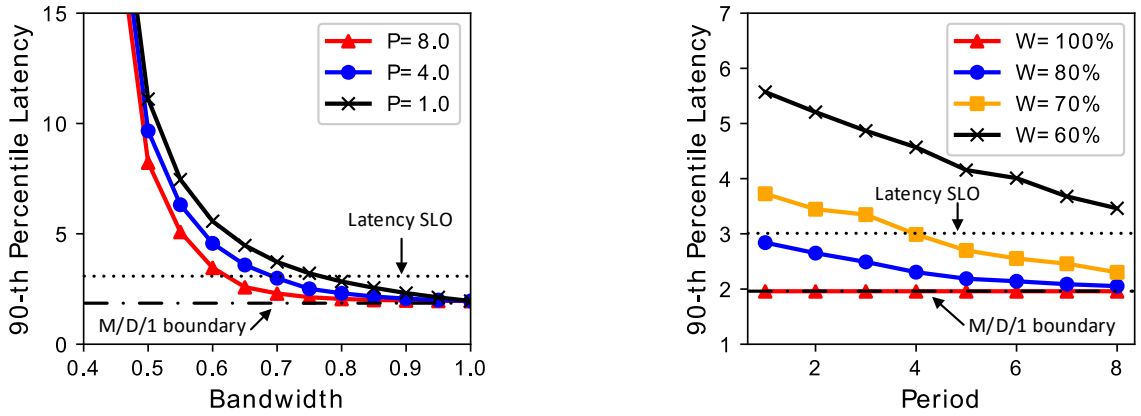


(d) Constant W, Sweep P

Figure 4.10: Response Time Distribution and Parameter Sweep, Base Condition $\lambda = 0.4, d = 1.0, P = 2.0, B = 1.2$

time distribution, we may be able to manipulate (B, P) of an M/D(DS)/1 queue to achieve the goal. Otherwise, the SLO is unachievable.

(2) Selecting (B, P): Suppose the SLO is achievable, and we want to provide a 90^{th} percentile latency of less than 3, i.e., a SLO $(d_0 = 3, p = 0.90)$ with an arrival rate of $\lambda = 0.4$ and a constant service duration $d = 1.0$. We can directly plot the curves as in Fig. 4.11 and focus on how the 90^{th} percentile latency changes versus W and P. By choosing a proper



(a) Choosing W

(b) Choosing P

Figure 4.11: Parameter Selection, with $\lambda = 0.4, d = 1.0$

bandwidth value W and period value P (so that $B = PW$ is effectively determined), the desired latency can be made less than the goal $d_0 = 3$ defined in the SLO.

Specifically, if overprovisioning by increasing the server bandwidth is acceptable, a drastic decrease in the tail latency can be achieved (Fig. 4.11a). For example, choosing $P = 4.0$, any $W \geq 0.7$ can satisfy the latency SLO. Latency also can be improved by keeping the same bandwidth and increasing the period and budget proportionally (Fig. 4.11b). For example, by choosing $W = 0.7$, any $P \geq 4.0$ can satisfy the latency SLO.

Again, none of these methods can yield a better latency than the M/D/1 limit: In Fig. 4.11a, all the curves (with different P) cross when $W = 100\%$ (i.e., $P = B$), and hence the system degenerates to an M/D/1 system and reaches the M/D/1 boundary. Similarly, in Fig. 4.11b, the curve $W = 100\%$ overlaps with the M/D/1 boundary, and other curves converge to the M/D/1 boundary when $P \to \infty$. However, a large P implies that a time-sensitive service may potentially monopolize the CPU for a long time. As a result, it will have negative impacts on general services which share the same CPU. Our analysis allows operators to select the period and budget to meet the tail latency without a unnecessarily large period.

4.4 Evaluation

In this section, we evaluate our algorithm as a predictive tool. By comparing our numerical results with empirical results on real systems, we can assess how closely the numerical prediction can approximate the empirical results.

We conducted experiments on a machine with one Intel E5-2683v4 16-core CPU and 64 GB memory. We disabled hyper-threading and power saving features and fixed the CPU frequency at 2.1 GHz to reduce unpredictability, as in [64, 141, 138]. We ran time-critical services and general services in Linux virtual machines (VMs) on the Xen 4.10.0 hypervisor. We modified the *Real-Time Deferrable Server* (RTDS) to support a partitioned fixed priority scheduling policy[10] [138, 139]. The modified RTDS scheduler treated each VCPU as a deferrable server with three parameters: budget, period, and priority. We used Linux 4.4.19 for all VMs. We configured Domain 0 with one full CPU pinned to one dedicated core, i.e., PCPU 0.

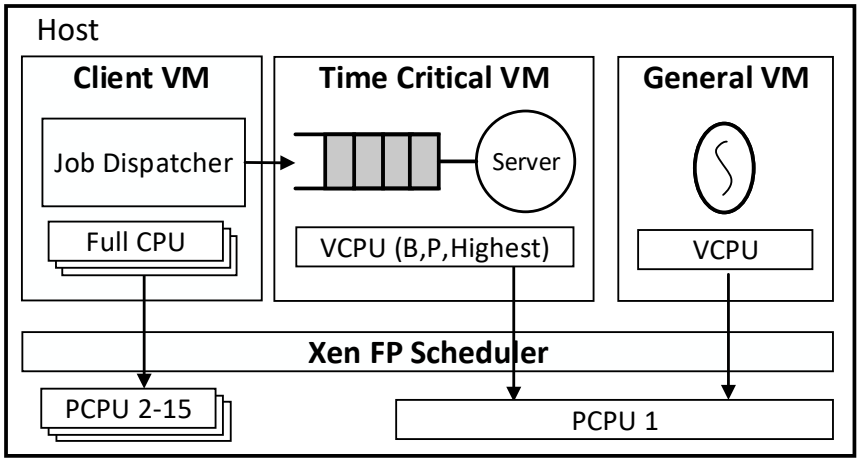


Figure 4.12: M/D(DS)/1 System for Evaluation

[10]The vanilla RTDS is a gEDF scheduler.

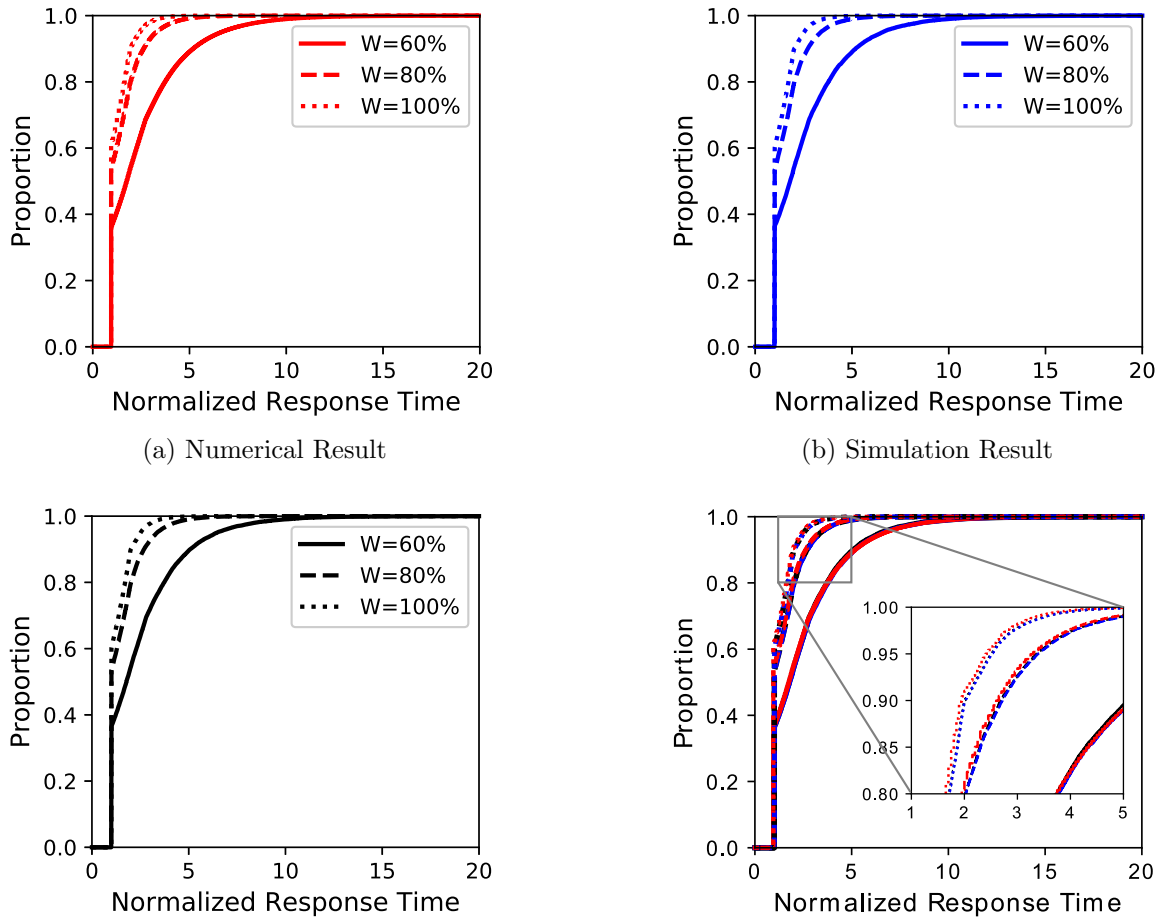
As shown in Fig. 4.12, the time-critical service ran on a single VCPU virtual machine, which had highest priority. The VCPU shared the same PCPU1 with another VCPU which was running a "CPU-Hog" process in a general VM. We used another 14-VCPU VM to run clients that dispatched jobs to the service under test.

We used two different services for the experiments: a synthetic server and a Redis [111] server. The synthetic server is designed to have a predictable execution time with litter variance. The Redis server worked as a representative time-critical service commonly used in cloud environments.

Synthetic Server. We used four parameters, (λ, d, B, P), to represent an M/D(DS)/1 system. For each configuration of the system, we measured the response time distribution in three different ways: (1) We directly computed the response time distribution via our algorithm. (2) We randomly generated $20,000$ samples following a Poisson process with a rate of λ, and then simulated the system behavior to get the response time distribution. (3) Using the same Poisson process, we let the job dispatcher stimulate the time-critical service on the testbed as shown in Fig. 4.12, then measured the empirical response time distribution. We expected that those three approaches would produce similar results.

We let the arrival rate be $\lambda = 0.004\ event/ms$, and the constant service duration be $d = 100ms$. We first fixed the period $P = 200ms$, while we varied B over $\{120ms, 160ms, 200ms\}$, corresponding to a bandwidth W of $\{60\%, 80\%, 100\%\}$. We chose $N = 20$ for our algorithm.

Fig. 4.13a, 4.13b, and 4.13c show the numerical, simulation, and empirical results. Fig. 4.13d is the superposition of the first three figures, and it supports two observations. (1) The numerical results, simulation results, and empirical results closely approximate each other. The fact that our numerical results approximate the simulated ones validates the correctness



(a) Numerical Result

(b) Simulation Result

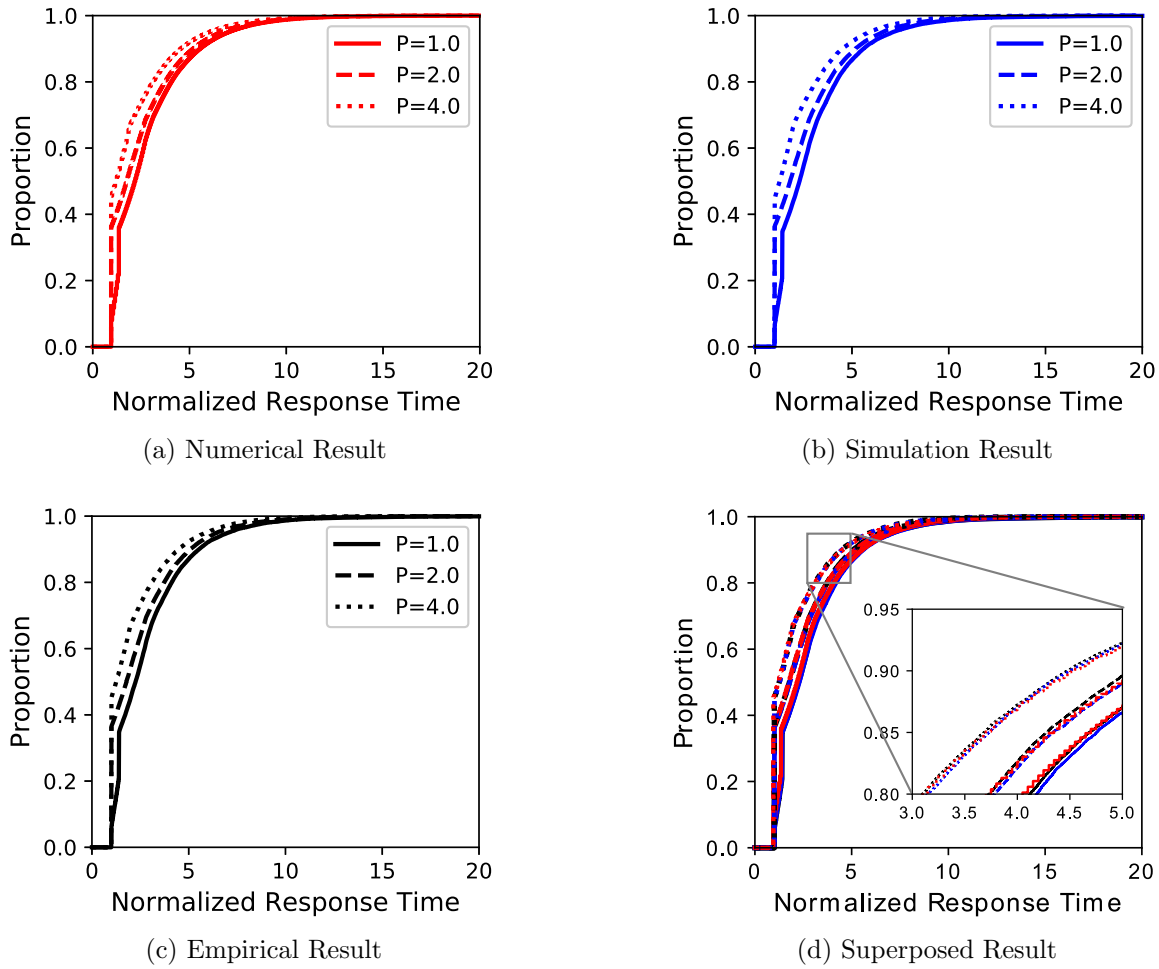
(c) Empirical Result

(d) Superposed Result

Figure 4.13: Response Time Distribution of Synthetic Server, Fixed $P = 200ms$, Results Normalized against $d = 100ms$

of our numerical algorithm. Both results also approximate the empirical one, indicating that our system and synthetic server implementation fit the M/D(DS)/1 model. (2) For the same period value, the larger the bandwidth, the better the response time distribution.

Keeping the same arrival rate and service duration, we then fixed the bandwidth $W = 60\%$, while varying P (and B proportionally) over $\{100ms, 200ms, 400ms\}$.



