ABSTRACT

QIAN, TAO. End-to-end Predictability for Distributed Real-Time Systems. (Under the direction of Rainer Frank Mueller.)

Distributed real-time systems impose a number of requirements, such as scalability, reliability, predictability, and high computational throughput. These demands can be met by designing protocols for participating nodes and network infrastructures specifically for these requirements. Unlike other distributed systems whose main objective is to provide high throughput or low request latency, our system addresses one additional significant requirement raised by real-time systems, namely end-to-end response time predictability for distributed real-time systems. Thus, the goal of our system is to execute distributed real-time tasks within their deadlines imposed by the clients. One of the potential applications is a wide-area monitoring system (WAMS) for power grid state monitoring and control. In such a system, state monitoring devices periodically gather data from the power grid infrastructure scattered over a wide area and the WAMS subsequently requests these data in a timely fashion.

The followings are the key contributions of this work:

1. To support storing and fetching wide-area monitoring data in a timely fashion, we implement a real-time storage system based on the Chord protocol. We employ an application-level task scheduler to arbitrate the execution of the distributed tasks for wide-area monitoring. In addition, we formally define the pattern of the workload and use queueing theory to stochastically derive the time bound for response times of these data requests. On the basis of the queueing model, the system can provide probabilistic deadline guarantees for data requests.

2. To realize the deadlines of data requests and schedule urgent requests first, we implement an earliest deadline first (EDF) based packet scheduler to transmit data requests via the underlying network infrastructures. This packet scheduler runs on end nodes. In addition, we utilize the hierarchical token bucket (HTB) traffic control mechanism provided by Linux to implement a bandwidth sharing policy. This policy regulates the message traffic that is transmitted via the network. The hybrid scheduler decreases the variance of network transmission delays, which increases the predictability of end-to-end delays for distributed real-time tasks.

3. Compared to tasks with soft deadlines, tasks with hard deadlines have more severe consequences when they miss their deadlines. To meet the hard deadline requirements for distributed real-time tasks, packet scheduling of the network infrastructure utilized by

communicating tasks must be managed actively. We implement a static routing algorithm to derive the forwarding paths for real-time packets according to the network capability (i.e., buffer size and computation speed of network devices). In addition, we implement a real-time packet scheduler, which runs on network devices to enforce the packet scheduling policy derived by the routing algorithm under stable network conditions.

Overall, these mechanisms allow clients to impose hard deadlines on their distributed real-time tasks so they can focus on high-level application design instead of predictability concerns of the underlying communication infrastructures.

End-to-end Predictability for Distributed Real-Time Systems

by
Tao Qian

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

Aranya Chakrabortty	Matthias Stallmann
Guoliang Jin	Rainer Frank Mueller Chair of Advisory Committee

BIOGRAPHY

Tao Qian was born in Mianyang, Sichuan, China. He received his Bachelor's and Master's degree in Computer Science from Zhejiang University. He then attended NC State University from 2012 to 2017 for Ph.D study.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Frank Mueller, for his continuous support to my study. I would like to thank Dr.Aranya Chakrabortty, Dr. Guoliang Jin, and Dr. Matt Stallmann for serving my dissertation committee. I would like to thank my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Real-Time Systems

Real-Time systems are computing environments in which the design of the system focuses on not just the functional correctness but also temporal correctness. This means that real-time systems provide execution platforms to running applications that can schedule tasks according to their temporal requirements. A real-time system is temporally correct if all the timing requirements of real-time tasks can be met. Control systems often impose real-time requirements to operate automobiles, aerial vehicles, nuclear systems, etc. A delayed response in such systems could impact the quality of service or even result in catastrophic consequences. For example, the anti-lock braking system, which monitors the brakes of automobiles and must react within milliseconds, could result in life loss if the system cannot respond in time and cause the wheels to lock-up for too long a time. Depending upon the purpose of the systems and the implications of system correctness, real-time systems can be divided into two categories. $Soft$ real-time systems can tolerate some misses of their temporal requirements, but eventually the quality of service will degrade if too many are missed. For example, an online video game that requires multiple players to cooperate in a timely fashion could provide a lagged visual presentation and delay the actions of players if the system misses the deadline of information exchanged between the players. This eventually decreases the quality of the game experience. In contrast, *hard* real-time systems must absolutely guarantee the temporal requirements of the applications to prevent catastrophic consequences. These applications are safety critical. The anti-lock braking system is one such system.