(a) Numerical Result

(b) Simulation Result

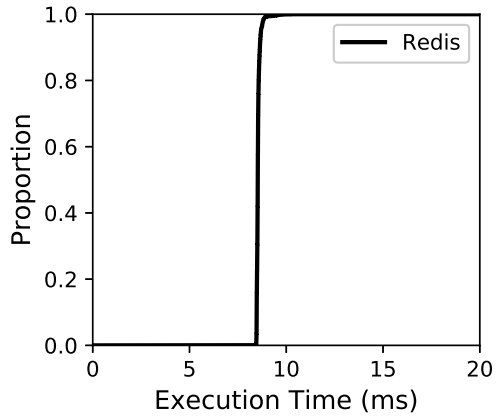
(c) Empirical Result

(d) Superposed Result

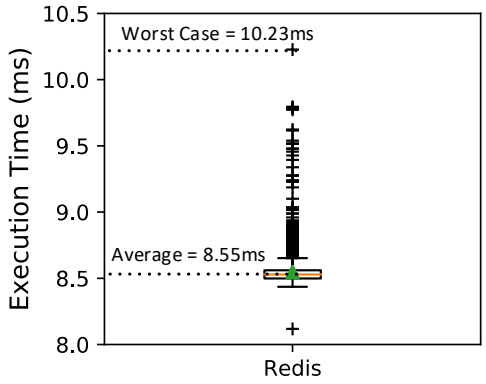
Figure 4.14: Response Time Distribution of Synthetic Server, Fixed $W = 60\%$, Results Normalized against $d = 100ms$

As shown in Fig. 4.14, these results support two observations: (1) The numerical, simulation, and empirical results closely approximate each other. (2) Given the same bandwidth value, the larger the period, the better the response time distribution.

Redis. Being a single-threaded in-memory data storage server, Redis is typically deployed as a micro-service or in a virtualized host, as in AWS ElastiCache for Redis [47]. We used Redis as a time-sensitive workload to verify whether our approach can effectively predict the stationary response time distribution for a real-world application.

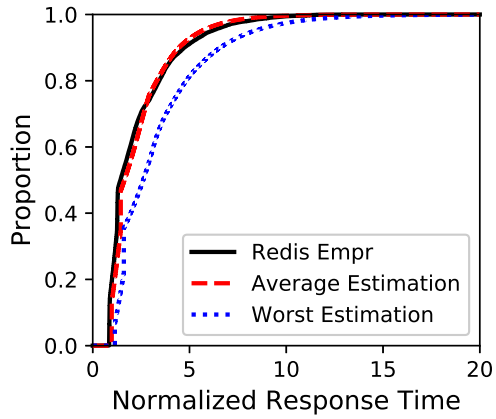


(a) CDF

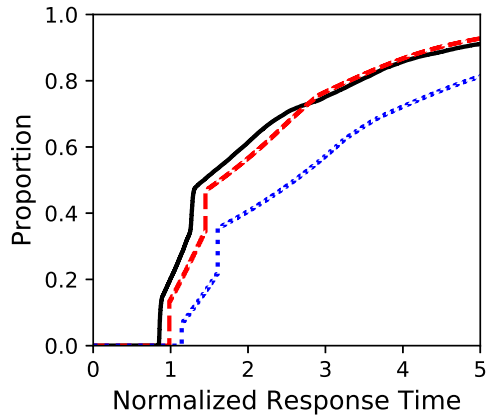


(b) Boxplot

Figure 4.15: Redis Execution Time Distribution



(a) Zoomed Out



(b) Zoomed In

Figure 4.16: Response Time Distribution of Redis, $P = 10ms$, $B = 6ms$, Normalized against $d = 8.55ms$

We used the Redis server in a time-sensitive VM as shown in Fig. 4.12. We chose the "SORT" query, which sorts a 20,000-item key-value based list. Unlike the synthetic server, whose job execution times can be calibrated and fine-tuned, we measured the execution time for Redis empirically. Fig. 4.15 shows the execution time distribution of the Redis "SORT" query. Though the service time is relatively predictable (Fig. 4.15a), but with non-negligible variance (Fig. 4.15b): The average execution time and worst case execution

time were $8.55ms$ and $10.23ms$, respectively. We used the average time for predicting an approximated distribution, and the worst case value for predicting the lower bound of the response time distribution.

We stimulated the system with a job arrival rate of $\lambda = 0.004\ event/ms$, setting the deferrable server period $P = 10ms$ and $B = 6ms$. We estimated the response time distribution by using an M/D(DS)/1 model with our numerical analysis. First, using the average execution time for Redis ($d = 8.55ms$) as an approximation of the deterministic service duration in an M/D(DS)/1 system, we computed the response distribution by using our numerical approach. Second, using the worst case execution time ($d = 10.23ms$), we derived a lower bound of the response time distribution.

As shown in Fig. 4.16, we observed that the response time distribution, when using the average Redis execution time ($d = 8.55ms$) as the deterministic execution duration, approximated the empirical result closely. However, due to the variability of the Redis execution time distribution (Fig. 4.15b), the head of the empirical distribution is better than estimated, while the tail portion is worse. In comparison, when using the worst case execution time for estimation, the estimated response time distribution provided a lower bound of the empirical distribution.

4.5 Related Work

Kaczynski, Lo Bello, and Nolte [58] extended the SAF model [32] by allowing aperiodic tasks to run within polling servers. The objectives of our work and their work are different. On one hand, the extended SAF model was not designed for deriving the exact response time

distribution of aperiodic tasks, which is the main objective of our work. On the other hand, another aspect of the extended SAF model is more general: It allows arbitrary arriving and arbitrary execution for aperiodic task, by using *Arrival Profile* and *Execution Time Profile (ETP)*, and running them within a polling server with any priority, while our system allows the aperiodic task to be a Poisson arrival with deterministic execution, running within a highest priority deferrable server. Unfortunately, given the difference between the deferrable server and the polling server, the ETP extraction cannot be directly adopted on a deferrable server.

Queueing systems with periodic service have been studied since 1956 [100]. Researchers encountered mathematical difficulties when trying purely analytical techniques to study virtual waiting time and response time distributions on continuous time domain models [115, 104, 103]. Eenige [35] focused on discrete time queueing systems with periodic services. He observed the mathematical difficulties in deriving a pure analytical method even on a simplistic B/D(PS)/1 queue whose service time equals one time slot, i.e., $d = 1$. As a result, Eenige employed numerical methods for calculating the virtual waiting time distribution for a discrete queueing system with periodic service. Inspired by the queueing model with periodic service and virtual waiting time analysis, here we first observe the virtual waiting time equivalence between the periodic server and the deferrable server. Combining scheduling analysis and virtual waiting time analysis, we derive an efficient method to compute the stationary response time distributions of M/D(DS)/1 models.

4.6 Conclusions

The proliferation of time-critical services in cloud computing has emphasized the importance of performance isolation. In this chapter we consider a time-critical service scheduled as a deferrable server for performance isolation, and we develop a numerical method for computing the stationary response time distribution of a stochastic Poisson arrival process. The numerical method takes advantage of the equivalence of virtual waiting times between periodic and deferrable server. Knowing the stationary response time distribution of the M/D(DS)/1 queue, we demonstrated how the method can enable an operator to meet the latency SLO of a time-critical service. We implemented a prototype testbed based on Xen 4.10.0 hypervisor and evaluated two case studies involving both a synthetic service and the commonly used Redis service. The results of those studies demonstrated the accuracy of our approach as a predictive tool in supporting time-sensitive services in practical real-world settings. In the future, we aim to extend the system model by allowing arbitrary job execution times with a known distribution, i.e., an M/G(DS)/1 queue .

Chapter 5

RT-ZooKeeper: Taming the Recovery Latency of a Coordination Service

Fault-tolerant coordination services have been widely used to develop distributed applications in cloud environments. In this chapter, we propose RT-ZooKeeper, a coordination service features fast and time-bounded recovery. In Section 5.1, we review the ZooKeeper [51] architecture and the basic concept of ZAB [57] protocol. In Section 5.2, we study each recovery phase in ZAB protocol, addressing the limitation and presenting the solution in RT-ZooKeeper. Thereafter, in Section 5.3, we establish a timing model for our the design to bound the recovery time for RT-ZooKeeper on edge.. In Section 5.4, we demonstrate the micro benchmark of RT-ZooKeeper. Moreover, we conduct a real case study with Kafka in Section 5.5, demonstrating the benefit of the RT-ZooKeeper.

5.1 Overview Of ZooKeeper

5.1.1 ZooKeeper

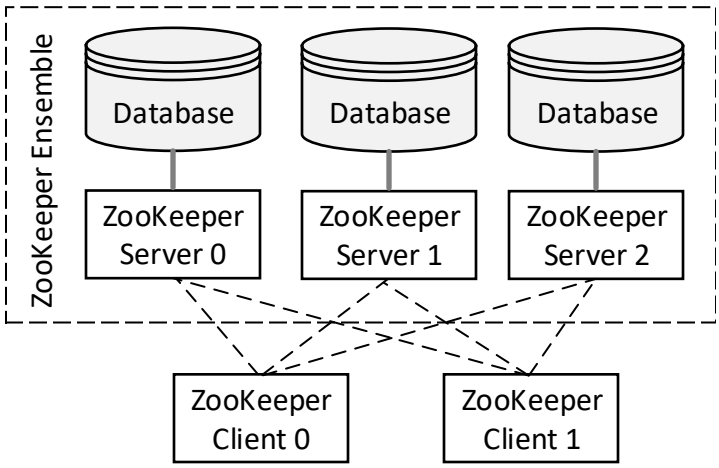


Figure 5.1: ZooKeeper: Ensemble, Servers, and Clients

Fig.5.1 illustrates the ZooKeeper service, which achieves fault tolerance by duplicating its in-memory database; each replica is handled by a *ZooKeeper server*. ZooKeeper maintains data synchronization among its replicas, and distributed applications can use *ZooKeeper clients* to access the database.

Failure Model. Each ZooKeeper server follows the crash-recovery model [57]: a server can suffer from a *fail-stop failure* and crash at arbitrary time, and then can recover after and additional interval. A ZooKeeper *ensemble* comprises N ZooKeeper servers $\Gamma = \{p_0, p_1, ..., p_{N-1}\}$. A *quorum* of Γ is a subset $Q \subseteq \Gamma$, where $|Q| > \frac{N}{2}$. Any two quorums have a non-empty intersection. The ZooKeeper service is available as long as a quorum (majority) of the ensemble is available. The number of servers in an ensemble is often an odd number, $N = 2f + 1$, which means f server crashes can be tolerated in this N-server ensemble.

Reliable and Fully Connected Network. Each ZooKeeper server is a node of a *complete graph*: any two servers can communicate with each other. The communication channels are *reliable* [57, 36]: messages sent by a correct server are never permanently lost and are received by all correct destinations in the order sent.

Primary-Backup. One of the servers works as the primary, referred to as the *leader* in the quorum, while the others are backup servers called *followers*.

Clients Feature Automatic Re-connection. By default, each client knows all the "entrances", i.e., server IP addresses and ports, of all ZooKeeper servers. Once a client connects to the ZooKeeper service, a ZooKeeper server will establish a *session* for the client and persist the session Id among all ZooKeeper servers. When a connection fails (due to a ZooKeeper server crash), the session will be available for a while; the client will automatically try an alternative entrance for the ZooKeeper service before the session expires.

Write Requests are Logged as Transactions. A client can read from and write to the database by sending requests to a ZooKeeper server. Any ZooKeeper server, regardless its role (as a leader or a follower), can handle read requests locally without forwarding the request to the whole network. However, a write request must be forwarded to the leader. The leader is responsible for updating and synchronising the database among the ZooKeeper ensemble. Each change and update is logged as a *transaction* associated with a *transaction identifier*, z. The transaction identifier, z, is a two-tuple (e, c): e is an *epoch* number, distinguishing the leader, as a ZooKeeper ensemble may change its leader during its lifetime; c is a unique *sequence* number assigned to a particular committed transaction during epoch e. Whenever a new leader starts an new epoch, the sequence number c is reset to 0. Transactions can be ordered by z. Given two different transactions, $z = (e, c)$ and $z' = (e', c')$, we define $z \prec z'$, if $e < e'$, or $e = e' \wedge c < c'$.

5.1.2 ZooKeeper Atomic Broadcast (ZAB) Protocol

ZAB is a crash-recovery atomic broadcast algorithm [57]. It is the most fundamental part of the ZooKeeper coordination service. ZAB helps the ensemble to maintain a mutually consistent state, especially when recovering from crashes. Although none of the Apache ZooKeeper implementations strictly follows the ZAB formulation that originally reported [95], in this paper we mainly focus on the latest mainstream protocol ZAB-1.0 [112] and analyse it based on its system-level implementation in ZooKeeper version 3.5.8.

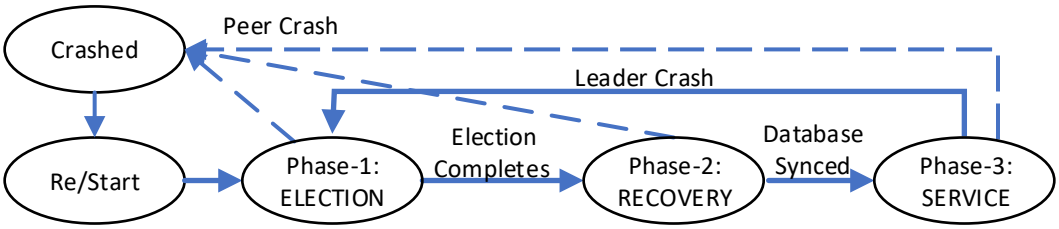


Figure 5.2: ZAB: Phase Transition Diagram

Fig. 5.2 shows the three-phase transition diagram of each ZooKeeper server. The ultimate goal of each ZooKeeper server is to reach the phase-3, SERVICE: only a ZooKeeper server in the SERVICE phase allows connections from clients and processes read and write requests.

Phase-1: ELECTION. When a server starts (or its leader is lost), it enters the ELECTION phase, running the leader election algorithm, and finally determining itself as either a follower or the new leader: the server with the latest (largest) transaction id, z, is elected as the new leader. The elected leader always starts a new *epoch*. The new epoch's value e, is always larger than any older epoch number. The leader election can converge as long as a quorum of peers alive.

Phase-2: RECOVERY. After completing ELECTION phase successfully, the leader executes a recovery protocol to ensure the followers' views of the database are synchronized with the leader's. When a peer's database has been synchronized in the RECOVERY phase, it then enters the SERVICE phase and allows clients to connect.

Service Recovery. A peer can crash at any arbitrary time in any phase. Usually, a peer's crash will not affect the availability of the entire service, except in two cases: either the leader crashes, or a follower crashes resulting in the number of live servers being less than $\frac{N}{2}$, such that no quorum can be achieved. In either case, all live peers transition to the ELECTION phase. In this paper, we focus on the case where the leader fails, and study the recovery procedure, model recovery latency, and optimize the recovery procedure.

5.2 Limitations and Solutions

To analyze the recovery time for a ZooKeeper service when leader failure occurs, it is crucial to understand ZooKeeper's internal design. In this section, we first state additional assumptions to scope our discussion to factors relevant to real-time distributed applications running in edge cloud environments. We then explore ZooKeeper's policies and mechanisms for leader election and recovery, and identify issues that may increase latency. We propose different protocols to address those limitations.

5.2.1 Additional Assumptions

The generic crash-recovery model for ZooKeeper is too loose to be useful for real-time distributed applications: crashes and recoveries can occur at arbitrary times for any server,

and it is possible that more than half of the servers fail at once, resulting in an unavailable service, whose recovery time cannot be modeled since there are no constraints on when the ensemble can reach a quorum again; also, ZooKeeper may be deployed in a public cloud, where communication latency can be unacceptably long, or even unbounded.

Thus, we make the following additional practical and reasonable assumptions to allow modeling of the latency of leader election and recovery:

- The quorum never fails, i.e., at any time, at least $f + 1$ alive servers are alive, making recovery possible.
- The new elected leader will not fail before the current recovery round finishes (i.e., reaching the phase-3).
- The message transmission latency can be bounded by d.
- Each message can be processed in a bounded time h.

Symbol	Definition
T	Overall recovery latency for RTZK
T_e	Phase-1 latency for RTZK
T_o	latency for running OptFloodMax algorithm
T_r	Phase-2 latency for RTZK
w	`finalizeWait` value
j	jitter of leader failure detection latency
d	Upper-bound of message transmission latency
h	Message processing latency
h_r	Message processing latency for phase-2 of RTZK
b	Disk I/O latency
s_e, s_r	Initialization latency for phase-1/phase-2

Table 5.1: Symbol Definition for the Timing Model

The first assumption is very practical: if the probability that more than half of the servers crash is non-negligible, then the fault-tolerance plan itself is flawed, such that the system

needs either more replicas (larger ensembles) with more stable hardware and more thoroughly validated software, or even a totally different fail-over strategy.

The second assumption confines the discussion to "single shot leader failure", avoiding unlikely, unpractical, or adversarial cases such as: "consecutive leader failures before any one finishes the recovery" or "live-lock of leader failure".

The third and fourth assumptions are reasonable for edge platforms, where servers are usually physically co-located and connected by high-speed LAN. Moreover, latency targets are usually soft and achieved with empirical approaches, as hard guarantees are typically unnecessary or overly pessimistic for such systems. It is practical to configure such systems to achieve bounded delays at high probability.

5.2.2 Phase-1: Leader Election

ZooKeeper Fast Leader Election

ZooKeeper's default leader election algorithm is *FastLeaderElection (FLE)*. We study the *FLE* algorithm and realize it is an implementation of the well-known *OptFloodMax* algorithm which is originally designed for synchronous networks [91, 92]. The FLE adapts OptFloodMax for asynchronous networks, with the "simulate purely asynchronously" approach.

This approach, however, needs a termination condition [93]. The FLE algorithm leverages a timeout mechanism for servers to make the final decision of electing the leader. If a server

does not receive any vote for w milliseconds, it will deduce the leader based on current acquired votes. The election termination timeout value, w, is a fixed variable `finalizeWait` in the ZooKeeper implementation.

In ZooKeeper, each server governs an atomic variable l, `logicalclock`, to determine the election epoch. This variable is initialized by `currentEpoch` at server's restart, and is increased when a peer goes into the `lookForLeader()` method for a new election.

Each server maintains a vote, $v = (z, i)$, a tuple consisting of the latest committed transaction id (z) of the server and the server's id (i). Given two different votes, $v = (z, i)$ and $v' = (z', i')$, we determine $v \prec v'$, if $z \prec z'$, or $z = z' \wedge i > i'$.

The FLE algorithm finally chooses the one with the largest v (effectively the one with the largest z) as the leader. Each server in the same election epoch executes the **FLE Algorithm** as follows:

1. Broadcast Initial Vote. The server broadcasts its vote v, along with the election epoch l.

2. Election Epoch Validation. Upon receiving a vote v', the server first check if the election epoch of the received vote is right: $l' = l$. If $l' < l$, the vote will be ignored; if $l' > l$, the server will update its election epoch to l' and rebroadcast its vote.

3. Flood the Maximum Vote. if $v \prec v'$, the server updates its vote: $v \leftarrow v'$, then re-broadcasts the updated vote, v'. If $v = v'$, the server tallies the vote. Otherwise, the server simply ignores the incoming vote.

4. Deduce the Leader. If the server does not receive any vote for $w =$`finalizeWait` ms, the server tries to determine its role: If the tally of current vote does not reach a quorum,

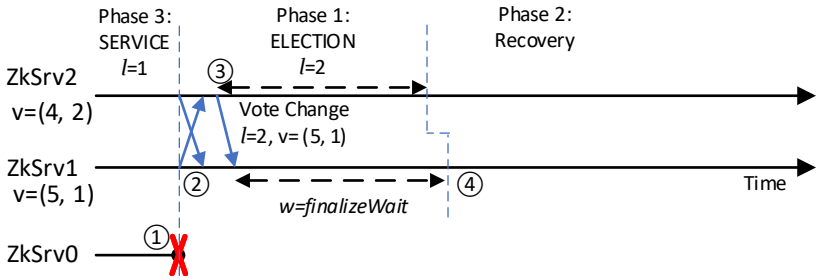
the server re-broadcasts the current vote v. If the tally of current votes $v = (z, i)$ has reached a quorum, the server determines server i to be the leader, while the others are followers. □

Limitations and Optimization

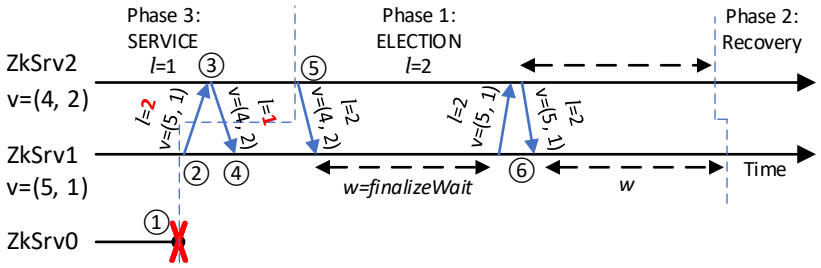
If all the peers can get into the phase-1 election simultaneously after detecting old leader's failure, we can claim that the FLE algorithm only need **one** `finalizeWait` round to converge. However, due to the asynchronously network's nature, each server cannot enter phase-1 at the same time, thus leading to unavoidable jitters. Based on the ZooKeeper system-level design and micro-benchmarks, we identified that the current FLE protocol adopts sub-optimal solutions to cover such cases, which may result in significant latency penalties: e.g., servers had to wait for **multiple** rounds before termination – a phenomenon that can be seen in the latency distribution of "vanilla ZooKeeper phase-1" in Fig. 5.8b.

When the old leader fails, the followers are supposed to transition from phase-3 (SERVICE) to phase-1 (ELECTION) simultaneously, starting a new election epoch with $l = 2$. As shown in Fig. 5.3a: ① The old leader dies. ② The two followers starts broadcasting their initial votes. ③ The ZkSrv2 receives a vote (5,1) which is "greater" than (4,2), changing its vote to (5,1) and broadcasting it again. Both ZkSrv1 and ZkSrv2 reach a quorum for (5,1). ④ After w ms, without receiving new message, they both finish the election and transit to phase-2.

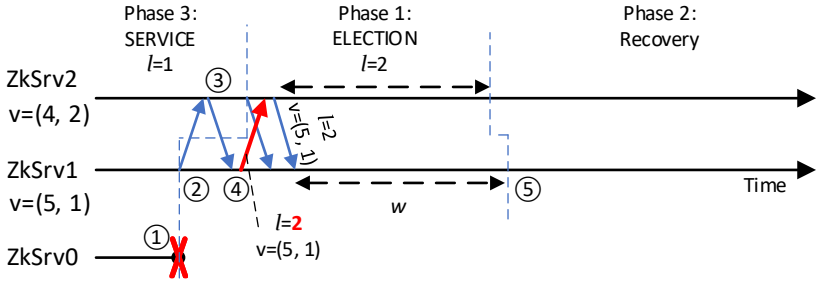
However, the followers do not always detect the leader failure simultaneously. As a result, the peers may think they are in different election epochs.



(a) Assumed Case: Peers detect the failure simultaneously



(b) Anomalous Case: Missing notification



(c) FCLE: Resend the notification

Figure 5.3: Different convergence cases for leader election

As shown in Fig.5.3b: ① ZkSrv1 detects the leader failure earlier than ZkSrv2 and enters phase-1 with new election epoch $l = 2$. ② ZkSrv1 broadcasts its initial vote (5,1) to ZkSrv2, along with the $l = 2$. ③ However, since ZkSrv2 is still in phase-3, it just sent a notification and claims it still follows the old leader ZkSrv0 by including $l = 1$. ④ However, ZkSrv1 just ignores the notification since it has a higher `logicalclock` value. ⑤ After some time, ZkSrv2 detects the failure of the leader, transitions into phase-1, and broadcasts vote (4,2) with the increased `logicalclock` ($l = 2$), but the vote cannot override the initial vote (5,1)

within ZkSrv1. None of the servers can reach a quorum on their votes. ⑥ After w ms, ZkSrv1 decides to resend its vote after ZkSrv2 changes its mind, waiting another w ms before phase-1 can be terminated. Thus, phase-1 experiences timeouts twice which enlarges the latency. Things can be even worse, if ZkSrv2 can reach a timeout faster than ZkSrv1 at stage 5 and re-send (4,2) again. In that scenario, ZkSrv1's timeout will be reset since it receives a new message, and wait an additional w ms.

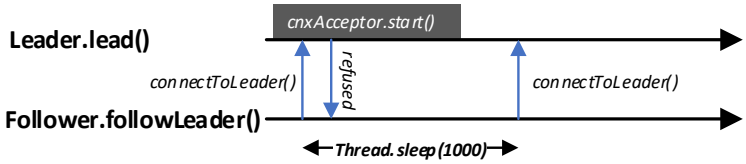
FCLE: Phase-1 Election Notification Re-sending. In order to address this unexpected multi-round waiting, we propose Fast Convergence Leader Election (FCLE) protocol: when a server receives a vote with a lower l =`logicalclock`, we re-send another notification (include its current vote) with an optional delay, as shown in Fig. 5.3c ④. Thus, we can avoid additional w ms in Fig. 5.3b.

5.2.3 Pre-Phase-2: Establish Quorum Channel

ZooKeeper: Two Communication Channels

In vanilla ZooKeeper, servers exchanges votes in phase-1 via the *election* channel (binding to TCP port 3888 by default), while other operations, including phase-2 recovery and phase-3 service, use another channel (TCP port 2888 by default) called *quorum channel*.

The election channel works like a complete graph: each node (ZooKeeper server) handles a TCP server and accepts connections from other peer. The election server is always active once a ZooKeepr has started. The quorum channel, on the contrast, forms a star-topology: the leader handles a TCP server; other followers connect to this TCP server as clients.



(a) Vanilla ZooKeeper: 1000ms re-connection retry

(b) Patch: Exploit the existing 3888 channel

Figure 5.4: Follower: Connect to the Lead via TCP Port: 2888

As a result, the quorum server cannot be established before phase-1, and it has to be established immediately by the new elected leader once its leadership has been confirmed.

Limitations and Solutions

At the end of phase-1, the leader calls the `Lead.lead()` method. Then, the leader calls `cnxAcceptor.start()` to bring up a new thread as a server, to accept incoming connection requests.

Meanwhile, at the end of phase-1, a follower immediately executes `Follower.followLeader()` and connects to the leader by calling the `connectToLeader()` method.

As shown in Fig. 5.4a, since the new leader has more work to do, it is very likely that a follower's first connection request is refused. According to the vanilla ZooKeeper implementation, the follower then must wait for a **fixed 1000ms** interval before the next retry, which significantly enlarges the overall recovery latency.

FQCE: Pre-Phase-2 Server Ready Notification. One might manually shorten the retry period for a more responsive system. However, since ZooKeeper has already established the leader election channel (on TCP port 3888), we can exploit that channel to achieve Fast Quorum Channel Establishment (FQCE), letting the leader to broadcast a *CNX_READY* packet to inform the follower once the new quorum channel is ready (Fig. 5.4b), and thus avoid unnecessary waiting time. Note that even the vanilla ZooKeeper assumes a "reliable" communication [57, 36], so packet loss is beyond the scope of this research. Nevertheless, we implement a time-out mechanism for robustness, handling the unlikely *CNX_READY* loss, similar to what ZooKeeper does in its mainstream version.

5.2.4 Phase-2: Recovery

ZooKeeper Recovery

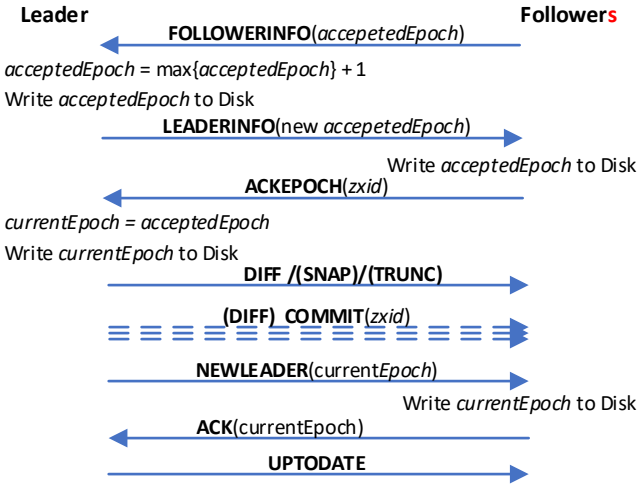


Figure 5.5: Protocol of Phase-2 RECOVERY

Fig. 5.5 shows the protocol in phase-2, RECOVERY. After electing the leader in phase-1, the quorum channel will be established. Each follower then connects to the leader and sends a *FOLLOWERINFO* packet to the leader, reporting its `acceptedEpoch`. When the leader receives a quorum of `FOLLOWERINFO` packets (including the one the leader sent to itself), it generates a new `acceptedEpoch` whose value is greater than any from the receiving set. The leader then persists the new `accpetedEpoch` to disk. The leader replies to each follower with the new `accpetedEpoch` encapsulated within a *LEADERINFO* packet. Upon receiving the packet, the follower updates the `accpetedEpoch` and persists it to disk. The follower acknowledges the leader with *ACKEPOCH* and informs the leader of the latest `zxid` of the follower's database. The leader collects a quorum of acknowledgements, then updates its `currentEpoch` and persists it to disk. According to the leader election in phase-1, the leader should has the most recent `zxid`. As a result, the leader can initiate the database synchronization by sending a *DIFF* packet, followed by several *COMMIT* packets, until finally each follower is synchronized. Then, the leader sends a *NEWLEADER* packet to update the `currentEpoch` value, to the followers. Followers persist the `currentEpoch` on disk, sending an *ACK* back to the leader. Upon receiving a quorum of *ACK*s, the leader finally sends an *UPTODATE* packet and goes into phase-3, SERVICE. Each follower also transitions into the SERVICE phase after receiving the *UPTODATE* packet.

Limitations and Optimization

As shown in both Fig. 5.5, the phase-2 procedure involves four disk I/O operations for persisting `acceptedEpoch` and `currentEpoch` in both the leader and followers. Our empirical results show that significant latency is introduced by that disk I/O (see "RTZK-L Phase-2"

in Fig. 5.8b), especially because the two variables are stored in distinct files. Each file write operation can take as long as 300ms.