Real-time systems have a different design goals than other systems whose performance metric is often the efficiency of the systems (e.g., input/output operations per second for general storage systems). In real-time systems, tasks impose deadline requirements on their response time. This means that the execution of the task (i.e., a job) must be finished before its deadline.

One effective goal for real-time systems is to avoid deadline misses for hard real-time tasks and reduce the deadline miss rate for soft real-time tasks. To achieve this, real-time systems often employ real-time *schedulers* to execute these jobs. A real-time scheduler can recognize the temporal requirements of these tasks and execute their jobs in a specific order to minimize the deadline miss rate. For example, a cyclic executive executes real-time jobs at the time specified in a schedule table [34, 5]. The schedule table is pre-calculated in a way that the task deadlines can be guaranteed. An earliest deadline first (EDF) scheduler executes the job with the shortest deadline first [34, 33].

One can increase the *predictability* of real-time systems by adopting real-time schedulers, since these schedulers guarantee specific orders of real-time jobs. The predictability of a real-time system subsequently facilitates the derivation of an upper bound for the job response time and deadline miss rate off-line when the workload is known a priori. This is essential for the clients of the real-time system to determine whether their tasks can be executed within a given deadline. A *schedulability test* is a mathematical model to determine the predictability of real-time systems.

1.2 Distributed Real-Time Systems

Distributed systems allow compute nodes to communicate via passing messages over the underlying network infrastructures. In such systems, the computation of distributed tasks requires coordination between resources located on different components. As an example, a credit payment system is a distributed system in the sense that it requires the credit card reader to communicate with the servers of payment organizations (e.g, credit card organizations and banks) to validate the credit card information and finalize the transaction. Scalability and resilience are two major concerns of designing distributed systems. Scalability defines the capability of a distributed system to accommodate the increasing demand in computation and communication. Resilience requires a distributed system to recover quickly from component failures.

In addition to the scalability and resilience requirements for distributed systems, distributed real-time systems have temporal constricts for their tasks. One example of a distributed real-time system is the monitoring and control system for the North America power grid. In this power grid, a large number of wind and photo-voltaic generation devices are deployed across a wide area. These devices require real-time monitoring and actuation to ensure sustained reliability. Thus, the North America power grid uses Wide-Area Monitoring System (WAMS) technology to monitor and control the state of the power grid [45, 38]. Phasor Measurement Units (PMU) are installed in the power grid to collect the real-time monitoring data. These data are sent to Phasor Data Concentrators (PDC) in a timely fashion so that they can be processed such that control feedback is sent back to the power grid infrastructure. Fig. 1.1

illustrates a setup of WAMS. PDCs and PMUs are connected via a communication network infrastructure to transmit real-time data. The system must meet the temporal requirements of the data transmission and task execution on PDCs and PMUs to provide a reliable service.

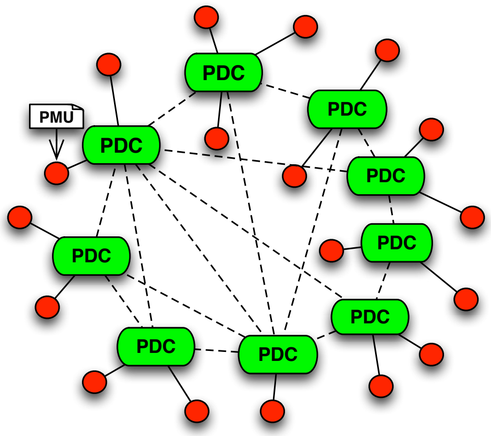

Figure 1.1: Wide-Area Monitoring System

Fig. 1.2 depicts the interaction of components within this distributed real-time system. End nodes $R1$, $R2$, $R3$, and $R4$ are connected by the underlying network, which utilizes network switches/routers to transmit messages. A message sent by a distributed task on one end node needs to pass through multiple layers of the system to reach the task on another node. First, the distributed task in the application layer injects a message via the network stack implementation on the end node. Second, the message is forwarded by network devices to reach the destination node. Then, the message is received by the network stack on the destination node and relayed to the corresponding distributed task.