The `acceptedEpoch` and `currentEpoch` variables play important roles in recovery: a server recovering from a crash must know the last epoch it accepted and was in, to deduce its status when it suffered a failure, and to take part successfully in leader election. The early implementation of ZAB, ZAB-Pre-1.0 [56], partially omitted these variables: It only recorded the epoch e in the latest persisted z, resulting in some "blocking level" bugs [67, 55]. The bugs were not completely resolved until ZAB-1.0 [112], which re-implements the handling of these two variables, in ZooKeeper version 3.4.

DEVP: Store Latest Epoch Values Over Quorum. To avoid the excessive latency introduced by disk I/O, we propose Distributed Epoch Value Persistence (DEVP) protocol, persisting the epoch values (i.e. `acceptedEpoch` and `currentEpoch`) distributively, among a quorum of ZooKeeper servers. Our approach is based on the observation that, in an edge computing environment with high-speed LAN (or Time-Sensitive Networks (TSN)), communication latency between local servers is usually smaller than that of disk I/O. To realize DEVP, we conduct holistic modifications in both phase-1 and phase-2 of vanilla ZAB protocol.

In DEVP, each ZooKeeper server maintains an *EpochTable*, as shown in Tab. 5.2, which contains global information about the `acceptedEpoch` and *currentEpoch* of all the servers in the current ensemble. We changes the means to access the two variables:

Phase-1 Modification. Each ZooKeeper server needs to get the `currentEpoch`, to initialize l =`logicalclock` before staring FLE. In the ZAB protocol, each ZooKeeper server just accesses its disk to read the persisted value from a specified file. In DEVP protocol,

Field	Type	Note
version	long	the version of this table
numServers	int	the size of the ensemble
acceptedEpochList	long[]	acceptedEpoch values of all servers
currentEpochList	long[]	currentEpoch values of all servers

Table 5.2: EpochTable: Structure

before entering the FLE algorithm, each server runs an OptFloodMax algorithm to reach a consensus on the EpochTable. The servers choose the one with the latest (greatest) `version` as the consensus result. Then the servers can retrieve the epoch value from the already synchronized and up to date EpochTable.

Phase-2 Modification. In phase-2, ZooKeeper servers will change and persist their epoch values. In ZAB, each ZooKeeper simply writes the new value to disk and sends an acknowledgement. When using , we allow only the leader to modify the contents of its EpochTable, while the followers can only propose their variable changes to the leader. This can be done via a two-step commit protocol: 1. The leader accumulates the changes from all the followers (in a quorum). Then, the leader generates a new EpochTable with corresponding updates, including a new version number, broadcasting the EpochTable to the followers. 2. Upon receiving a new EpochTable, the follower commits it, sending ACK to the leader. 3. Upon receiving a quorum of ACKs, the leader continues the ZAB protocol.

Persist Epoch Values: The Implementation. We use the same channel as leader election (TCP port 3888) for delivering our EpochTable-related messages, and implement the function handling them within `FastLeaderElection` class, to address the following design considerations: 1. The latest EpochTable needs to be detected before the FLE algorithm, so the quorum channel (TCP port 2888), which is established in pre-phase-2, cannot be used for this. 2. The latest EpochTable detection is based on the same algorithm as the FLE

(OptFloodMax). 3. The application-level packet framing class, `ToSend`, has an enumerated type field `mType`, which facilitates message extension and multiplexing.

We add a `lookForLatestEpochTable()` method, for detecting and synchronizing the EpochTable before running the FLE. It is similar in structure to the `lookForLeader()` method, since they both implement the OptFloodMax algorithm in a similar way. To support broadcasting the EpochTable via the election TCP channel, we modified the message sending method (`WorkSender.process()`), allowing it to frame two different message types: `notification` and `epochtable`, for FLE and EpochTable detection, respectively (based on `mType`, which is in the head of the frame). We also modified the application layer message receiver and forwarding thread (`WorkerReceiver.run()`): by parsing the message header, the receiver can forward the messages to the corresponding processing queue, for polling by `lookForLatestEpochTable()` or `lookForLeader()`.

5.3 Recovery Time Analysis

In this section, we establish recovery time models for RT-ZooKeeper, which leverages FCLE, FQCE, and DEVP protocols. As shown in the phase transition diagram (Fig. 5.2), the overall recovery latency T comprises two parts, the phase-1 election latency T_e and the phase-2 recovery latency T_r:

$$T = T_e + T_r. \tag{5.1}$$

We start with the model of phase-1 election latency, revealing how the jitter and communication delay can affect the model and then discussing the timing behaviour of OptFloodMax

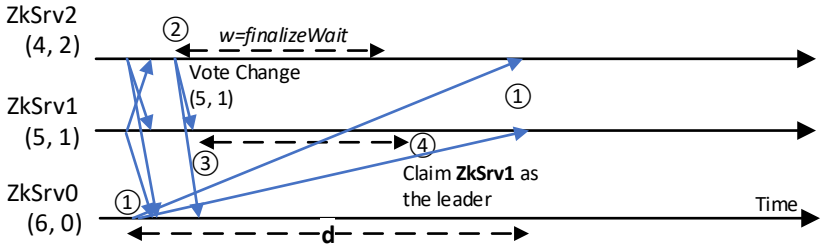
implemented with FCLE. We then model the phase-2 and the communication cost for maintaining EpochTable via DEVP protocol.

5.3.1 Timing Model for Phase-1

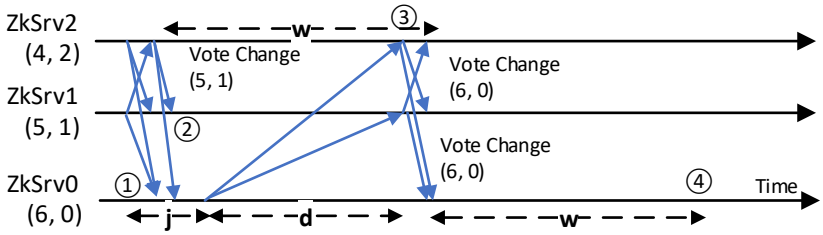
How long should finalizeWait be. In vanilla ZooKeeper, w has a hard-coded fixed value: 200ms. We argue that this value is unnecessarily large for a local area network (LAN) environment, where the transmission latency d can be significantly less than 200ms. However, w cannot be too short either, or the ensemble may elect the wrong leader, which would significantly increase phase-2 recovery latency. For example, in Fig. 5.6a, server ZkSrv0 has the most recent transaction and hence should be the new leader. However, ① ZkSrv0's messages suffer from a long transmission delay . ②③ ZkSrv1 and ZkSrv2 can reach a quorum (2 of 3) with the vote (5,1) being the maximum among them . ④ After w time units without receiving new votes, ZkSrv1 and ZkSrv2 will claim ZkSrv1 as the (false) leader.

It is not sufficient to just have $w > d$, since the ZooKeeper servers may not start leader election at exactly the same time. Instead, we can use a value j to represent the jitter in their collective start times. As shown in Fig. 5.6b: ① ZkSrv0 suffers from both jitter j and long transmission delay d. ② ZkSrv0 ignores the votes from other two servers. ③ We need a sufficiently large w to ensure ZkSrv1 and ZkSrv2 receives ZkSrv0's vote before they start deducing the leader. Hence, we require:

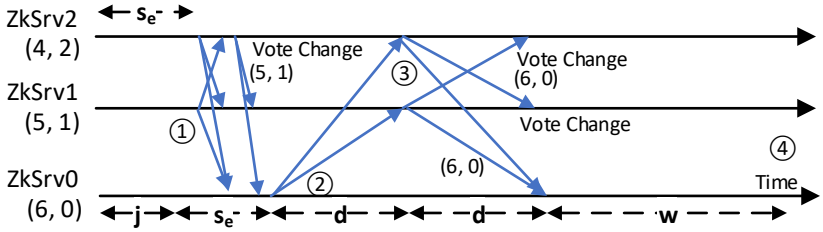
$$w > j + d, \tag{5.2}$$



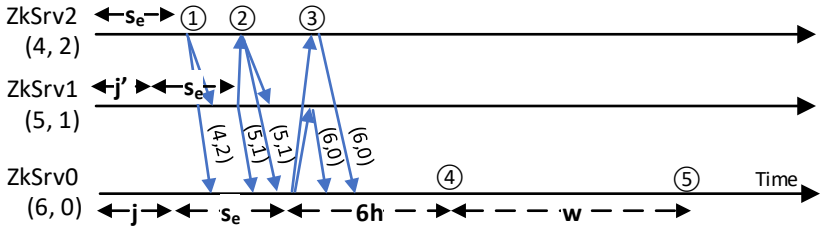
(a) A short `finalizeWait`, $w < d$, results in a false leader:



(b) A proper `finalizeWait`, $w > d + j$, which considers the jitter



(c) Cloud Environment: OptFloodMax $T_{o'}$ dominated by d



(d) LAN Environment: OptFloodMax T_o dominated by h

Figure 5.6: Timing Model of Leader Election.

to avoid the false leader scenario. ④ ZkSrv1 and ZkSrv2 change they vote to (6,0) and rebroadcast them and everyone reaches a quorum; the procedure converges.

Estimate OptFloodMax Latency. In a public cloud scenario (Fig. 5.6c), where the transmission latency can significantly larger than the message processing latency (e.g., where d

can be more than 100ms and h can be several ms), the latency, T'_o, for running OptFloodMax algorithm once can be modeled as:

$$T_{o'} = 2d + w. \tag{5.3}$$

① It takes a constant initialization time s_e before running the actual OptFloodMax algorithm. ② In addition, ZkSrv0 suffers from both the jitter and long transmission delay d. ③ Another transmission delay d is introduced when ZkSrv1 and ZkSrv2 wants to rebroadcast (6, 0). ④ The servers need to wait w for finally finishing the election.

However, in a LAN setting where d is less then 10ms, and most of the time is less than 1ms, the model in Eq.(5.3) does not work since the single message processing time h can affect the overall leader election time significantly. Thus, we use a different model (T_o) for this scenario:

$$T_o = \frac{N(N+1)}{2}h + w. \tag{5.4}$$

The first term indicates the potential message processing time of the leader. In an ensemble of n servers, the server with n^{th} largest vote can change its mind $n-1$ times, thus broadcasting at most n messages to the network. As a result, the leader will receive $1+2+...+N$ messages in leader election process. Fig. 5.6d shows an example: ① ZkSrv1 broadcast its initial vote (4,2) first, while was ignored by the other two. ② After a jitter j', ZkSrv2 broadcasts its vote (5,1), resulting in a vote change in ZkSrv1. ③ ZkSrv0 broadcasts its vote and causes other two change their mind. ④ ZkSrv0 receiving $3 \times (3+1)/2 = 6$ messages from other peers, 5 from other peers plus 1 when ZkSrv0 broadcasts (6,0) to itself. ⑤ The procedure converges.

RT-ZooKeeper needs to run OptFloodMax algorithms twice in phase-1, one for EpochTable Detection, one for leader election. But we only need to count initialization (including jitter) once:

$$T_e = s_e + j + 2T_o. \tag{5.5}$$

5.3.2 Timing Model for Phase-2

As shown in Fig. 5.5, the ensemble first reaches a consensus on a new epoch, which is greater than any previous epoch and must persist on disk. Though the recovery phase involves several rounds of message exchanges, the most time consuming part is to persist the `acceptedEpoch` and `currentEpoch` values to disk: The leader and each of the followers have to write to the disk two times, resulting in 4x disk I/O latency.

In phase-3, ZooKeeper keeps the transaction synchronous among all peers. As a result, when a leader failure occurs, the unsynchronized transactions can be bounded. Thus, the timing model for phase-2 of vanilla ZooKeeper can be relatively simple:

$$T_{r'} = s_r + 4b. \tag{5.6}$$

Note that b stands for the I/O latency when writing either `acceptedEpoch` or `currentEpoch` to disk, while initialization and other fixed message communication and processing latency can be combined into a single constant s_r.

DEVP protocol leverages a two-phase-commit to update the *EpochTable*. Note that we remove the latency term $4b$ from Eq.(5.6) by avoiding disk I/O. The relationship between

the size of the ensemble and additional epoch messages to be processed is linear. Thus, the phase-2 timing model for RT-ZooKeeper is:

$$T_r = s_r + Nh_r. \tag{5.7}$$

With Eq.(5.8), (5.5), (5.4), and (5.7), we can have:

$$T = s_e + j + 2(\frac{N(N+1)}{2}h + w) + s_r + Nh_r. \tag{5.8}$$

5.4 Empirical Evaluation

In this section, we use a micro-benchmark on a real testbed to measure and evaluate the recovery latency for three ZooKeeper variants:

- **Vanilla ZooKeeper.** The original ZooKeeper 3.5.8 without any patch (whose `finalizeWait`, $w = 200$ms).
- **RT-ZooKeeper Lite (RTZK-L).** The ZooKeeper with partial features of RT-ZooKeeper: only FCLE and FQCE features are enabled.
- **RT-ZooKeeper (RTZK).** The full-fledged RT-ZooKeeper with all three protocol, FCLE, FQCE, and DEVP enabled.

The RTZK-L is configured for evaluation purpose: for differentiating the performance gain from different features. On one hand, we combine the FCLE and FQCE to RTZK-L since those two protocols and modifications are well-isolated and independent: their performance

gains can be easily differentiated by comparing to the phase-1 and phase-2 latency from vanilla ZooKeeper, respectively. On the other hand, using RTZK-L as a baseline, we can evaluate the pros and cons from DEVP, which involves modifications in both phases. We also evaluate whether the empirical maximum latency results for RTZK are bounded by the estimations from Eq.(5.8).

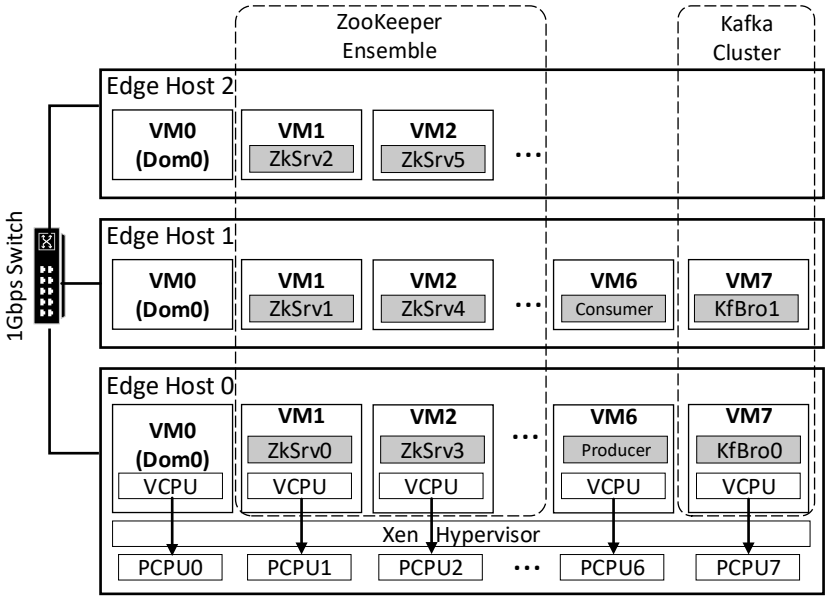
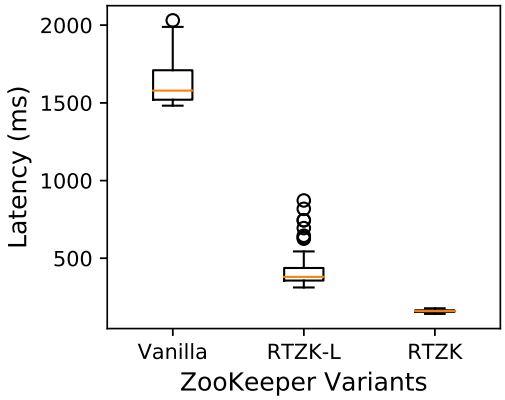


Figure 5.7: Testbed for ZooKeeper Recovery Evaluation

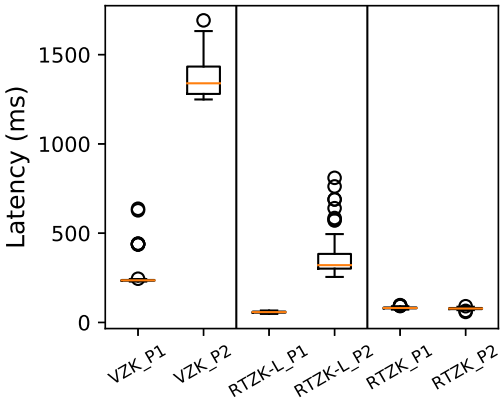
We conducted experiments on a testbed with three physical machines working as edge hosts. Each host has one Intel E5-2680v4 8-core CPU, 64 GB memory, and 1TB 7200 RPM-HDD. We disabled hyper-threading and power saving features and fixed the CPU frequency at 2.1 GHz to improve predictability, as in [64, 141, 138]. As virtualization is often used in edge computing environments to facilitate deployment and other practical considerations, each physical host runs a Xen 4.12.0 hypervisor to consolidate multiple virtual machines (VMs). As shown in Fig. 5.7, each ZooKeeper server (or Kafka broker) is encapsulated with a Linux VM (kernel version 5.4.0) and consolidated on one of the physical hosts. Each VM has one

VCPU which is pinned on a dedicated PCPU. The administrative VM of Xen, Dom 0, has one VCPU which is pinned onto PCPU0. The physical hosts are connected to a 1Gbps switch, and all the VMs are in the same LAN. All ZooKeeper variants we evaluated are based on version 3.5.8.

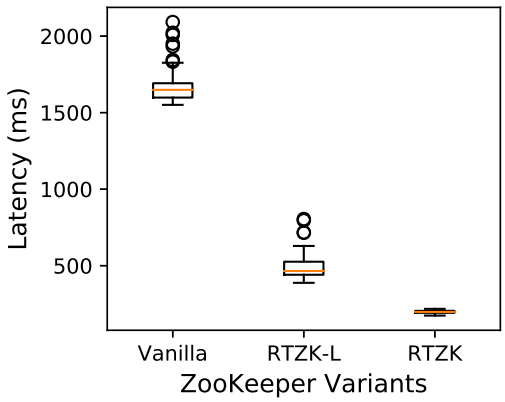
As shown in Fig. 5.7, each ZooKeeper ensemble in our evaluations consists of up to seven ZooKeeper servers, whose ids range from 0 to 6 (*ZkSrv0 to ZkSrv6*), which we allocate to physical hosts in a round-robin fashion. We bring up a ZooKeeper ensemble, kill the leader to trigger the ZooKeeper recovery procedure, and then measure the recovery latency: for each of the three ZooKeeper variants, we repeat that measurement procedure 100 times.



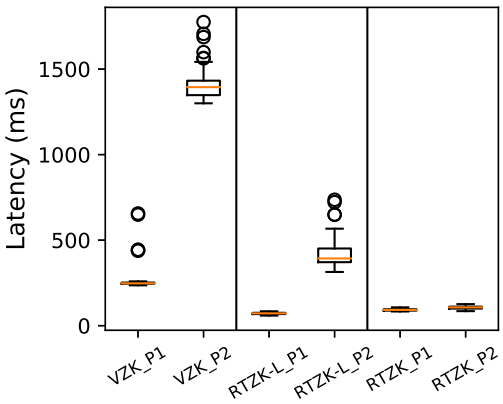
(a) Overall, 3 servers



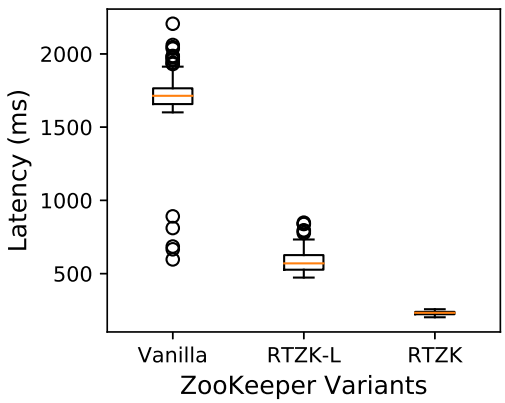
(b) Breakdown, 3 servers



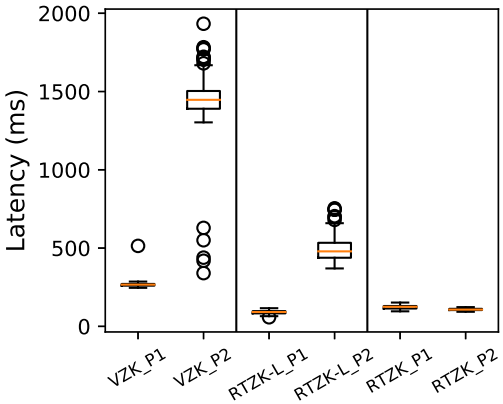
(c) Overall, 5 servers



(d) Breakdown, 5 servers



(e) Overall, 7 servers



(f) Breakdown, 7 servers

Figure 5.8: ZooKeeper Recovery Latency Distribution

We measured the maximum transmission latency at $d = 3$ms, and the maximum jitter at $j = 13$ms for failure detection. Thus we choose the `finalizeWait`, $w = 20$ms following Eq.(5.2) and using the other parameters shown in Tab. 5.3.

Parameter	s_e	j	w	h	s_r	h_r
Value(ms)	35	13	20	3	55	25

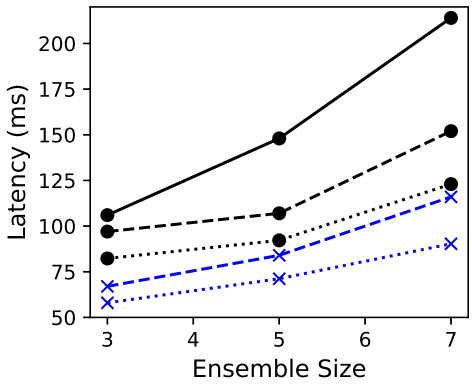
Table 5.3: Parameters for Latency Estimation

We ran the same experiments using ensembles consisting of 3, 5, or 7 ZooKeeper servers: although it is possible to have more servers, running a ZooKeeper ensemble with more than seven servers is rare in production settings, even in a public cloud environment. For example, CloudKarafka allows up to seven ZooKeeper servers [135] and Solr recommends no more than five ZooKeeper servers [90]. Moreover, a seven-server-ensemble can easily serve more than $1,000$ nodes which is beyond the scale of a normal edge cloud: we argue that especially since the number of compute nodes in an edge cloud is much less than in a public production environment, it is reasonable to test against the commonplace settings.

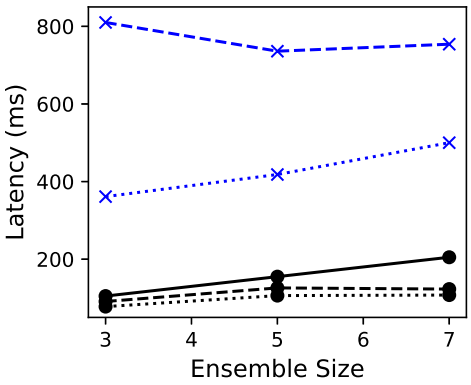
Figures 5.8a, 5.8c and 5.8e show the overall latency distributions for the three ZooKeeper variants (vanilla ZooKeeper, RTZK-L, and RTZK) with a 3, 5, or 7 server ensemble. We observe that RTZK-L outperforms vanilla ZooKeeper and that among the three variants RTZK has the best performance, significantly reducing maximum latency from 2031ms to less than 178ms, i.e., RTZK shortens latency by 92.1%.

We further analyse the results by breaking down the overall latency for each ZooKeeper variant into two parts: phase-1 leader election latency and phase-2 database recovery latency, as shown in Figures 5.8b, 5.8d and 5.8f. We make three observations based on those results:
1. The phase-1 leader election latency samples for vanilla ZooKeeper (*VZK_P1*) are clustered

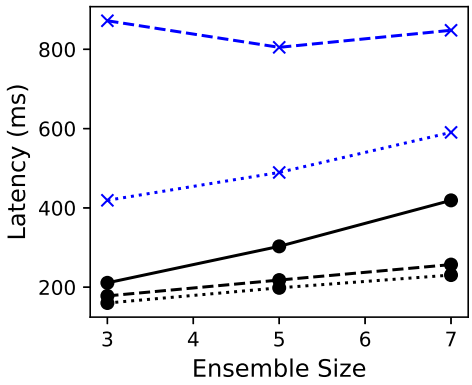
into three levels: 250ms, 450ms, and 650ms. This phenomenon is due to the unnecessary w ="finalizeWait" rounds, which we explained in Sect. 5.2.2. Our corresponding FCLE protocol fixes this problem and reduces w value, as we can see in *RTZK-L_P1*. 2. The phase-2 database recovery latency dominates the overall latency for vanilla ZooKeeper. RTZK-L improves phase-2 recovery latency (RTZK-L_P2) by avoiding the unnecessary one-second reconnection wait via the FQCE protocol, as we discussed in Sect. 5.2.3. However, even that optimized latency is still relatively long and has significant outliers due to disk operations. 3. RTZK avoids those disk operations by persisting those values among the quorum via the DEVP protocol. RTZK thus significantly shortens the phase-2 recovery latency and achieves the best overall results among the three variants.



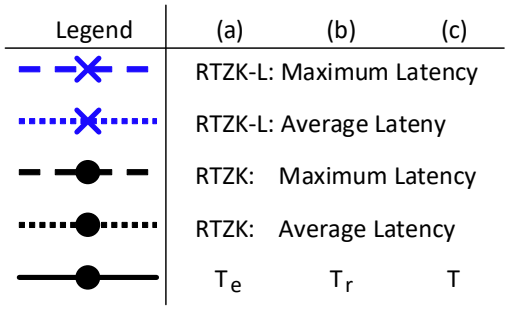
(a) Phase-1: Election



(b) Phase-2: Recovery



(c) Overall



(d) Legends for (a),(b),(c)

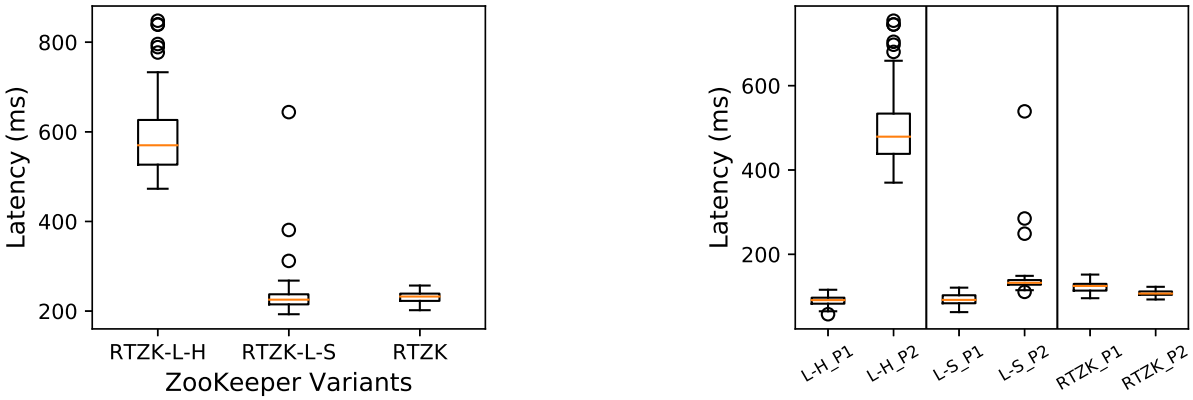
Figure 5.9: Latency on different Variant/Ensemble

Given the parameters in Tab. 5.3, we calculate an estimated maximum overall (T), phase-1 (T_e), and phase-2 (T_r) latency, for RTZK based on Eq.(5.8), (5.5), and (5.7), respectively. Note that since a server has crashed, the number of servers taking part in recovery is one less than the ensemble size. We show the estimated maximum latency (T_e, T_r, and T) in Figures 5.9a to 5.9c, which provide upper bounds for gauging RTZK's empirical results.

We note that RTZK achieves the best performance by avoiding disk operation latency (Fig. 5.9c) but at a cost of more network operations. Hence the phase-1 leader election

latency of RTZK is larger than the one for RTZK-L (Fig. 5.9a). Specifically, RTZK-L only needs to run the *OptFloodMax* algorithm once for electing a leader, but RTZK needs to run it twice, one for synchronizing the `EpochTable`, and another for leader election. However, it is not a major concern in light of real-world ZooKeeper ensemble sizes, which are rarely greater than seven even in production [135, 90].

RTZK-Lite on SSD. To further address the capability and advantage of the DEVP protocol of RT-ZooKeeper, we replaced the HDD disk for RTZK-Lite with an SSD disk.



(a) Overall, 7 servers

(b) Breakdown, 7 servers

Figure 5.10: RTZK-Lite on SSD

As shown in Fig.5.10, we demonstrated the latency distribution of three different configurations on a seven-server-ensemble: RTZK-Lite on HDD (RTZK-L-H), RTZK-Lite on SSD (RTZK-L-S), and RTZK (which does not need disk I/O for accessing epoch values). The RTZK-L-S shorten phase-2 latency in most cases (L-S_P2 in Fig.5.10b) when comparing to its HDD variant. However, it still suffers from excessive latency occasionally. Hence the

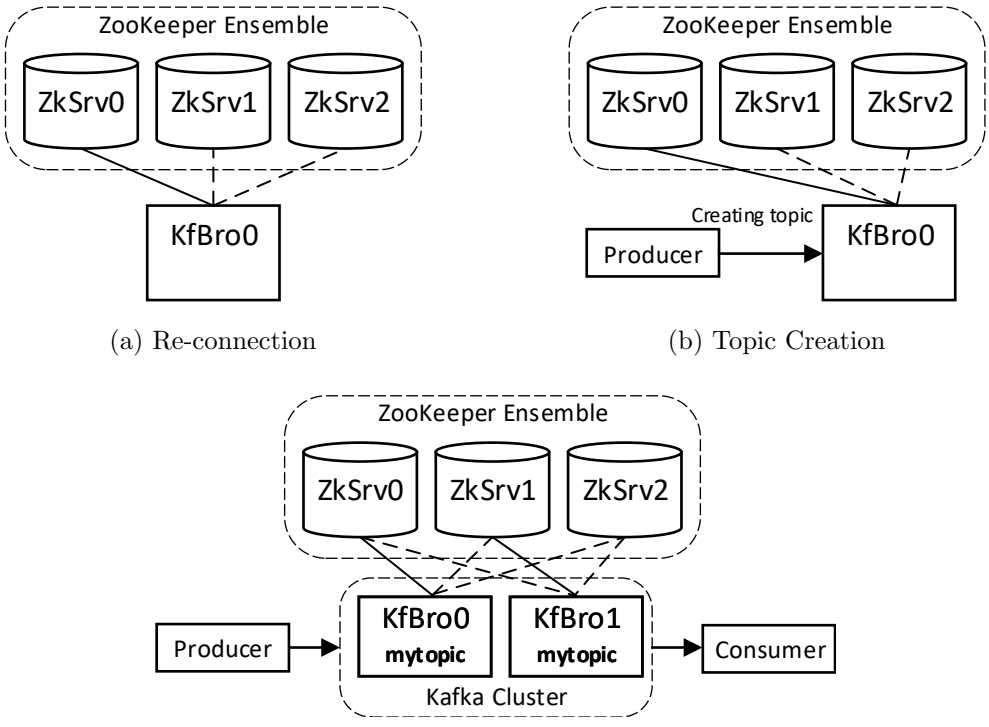
maximum latency performance of RTZK-Lite, even equipped with an SSD, is still uninspiring. RTZK, on the contrary, depriving of the disk I/O accessing, yields much consistent result without latency samples (outliers) that significantly deviates the majority.

5.5 Case Study with Kafka

In this section, we conduct case studies involving a distributed real-time messaging service, Kafka, on the same testbed shown in Fig. 5.7, with Kafka version 2.6.0. Kafka brokers work as ZooKeeper clients, using ZooKeeper for topic creation, leader election for brokers and topic partition pairs, and monitoring topology changes for the Kafka cluster.

We built a Kafka cluster with two Kafka Brokers (*KfBro0 and KfBro1*) distributed on two physical hosts. We also use other VMs for a Kafka message producer and consumer.

We conduct three case studies of increasing complexity, to evaluate the benefits of low recovery latency using RT-ZooKeeper: 1. a Kafka re-connection latency test, 2. a Kafka topic creation latency test, and 3. a Kafka message end-to-end latency test. In each case study, we measure the corresponding operation's latency following ZooKeeper leader failure.



(a) Re-connection

(b) Topic Creation

(c) Message End-to-End Latency

Figure 5.11: System architecture for each Kafka case study

5.5.1 Kafka Re-connection

As shown in Fig. 5.11a, we use a ZooKeeper ensemble consisting of three servers, and a Kafka broker, which knows all the entry points of the ZooKeeper ensemble.

We conduct an experiment to measure the Kafka re-connection latency following ZooKeeper leader failure. In each run, we first kill the ZooKeeper leader. The ZooKeeper service then becomes temporarily unavailable and the Kafka broker loses its connection to the ensemble. After some time, ZooKeeper service recovers. Meanwhile, the Kafka broker, as a ZooKeeper client, will keep trying to reconnect. The re-connection request cannot succeed

until ZooKeeper ensemble completes its recovery. We then measure the elapsed time from Kafka's lost connection, to its re-connection.

ZooKeeper Client Re-connection Strategy. First, we highlight that the client re-connection strategy is a significant factor for re-connection latency, due to two different innate strategies in the vanilla ZooKeeper client library. First, after failure of the client's connection to a ZooKeeper server, the client will wait for a randomized interval, ranging from 0 to 1000ms, before retrying. Second, vanilla ZooKeeper has a "delayed server redirection" mechanism: if a client tries to reconnect to a recovered ZooKeeper ensemble, but picks up the failed server to connect with, the client will incur an additional one second delay before it can try another server.

Aggressive Re-connection Strategy. Unfortunately, these re-connection delay introduced by the vanilla ZooKeeper client library can even eclipse the benefits of RTZK. Therefore, we patched the ZooKeeper client library to use significantly more responsive settings: 1. the client will retry every 50ms; 2. the client will immediately try another entry point if it accidentally picks up a failed server.

We ran our experiments 100 times for each of six different ZooKeeper configurations: the vanilla ZooKeeper, RTZK-L, and RTZK, each with either the original ZooKeeper client library or with the one patched to use or aggressive re-connection strategy.

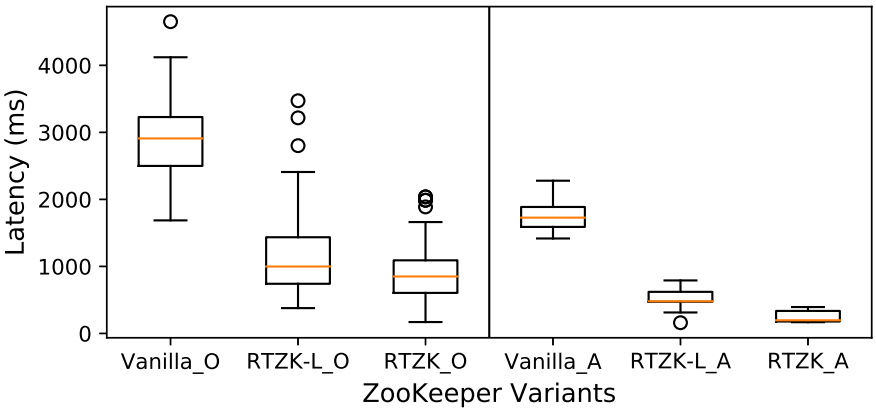


Figure 5.12: Kafka Re-connection Overall Latency

Fig. 5.12 shows the Kafka re-connection latency distributions for the different ZooKeeper configurations. We make four observations: 1. RTZK always outperforms the corresponding vanilla ZooKeeper and RTZK-L configurations in terms of both median and maximum latency, regardless of the re-connection strategy. 2. When using the original conservative re-connection strategy (labeled *_O), the Kafka broker is likely to take more time to reestablish the connection, hindering the potential benefit from using optimized ZooKeeper or RT-ZooKeeper. 3. The aggressive re-connection strategy (labeled as *_A) can efficiently alleviate the latency penalty introduced by client, hence making the overall results close to what we report in Fig. 5.8. Moreover, by combining the aggressive connection strategy with RT-ZooKeeper, we achieve the best performance and significantly reduce latency. 4. Similarly low latency is not achieved by only applying an aggressive re-connection strategy with vanilla ZooKeeper, nor by only using RT-ZooKeeper without adopting the aggressive re-connection strategy.

5.5.2 Kafka Topic Creation

In this case study, we add a Kafka producer to create a new topic (Fig. 5.11b). The topic's metadata is stored in ZooKeeper. Thus, ZooKeeper is on the critical path for topic creation.

We measure the topic creation latency under the following conditions: We first test the latency distribution in normal conditions, where no failure occurs; then, we intentionally kill the ZooKeeper leader during topic creation, to evaluate the impact of ZooKeeper recovery. We repeat the experiments for the three ZooKeeper variants: vanilla Zookeeper, RTZK-L, and RTZK, using the aggressive re-connection strategy.

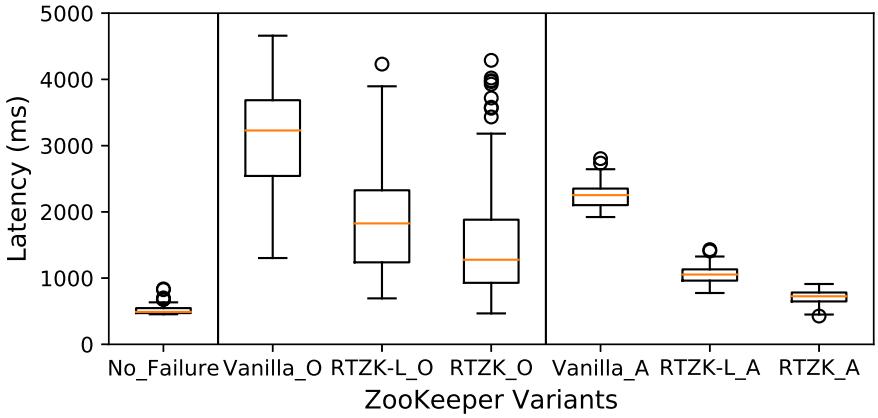


Figure 5.13: Kafka Topic Creation Overall Latency

Based on the results in Fig. 5.13, we make two observations: 1. ZooKeeper failure does have negative impact on the Kafka topic creation latency: None of the configurations (with a ZooKeeper failure) achieves the optimal topic creation latency distribution seen without a ZooKeeper failure. 2. By combining the aggressive connection strategy with RTZK, we can alleviate such a negative impact to the greatest extent, significantly reducing latency when compared to other configurations.

5.5.3 Kafka Message End-to-End Latency

In this case study, we explore how ZooKeeper leader and Kafka leader failure affect message end-to-end latency.

As shown in Fig. 5.11c, we create a three-server ZooKeeper ensemble for a Kafka cluster consisting of two Kafka brokers. We create a topic "*mytopic*" with a replication factor of 2, i.e., Kafka is responsible to duplicate the message log on both Kafka brokers. After initialization, Kafka broker 0 and ZooKeeper server 0 are the leaders for the Kafka cluster and the ZooKeeper ensemble, respectively. A producer publishes a message to "mytopic" every 100ms (periodically), and a consumer subscribes to the same topic. We measure the end-to-end message latency under different circumstances.

Only The ZooKeeper Leader Fails. Kafka leverages the ZooKeeper service for topic creation, electing a new leader whenever a previous leader fails. However, for regular message routing, Kafka does not need to consult ZooKeeper for delivering messages. Hence, a failure of the ZooKeeper service will **not** affect the message end-to-end latency.

Only The Kafka Leader Fails. If the Kafka topic leader fails, the Kafka broker cannot deliver messages until the Kafka replica recognizes the failure of the old leader and establishes a new one. Kafka failure detection is handled by ZooKeeper: ZooKeeper establishes an entity called a "*Session*" to represents the existence of a living Kafka broker. The Kafka broker sends heartbeats to the ZooKeeper, indicating its existence. If the session times out, ZooKeeper treats the broker as failed and then notifies the other corresponding brokers.

The `tickTime` setting affects latency. Clearly, the session timeout value, `zookeeper.session.timeout.` in the Kafka broker configuration can significantly affect the failure detection latency: the

lower that value, the faster ZooKeeper can detect the Kafka broker failure. The minimal session timeout value is gauged by the ZooKeeper *tickTime*, and can at least be two times of the `tickTime`. In the following experiment, we always set `zookeeper.session.timeout.ms` to be two times the value of the ZooKeeper `tickTime`.

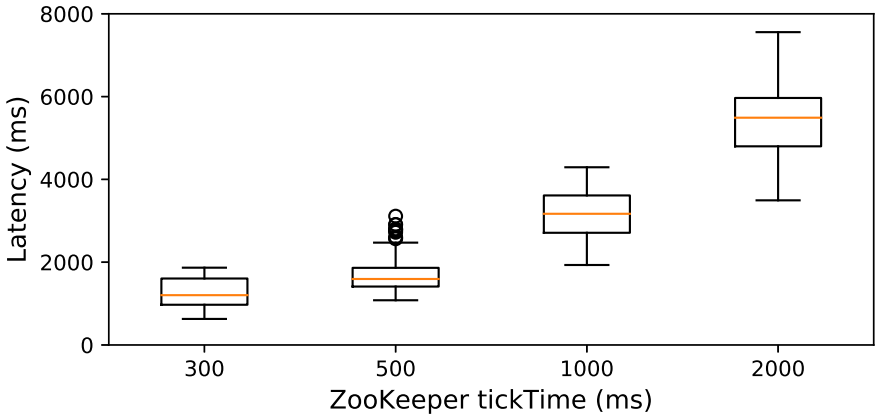


Figure 5.14: E2E Latency: Only Kafka Leader Fails

We first ran the experiment with **only** a Kafka leader failure, but **without** a ZooKeeper leader failure. We measured the worst message end-to-end latency in each run. We performed 100 runs for each of the following ZooKeeper `tickTime` settings: {300, 500, 1000, or 2000} milliseconds. As shown in Fig. 5.14, even without ZooKeeper undergoing a recovery, the `tickTime` affects the latency significantly: The larger the `tickTime`, the worse the end-to-end latency.

Both the Kafka Leader and the ZooKeeper Leader Fail. In this case, ZooKeeper is on the critical path of the service: the other Kafka broker relies on ZooKeeper for the failure notification:

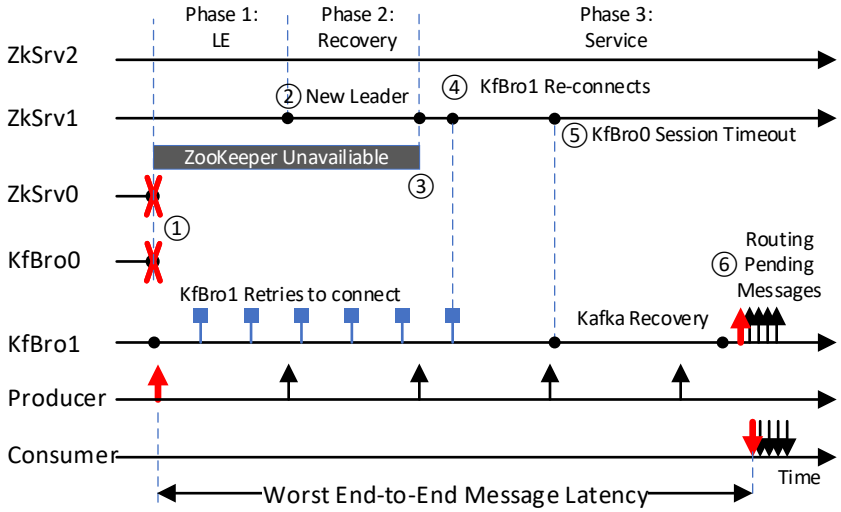


Figure 5.15: Timeline for Measured Kafka End-to-End Message Latency with Kafka and ZooKeeper Leader Failures

Fig. 5.15 shows the timeline for such a case: ① The Kafka leader (KfBro0) and the ZooKeeper leader (ZkSrv0) fail simultaneously. The other two ZooKeeper servers lose their leader, closing all connections of its client (KfBro1) and starting leader election. The other Kafka broker (KfBro1) loses its connection to ZooKeeper, and immediately tries to re-connect to the ZooKeeper ensemble (we assume the client adopts the "aggressive re-connection" strategy). ② The remaining ZooKeeper establishes an new ensemble consisting of two servers, elects a new leader, and starts phase-2 recovery. ③ The ZooKeeper ensemble recovery completes and resets the session timeout deadline. ④ KfBro1 finally reconnects to the recovered ZooKeeper service, avoiding a session timeout. ⑤ The KfBro0 session in the ZooKeeper ensemble must expire later, since KfBro0 had been dead. As a result, the ZooKeeper ensemble can send the notification for KfBro1. ⑥ KfBro1 finishes its recovery, including Kafka cluster leader election and data recovery, and can then process the pending messages.

The producer, regardless of the temporarily unavailable Kafka service, sends messages periodically. These messages cannot be routed until the Kafka cluster recovers. As a result, the messages sent just after the failure occurring suffer the longest latency. We record that for each run.

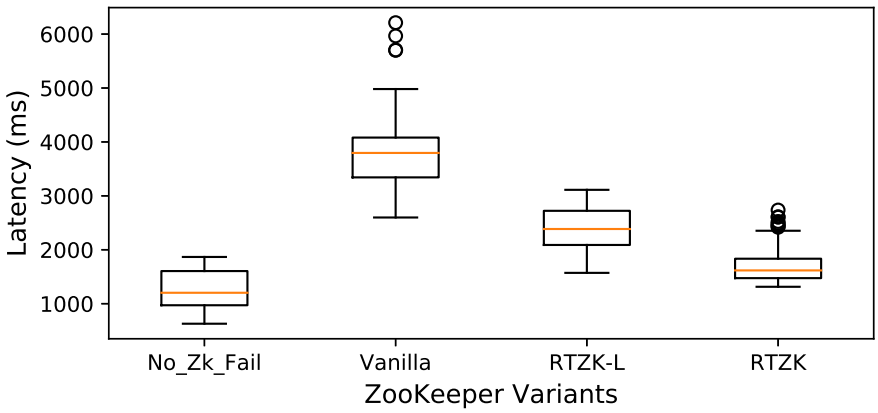


Figure 5.16: E2E Latency: Both Kafka and ZK Leader Fail

We set the ZooKeeper `tickTime` to 300ms, and the Kafka session timeout to 600ms. We made 100 runs for each ZooKeeper variant and show the distribution of the worst end-to-end latency in Fig. 5.16. We make following observations: 1. The only-Kafka-leader-failure case represents the optimal lower bound for this experiment. The experiment results involving both Kafka and ZooKeeper failures can never outperform the one without ZooKeeper failure. 2. RT-ZooKeeper still achieves the best performance, by combining rapid recovery with the "aggressive re-connection" strategy.

Based on the micro benchmark and the three different real-world case studies presented in this section, we can draw the following conclusions: 1. The FCLE and FQCE protocols effectively shorten ZooKeeper recovery latency. 2. The DEVP protocol further improves

recovery latency by avoiding disk I/O. 3. The combination of RT-ZooKeeper with an aggressive client reconnection strategy is effective in constraining latency for services such as Kafka, which depend on ZooKeeper.

5.6 Related Work

Many research communities have addressed reliability and fault-tolerance of distributed systems in different ways. Protocols like Paxos [70] are widely used for building replicated state machines (RSMs) [71]: e.g., Gaios [18], S-Paxos [17], and the Chubby [23]. ZooKeeper [51] and its ZAB [57] protocol address consistency issues for Paxos-based protocols: the primary fault can result in conditions that violate casual ordering. The ZAB resolves this issue by fulfilling the *Primary Order* property.

While ZooKeeper and this work focused on crash faults, there have also been research on real-time fault-tolerant systems involving more complicated transient faults and Byzantine faults have been widely studied and developed[39, 46, 123, 116]. There have been previous work on achieving real-time performance in the presence of crash failures. DeCoRAM [10] provides a task allocation algorithm for replicated systems, to meet real-time and fault-tolerance requirements with less resources. FRAME [136], a fault-tolerant real-time messaging service for edge cloud, exploits relationships between service parameters and loss-tolerance/latency requirements. None of those projects on crash failures dealt with time-bounded leader election or failure recovery.

The rarity of ZooKeeper recovery latency studies implies a potential challenge for adopting it to edge environment, regardless of its popularity. Most existing research and evaluation

studies on ZooKeeper have focused on throughput and latency performance in phase-3 [61, 52, 36], leaving the recovery procedure largely unaddressed. In this paper, we proposed RT-Zookeeper, which focuses on reducing the recovery latency of ZooKeeper for distributed real-time applications on edge computing systems, with several contributions addressing phase-1 and phase-2 that can greatly reduce recovery latency.

5.7 Conclusions

We have studied the recovery latency of fault-tolerant coordination services in edge computing environments, with a specific focus on ZooKeeper, the prevailing reliable coordination service. We focus on improving recovery latency following leader failure and establish a recovery time analysis for ZooKeeper, including timing models for both the leader election phase and the recovery phase. We identified bottlenecks that impede recovery, and propose news protocols to alleviate those limitations. We implement those advances in RT-ZooKeeper, as extensions to ZooKeeper 3.5.8. Results of our empirical evaluations, including case studies using the Kafka, demonstrate that our analyses and the improvements that arose from them can significantly reduce the recovery latency. RT-ZooKeeper provides appropriate support for latency-sensitive distributed applications running in edge computing environments. Targeting ZooKeeper as a representative system, this work highlights the importance of revisiting the fundamental designs of cloud services from a real-time system perspective to meet the need of real-time edge computing systems.

Chapter 6

Conclusion

This dissertation seeks answers and system solutions to the field of real-time edge computing, addressing dynamic resource provisioning, predicting latency distribution and tail-latency SLO for periodic and stochastic tasks, and achieving fast recovery in coordination services.

With regards to dynamic resource provisioning, we have built M2-Xen, which enables dynamic resource interface change, allowing VMs to switch modes with different CPU resource requirements at run-time while maintaining desired real-time performance. The VAS and M/D(DS)/1 system have coped with the latency distribution of periodic and stochastic tasks on virtualized hosts. RT-ZooKeeper extends Apache ZooKeeper, the prevailing open-source coordination service, significantly reducing the latency in recovering from leader failures in replicated coordination services.

6.1 Summary of Results

We briefly recap the key points of Chapter 2-5 here.

6.1.1 M2-Xen

Real-time virtualization is an emerging technology for embedded systems integration and latency-sensitive cloud applications. Earlier real-time virtualization platforms require offline configuration of the scheduling parameters of virtual machines (VMs) based on their worst-case workloads, but this static approach results in pessimistic resource allocation when the workloads in the VMs change dynamically. In Chapter 2 we present *Multi-Mode-Xen (M2-Xen)*, a real-time virtualization platform for dynamic real-time systems where VMs can operate in modes with different CPU resource requirements at run-time. M2-Xen has three salient capabilities: (1) dynamic allocation of CPU resources among VMs in response to their mode changes, (2) overload avoidance at both the VM and host levels during mode transitions, and (3) fast mode transitions between different modes. M2-Xen has been implemented within Xen 4.8 using the real-time deferrable server (RTDS) scheduler. Experimental results show that M2-Xen maintains real-time performance in different modes, avoids overload during mode changes, and performs fast mode transitions.

6.1.2 VAS

As time-sensitive applications are deployed spanning multiple edge clouds, delivering consistent and scalable latency performance across different virtualized hosts becomes increasingly challenging. In contrast to traditional real-time systems requiring deadline guarantees for all jobs, the latency service-level objectives of cloud applications are usually defined in terms of *tail latency*, i.e., the latency of a certain percentage of the jobs should be below a given

threshold. This means that neither dedicating entire physical CPU cores, nor combining virtualization with deadline-based techniques such as compositional real-time scheduling, can meet the needs of these applications in a resource-efficient manner.

To address this limitation, and to simplify the management of edge clouds for latency-sensitive applications, we introduce *virtualization-agnostic latency (VAL)* as an essential property to maintain consistent tail latency assurances across different virtualized hosts. VAL requires that an application experience similar latency distributions on a shared host as on a dedicated one. Towards achieving VAL in edge clouds, in Chapter 3, we presents a *virtualization-agnostic scheduling (VAS)* framework for time-sensitive applications sharing CPUs with other applications. We show both theoretically and experimentally that VAS can effectively deliver VAL on shared hosts. For periodic and sporadic tasks, we establish theoretical guarantees that VAS can achieve the same task schedule on a shared CPU as on a full CPU dedicated to time-sensitive services. Moreover, this can be achieved by allocating the minimal CPU bandwidth to time-sensitive services, thereby avoiding wasting CPU resources. VAS has been implemented on Xen 4.10.0. In case studies running time-sensitive workloads on Redis and Spark streaming services, we show that in practice the task schedule on a shared CPU can closely approximate the one on a full CPU.

6.1.3 M/D(DS)/1

When confronting stochastic tasks in edge cloud, we usually involve latency requirements in terms of probabilistic tail latency instead of hard deadlines. Cloud services need to handle aperiodic requests for stochastic arrival processes instead of traditional periodic or sporadic

models. Finally, the computing platform must provide performance isolation between time-critical services and other workloads. It is therefore essential to provision resources to meet different tail latency requirements. As a step towards cloud services with stochastic latency guarantees, in Chapter 4, we presents a stochastic response time analysis for aperiodic services following a Poisson arrival process on computing platforms that schedule time-critical services as deferrable servers. The stochastic analysis enables a service operator to provision CPU resources for aperiodic services to achieve a desired tail latency. We evaluated the method in two case studies, one involving a synthetic service and another involving a Redis service, both on a testbed based on Xen 4.10. The results demonstrate the validity and efficacy of our method in a practical setting.

6.1.4 RT-ZooKeeper

To adopt distributed applications from cloud environments to edge environment, a time-sensitive fault-tolerant coordination services should be established. Coordination services must recover from failures in a timely manner. We analyze the limitations of the leader election and recovery protocols underlying Apache ZooKeeper, the prevailing open-source coordination service. In Chapter 5, we propose RT-ZooKeeper which features: 1. a new fast convergence leader election protocol, 2. a fast quorum channel establishment protocol, and 3. a distributed epoch value persistence protocol. They reduce the latency in coordination service recovering from leader failures. We implement RT-Zookeeper based on ZooKeeper version 3.5.8. Empirical evaluation shows that RT-ZooKeeper can significantly reduce recovery latency. We also conduct case studies showing our fast failure recovery in RT-ZooKeeper can benefit a common messaging service like Kafka.

6.2 Closing Remarks

Edge computing brings about major advantages for building modern time-sensitive applications. However, it is not sufficient by just migrating those applications to the edge and expecting achieving timeliness automatically. The existing virtualization technology and coordination solutions cannot satisfy real-time requirements, which fundamentally requires changes in both the system architecture and protocols.

In this dissertation, we have presented four projects – M2-Xen, VAS, M/D(DS)/1 and RT-ZooKeeper – to support time-sensitive applications in edge environments. Our system and experimental studies exhibit the potential of the novel methodology for achieving responsiveness on edge. This dissertation therefore represents a promising step towards real-time virtualization and coordination in edge computing.

References

[1] Tarek F Abdelzaher, Vivek Sharma, and Chenyang Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Transactions on Computers*, 53(3):334–350, 2004.

[2] Saeed Abedi, Neeraj Gandhi, Henri Maxime Demoulin, Yang Li, Yang Wu, and Linh Thi Xuan Phan. Rtnf: Predictable latency for network function virtualization. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 368–379, Montreal, 2019. IEEE.

[3] Luca Abeni, Alessandro Biondi, and Enrico Bini. Hierarchical scheduling of real-time tasks over linux-based virtual machines. *Journal of Systems and Software*, 149:234–249, 2019.

[4] Luca Abeni and Dario Faggioli. An experimental analysis of the xen and kvm latencies. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 18–26. IEEE, 2019.

[5] Luca Abeni, Nicola Manica, and Luigi Palopoli. Efficient and robust probabilistic guarantees for real-time tasks. *Journal of Systems and Software*, 85(5):1147–1156, 2012.

[6] Amazon.com Inc. Amazon ElastiCache for Redis. `https://aws.amazon.com/elasticache/redis/`, 2018.

[7] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613, 2018.

[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[9] CM Bailey. Hard real-time operating system kernel. investigation of mode change. *British Aerospace Systems Ltd*, 1993.

[10] Jaiganesh Balasubramanian, Aniruddha Gokhale, Abhishek Dubey, Friedhelm Wolf, Chenyang Lu, Chris Gill, and Douglas Schmidt. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 69–78, Stockholm, Sweden, 2010. IEEE.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[12] Sanjoy K Baruah and Nathan Fisher. Component-based design in multiprocessor real-time systems. In *International Conference on Embedded Software and Systems (ICESS)*, 2009.

[13] Sanjoy K Baruah, Louis E Rosier, and Rodney R Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-time systems*, 2(4):301–324, 1990.

[14] Andrea Bastoni, Bjorn B Brandenburg, and James H Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS)*, 2010.

[15] Matthias Beckert and Rolf Ernst. Response time analysis for sporadic server based budget scheduling in real time virtualization environments. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):161, 2017.

[16] Matthias Beckert, Kai Björn Gemlau, and Rolf Ernst. Exploiting sporadic servers to provide budget scheduling for arinc653 based real-time virtualization environments. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 870–875. European Design and Automation Association, 2017.

[17] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120, Irvine, 2012. IEEE.

[18] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation*, pages 141–154, Boston, 2011. ACM.

[19] Richard J Boucherie and Nico M Van Dijk. *Queueing networks: a fundamental approach*, volume 154. Springer Science & Business Media, 2010.

[20] Abdeldjalil Boudjadar, Alexandre David, Jin Hyun Kim, Kim G Larsen, Marius Mikučionis, Ulrik Nyman, and Arne Skou. Hierarchical scheduling framework based

on compositional analysis using uppaal. In *International Workshop on Formal Aspects of Component Software*, pages 61–78. Springer, 2013.

[21] Alan Burns. System mode changes-general and criticality-based. In *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, pages 3–8, 2014.

[22] Alan Burns and Sanjoy Baruah. Towards a more practical model for mixed criticality systems. In *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

[23] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Seattle, 2006. ACM.

[24] John M Calandrino, Dan Baumberger, Tong Li, Scott Hahn, and James H Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 101–112. IEEE, 2007.

[25] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):1–1, 2015.

[26] Fabio Checconi, Tommaso Cucinotta, and Manuel Stein. Real-time issues in live migration of virtual machines. In *Euro-Par 2009 – Parallel Processing Workshops*, 2009.

[27] Jonathan Corbet. SCHED_FIFO and realtime throttling. `https://lwn.net/Articles/296419/`, 2008.

[28] Tommaso Cucinotta, Fabio Checconi, George Kousiouris, Kleopatra Konstanteli, Spyridon Gogouvitis, Dimosthenis Kyriazis, Theodora Varvarigou, Alessandro Mazzetti, Zlatko Zlatev, Juri Papay, et al. Virtualised e-learning on the irmos real-time cloud. *Service Oriented Computing and Applications*, 6(2):151–166, 2012.

[29] Dionisio de Niz and Linh TX Phan. Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[30] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[32] José Luis Díaz, Daniel F García, Kanghee Kim, Chang-Gun Lee, L Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 289–300. IEEE, 2002.

[33] Michael Drescher, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. A flattened hierarchical scheduler for real-time virtualization. In *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.

[34] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[35] M.J.A. Eenige, van. *Queueing systems with periodic service*. PhD thesis, Department of Mathematics and Computer Science, 1996.

[36] Ibrahim EL-Sanosi and Paul Ezhilchelvan. Improving zookeeper atomic broadcast performance when a server quorum never crashes. *EAI Endorsed Transactions on Energy Web*, 5(17):1–1, 2018.

[37] Jeremy P Erickson, Greg Coombe, and James H Anderson. Soft real-time scheduling in google earth. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 141–150. IEEE, 2012.

[38] Agner Krarup Erlang. The theory of probabilities and telephone conversations. *Nyt. Tidsskr. Mat. Ser. B*, 20:33–39, 1909.

[39] Neeraj Gandhi, Edo Roth, Robert Gifford, Linh Thi Xuan Phan, and Andreas Haeberlen. Bounded-time recovery for distributed real-time systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 110–123, Sydney, 2020. IEEE.

[40] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726–740, 2014.

[41] GHS-TECH. Green hills platform for in-vehicle infotainment. `https://www.ghs.com/products/auto_infotainment.html`. Accessed: 2021-02-01.

[42] GlobalLogic. Solution accelerators and services from globallogic. `https://www.globallogic.com/wp-content/uploads/2016/12/GlobalLogic-Nautilus-Platform.pdf`. Accessed: 2021-02-01.

[43] Racha Gouareb, Vasilis Friderikos, and Abdol-Hamid Aghvami. Virtual network functions routing and placement for edge cloud latency minimization. *IEEE Journal on Selected Areas in Communications*, 36(10):2346–2357, 2018.

[44] Giovani Gracioli, Antônio Augusto Fröhlich, Rodolfo Pellizzoni, and Sebastian Fischmeister. Implementation and evaluation of global and partitioned scheduling in a real-time os. *Real-Time Systems*, 49(6):669–714, 2013.

[45] Xiaozhe Gu, Arvind Easwaran, Kieu-My Phan, and Insik Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[46] Arpan Gujarati, Sergey Bozhko, and Björn B Brandenburg. Real-time replica consistency over ethernet with reliability bounds. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 376–389, Sydney, 2020. IEEE.

[47] Itamar Haber. 5 tips for running Redis over AWS. `https://redislabs.com/blog/5-tips-for-running-redis-over-aws/`, 2018.

[48] Song Han, Tao Gong, Mark Nixon, Eric Rotvold, and Kam-Yiu Lam. RT-DAP: A real-time data analytics platform for large-scale industrial process monitoring and control. In *IEEE ICII*, pages 59–68, 2018.

[49] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16. ACM, 2008.

[50] Pengcheng Huang, Pratyush Kumar, Nikolay Stoimenov, and Lothar Thiele. Interference constraint graph—a new specification for mixed-criticality systems. In *Emerging Technologies & Factory Automation (ETFA)*, 2013.

[51] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, pages 1–1, Boston, 2010. ACM.

[52] EL-Sanosi Ibrahim and Paul Ezhilchelvan. Improving zookeeper atomic broadcast performance by coin tossing. In *European Workshop on Performance Engineering*, pages 249–265, Berlin, 2017. Springer.

[53] RedHat Inc. Enabling rt-kvm for nfv workloads. `https://shorturl.at/tuyR6`, 2018.

[54] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 435–448, 2015.

[55] Flavio Junqueira. Last processed zxid set prematurely while establishing leadership. `https://issues.apache.org/jira/browse/ZOOKEEPER-790`, 2010. Accessed: 2020-10-01.

[56] Flavio Junqueira. Zab pre 1.0. `https://cwiki.apache.org/confluence/display/ZOOKEEPER/ Zab+Pre+1.0`, 2013. Accessed: 2020-10-01.

[57] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, Hong Kong, 2011. IEEE.

[58] G. A. Kaczynski, L. Lo Bello, and T. Nolte. Deriving exact stochastic response times of periodic tasks in hybrid priority-driven soft real-time systems. In *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*, pages 101–110, Sep. 2007.

[59] Robert Kaiser. Applicability of virtualization to embedded systems. In *Solutions on Embedded Systems*, pages 215–226. Springer, 2011.

[60] Robert Kaiser and Dieter Zöbel. Quantitative analysis and systematic parametrization of a two-level real-time scheduler. In *2009 IEEE Conference on Emerging Technologies & Factory Automation*, pages 1–8. IEEE, 2009.

[61] Babak Kalantari and André Schiper. Addressing the zookeeper synchronization inefficiency. In *International Conference on Distributed Computing and Networking*, pages 434–438, Mumbai, 2013. Springer.

[62] Samuel Karlin. *A first course in stochastic processes*. Academic press, 2014.

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

[64] Hyoseung Kim and Ragunathan Rajkumar. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2016.

[65] Hyoseung Kim, Shige Wang, and Ragunathan Rajkumar. vmpcp: A synchronization framework for multi-core virtual machines. In *2014 IEEE Real-Time Systems Symposium*, pages 86–95. IEEE, 2014.

[66] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, 2007.

[67] Mahadev Konar. Zookeeper servers should commit the new leader txn to their logs. `https://issues.apache.org/jira/browse/ZOOKEEPER-335`, 2011. Accessed: 2020-10-01.

[68] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, Athens, 2011. IEEE.

[69] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 93–102. ACM, 2012.

[70] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[71] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[72] Ron Larson and Bruce H Edwards. *Calculus*. Cengage Learning, 2009.

[73] Jaewoo Lee, Hoon Sung Chwa, Linh TX Phan, Insik Shin, and Insup Lee. MC-ADAPT: Adaptive task dropping with task-level mode switch in mixed-criticality scheduling. In *Embedded Software (EMSOFT)*, 2017.

[74] Kilho Lee, Minsu Kim, Hayeon Kim, Hoon Sung Chwa, Jinkyu Lee, and Insik Shin. Fault-resilient real-time communication using software-defined networking. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 204–215, Montreal, 2019. IEEE.

[75] John P Lehoczky. Real-time queueing theory. In *17th IEEE Real-Time Systems Symposium*, pages 186–195. IEEE, 1996.

[76] John P Lehoczky. Using real-time queueing theory to control lateness in real-time systems. *ACM SIGMETRICS Performance Evaluation Review*, 25(1):158–168, 1997.

[77] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global edf in linux. In *proc. of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11)*, pages 6–15. Citeseer, 2011.

[78] Joseph Y-T Leung and ML Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118, 1980.

[79] Haoran Li, Chenyang Lu, and Chistopher Gill. Predicting latency distributions of aperiodic time-critical services. In *RTSS*, pages 30–42. IEEE, 2019.

[80] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I Lee, Chenyang Lu, Kathryn S McKinley, et al. Work stealing for interactive services to meet target latency. In *ACM SIGPLAN Notices*, volume 51, page 14. ACM, 2016.

[81] Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. Predictable communication and migration in the quest-v separation kernel. In *2014 IEEE Real-Time Systems Symposium*, pages 272–283. IEEE, 2014.

[82] Ye Li, Richard West, and Eric Missimer. A virtualized separation kernel for mixed criticality systems. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 201–212, 2014.

[83] Tein-Hsiang Lin and Wernhuar Tarng. Scheduling periodic and aperiodic tasks in hard real-time computing systems. In *ACM SIGMETRICS performance evaluation review*, volume 19, pages 31–38. ACM, 1991.

[84] Jianhui Liu and Qi Zhang. Offloading schemes in mobile edge computing for ultra-reliable low latency communications. *Ieee Access*, 6:12825–12837, 2018.

[85] Shokunin Consulting LLC. Running Redis in production. `http://shokunin.co/blog/2014/11/11/operational_redis.html`, 2014.

[86] VMWare LLC. Deploying extremely latency-sensitive applications in vmware. `http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/latency-sensitive-perf-vsphere55-white-paper.pdf`, 2015.

[87] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[88] Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. A statistical response-time analysis of real-time embedded systems. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 351–362. IEEE, 2012.

[89] Eric Lubow. Redis setup notes and one-liners. `https://eric.lubow.org/2013/redis-setup-notes-and-one-liners-2/`, 2013.

[90] Lucidworks. Setting up an external zookeeper ensemble. `https://doc.lucidworks.com/fusion-server/4.2/reference/solr-reference-guide/7.5.0/setting-up-an-external-zookeeper-ensemble.html`, 2018. Accessed: 2020-10-01.

[91] Nancy A Lynch. *Distributed algorithms*. Elsevier, San Francisco, 1996.

[92] Nancy A Lynch. *Distributed algorithms*, chapter Algorithms in General Synchronous Networks, pages 51–80. Elsevier, San Francisco, 1996.

[93] Nancy A Lynch. *Distributed algorithms*, chapter Leader Election in an Arbitrary Network, pages 495–496. Elsevier, San Francisco, 1996.

[94] Dorin Maxim and Liliana Cucu-Grosjean. Response time analysis for fixed-priority tasks with multiple probabilistic parameters. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 224–235. IEEE, 2013.

[95] André Medeiros. Zookeeper's atomic broadcast protocol: Theory and practice. Technical report, Technical report, 2012.

[96] Alex F Mills and James H Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 311–320. IEEE, 2010.

[97] Eric Missimer, Richard West, and Ye Li. Distributed real-time fault tolerance on a virtualized multi-core system. *Operating Systems Platforms for Embedded Real-Time Application*, 2014.

[98] Malcolm S Mollison, Jeremy P Erickson, James H Anderson, Sanjoy K Baruah, and John A Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Computer and Information Technology (CIT)*, 2010.

[99] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[100] Gordon Frank Newell. Statistical analysis of the flow of highway traffic through a signalized intersection. *Quarterly of Applied Mathematics*, 13(4):353–369, 1956.

[101] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 385–398, 2013.

[102] Shuichi Oikawa and Ragunathan Rajkumar. Portable rk: A portable resource kernel for guaranteed and enforced timing behavior. In *RTAS*, pages 111–120. IEEE, 1999.

[103] Teunis J Ott. On the stationary waiting-time distribution in the GI/G/1 queue, i: transform methods and almost-phase-type distributions. *Advances in applied probability*, 19(1):240–265, 1987.

[104] Teunis J Ott. The single-server queue with independent GI/G and M/G input streams. *Advances in applied probability*, 19(1):266–286, 1987.

[105] Luigi Palopoli, Daniele Fontanelli, Nicola Manica, and Luca Abeni. An analytical bound for probabilistic deadlines. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 179–188. IEEE, 2012.

[106] Andrew Patterson. Infotainment and telematics solutions with renesas r-car. https://docplayer.net/9492007-Infotainment-and-telematics-solutions-with-renesas-r-car-course-id-0c18i.html. Accessed: 2021-02-01.

[107] Linh TX Phan, Samarjit Chakraborty, and PS Thiagarajan. A multi-mode real-time calculus. In *2008 Real-Time Systems Symposium*, pages 59–69. IEEE, 2008.

[108] Linh TX Phan, Insup Lee, and Oleg Sokolsky. Compositional analysis of multi-mode systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[109] Linh TX Phan, Jaewoo Lee, Arvind Easwaran, Vinay Ramaswamy, Sanjian Chen, Insup Lee, and Oleg Sokolsky. Carts: a tool for compositional analysis of real-time systems. *ACM SIGBED Review*, 8(1):62–63, 2011.

[110] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-time systems*, 26(2):161–197, 2004.

[111] RedisLabs. Introduction to Redis. https://redis.io/topics/introduction, 2018.

[112] Benjamin Reed. Zab 1.0. https://cwiki.apache.org/confluence/display/ZOOKEEPER/ Zab1.0, 2016. Accessed: 2020-10-01.

[113] Benjamin Reed and Norbert Kalmar. Applications powered by ZooKeeper. https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy, 2019.

[114] Jiankang Ren and Linh Thi Xuan Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[115] İzzet Şahin and U Narayan Bhat. A stochastic system with scheduled secondary inputs. *Operations Research*, 19(2):436–446, 1971.

[116] Maurice Sebastian, Philip Axer, and Rolf Ernst. Utilizing hidden markov models for formal reliability analysis of real-time communication systems with errors. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 79–88, Pasadena, 2011. IEEE.

[117] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[118] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[119] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13. IEEE, 2003.

[120] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *Real-Time Systems Symposium (RTSS)*, 2004.

[121] Insik Shin and Insup Lee. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):30, 2008.

[122] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10, Incline Village, 2010. IEEE.

[123] Jiguo Song, John Wittrock, and Gabriel Parmer. Predictable, efficient system-level fault tolerance in c^ 3. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 21–32, Vancouver, 2013. IEEE.

[124] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[125] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[126] Bogdan Tanasa, Unmesh D Bordoloi, Petru Eles, and Zebo Peng. Probabilistic response time and joint analysis of periodic tasks. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 235–246. IEEE, 2015.

[127] The Open Group. O-PAS standard version 2.0. *Feb*, 2020.

[128] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No. 00CH36353)*, volume 4, pages 101–104. IEEE, 2000.

[129] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[130] Ken Tindell and Alejandro Alonso. A very simple protocol for mode changes in priority preemptive systems. *Universidad Politécnica de Madrid, Tech. Rep*, 1996.

[131] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, Snowbird, 2014. ACM.

[132] SS Vallender. Calculation of the wasserstein distance between probability distributions. *Theory of Probability & Its Applications*, 18(4):784–786, 1974.

[133] Nico M. van Dijk. On the arrival theorem for communication networks. *Comput. Netw. ISDN Syst.*, 25(10):1135–1142, May 1993.

[134] Manohar Vanga, Arpan Gujarati, and Björn B Brandenburg. Tableau: a high-throughput and predictable vm scheduler for high-density workloads. In *EuroSys*, 2018.

[135] Elin Vinka. How many Zookeepers in a cluster? `https://www.cloudkarafka.com/blog/2018-07-04-cloudkarafka-how-many-zookeepers-in-a-cluster.html`, 2018. Accessed: 2020-10-01.

[136] Chao Wang, Christopher Gill, and Chenyang Lu. Frame: Fault tolerant and real-time messaging for edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 976–985, Dallas, 2019. IEEE.

[137] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT)*, 2011.

[138] Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in Xen. In *2014 International Conference on Embedded Software (EMSOFT)*, 2014.

[139] Meng Xu. RTDS-based-scheduler. `https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler`, 2015.

[140] Meng Xu, Linh Thi Xuan Phan, Oleg Sokolsky, Sisu Xi, Chenyang Lu, Christopher Gill, and Insup Lee. Cache-aware compositional analysis of real-time multicore virtualization platforms. *Real-Time Systems*, 51(6):675–723, 2015.

[141] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. vcat: Dynamic cache management using cat virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222. IEEE, 2017.

[142] Ying Ye, Richard West, Jingyi Zhang, and Zhuoqun Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *Real-Time Systems Symposium (RTSS)*, 2016.

[143] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[144] Lin Zhang, Xin Chen, Fanxin Kong, and Cardenas Alvaro. Real-time recovery for cyber-physical systems using linear approximations. In *Real-Time Systems Symposium (RTSS), 2020 IEEE*, pages 1–1, Houston, 2020. IEEE.

[145] Ming Zhao and Jorge Cabrera. Rtvirt: enabling time-sensitive computing on virtualized systems through cross-layer cpu scheduling. In *Proceedings of the Thirteenth EuroSys Conference*, page 27. ACM, 2018.

[146] Timothy Zhu, Daniel S Berger, and Mor Harchol-Balter. Snc-meister: Admitting more tenants with tail latency slos. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 374–387, 2016.

Vita

Haoran Li

Degrees

B.S. Electronic Information Science and Technology, July 2011
M.S. Information Science and Communication Engineering, July 2014

Professional

Real-Time Virtualization and Distributed Systems

Publications

H. Li, M. Xu, C. Li, C. Lu, C. Gill, L. Phan, I. Lee, and O. Sokolsky, Towards Virtualization-Agnostic Latency for Time-Sensitive Applications, RTNS 2021

H. Li, C. Lu, and C. Gill (2019). Predicting Latency Distributions of Aperiodic Time-Critical Services, RTSS 2019.

M. Xu, L. Phan, H. Choi, Y. Lin. **H. Li**, C. Lu, and I. Lee. Holistic Resource Allocation for Multicore Real-Time Systems, RTAS 2019 (Best Paper Award).

H. Li, M. Xu, C. Li, C. Lu, C. Gill, L. Phan, I. Lee, and O. Sokolsky. Multi-Mode Virtualization for Soft Real-Time Systems, RTAS 2018.

J. Li, **H. Li**, Y. Ma, Y. Wang, A. Abokifa, C. Lu, and P. Biswas. Spatiotemporal distribution of indoor particulate matter concentration with a low-cost sensor network, Building and Environment 2018.

May 2021