All the layers in the system need to abide by the temporal requirements of distributed real-time tasks and schedule respective entities accordingly. A *task scheduler* is a scheduler running at the distributed application layer on the end node to execute real-time jobs. A *packet scheduler* runs either on the end-node network stack to inject real-time messages into the network or on the network devices to forward real-time messages to their destinations. These schedulers must cooperate to guarantee end-to-end deadlines of real-time tasks. The task scheduler and packet scheduler on end nodes cannot actively control the behavior of the network infrastructure and limit the impact of network traffic imposed by other end nodes that share the network resources with the real-time tasks. Thus, commodity networking devices are only sufficient for distributed real-time systems with soft deadlines (e.g, probabilistic deadlines). To meet the requirements of

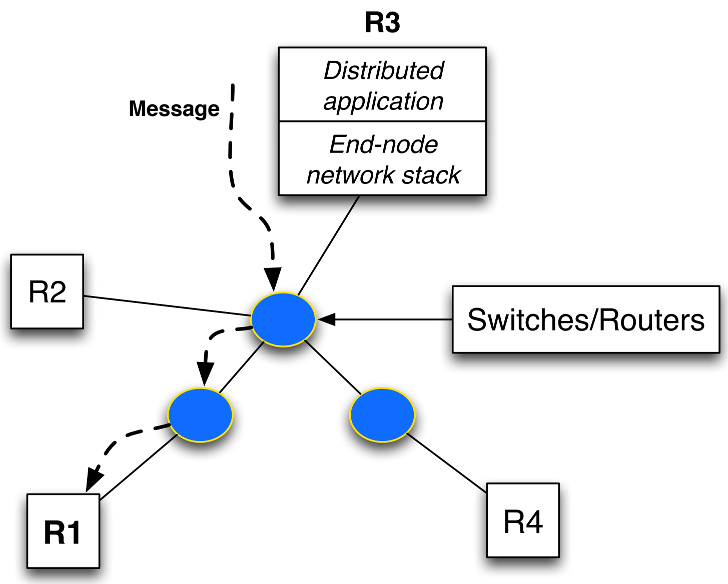

Figure 1.2: Distributed Systems Abstract

real-time systems with hard deadlines, a packet scheduler on network devices is required. This packet scheduler can prioritize packets according to their temporal requirements and avoid dropping them when network contention occurs. The focus of this work is to implement real-time schedulers on all layers to improve the predictability for distributed real-time systems.

1.3 Temporal Requirements

Real-time tasks can be executed periodically or sporadically. Periodic tasks have regular release times and sporadic tasks have irregular release times. The term *job* represents a release of a real-time task. This work assumes real-time tasks to be periodic. A periodic task has the following temporal properties:

1. *Phase* denotes the release time of the first job of the task.
2. *Period* denotes the inter-arrival time between two consecutive releases of the task.
3. *WCET* denotes the worst-case execution time of a job of the task,
4. *Deadline* denotes the time instance by which the job must be finished.
5. *Relative Deadline* denotes the time difference between the release time and the deadline of the job. Jobs of the same task have the same relative deadline.

6. *Response Time* denotes the time difference between the release time and the time when the job is finished.

Periodic tasks pass messages to communicate with each other in a periodic manner. These messages can be organized as message flows. One message flow includes the periodic messages between two periodic tasks located on different end nodes. Similar to real-time tasks, a message flow has the following properties:

1. *Period* denotes the time interval that an end node transmits two consecutive messages from the same flow to the network.

2. *Deadline* denotes the time instance by when the message has to arrive at its destination node.

3. *Relative Deadline* denotes the time difference between the release time and the deadline of the message.

4. *Size* denotes the size of the packet that carries the message.

These properties of real-time tasks and message flows are utilized in the schedulability test to ensure that their deadline requirements can be met by the schedulers.

1.4 Contributions

This dissertation assesses if the end-to-end predictability for distributed real-time systems can be provided by controlling the behavior of different layers of the systems. The following are the major contributions in this document:

1. In Chapter 2, we present a real-time distributed storage system to support the Wide-Area Monitoring System for the North America power grid [47, 48]. Our storage system adopts an application-level task scheduler, a cyclic executive, to execute the periodic tasks to store and retrieve monitoring data as a real-time service to an upper layer decentralized monitoring and control system. This chapter puts forth a fundamentally new approach to derive the probabilistic upper bound for the response times of the data requests. This approach formally defines the pattern of the workloads in such monitoring systems and uses queueing theory to derive a stochastic model to predict the probabilistic response times. Our experimental evaluation on distributed nodes shows that our model is well suited to provide time bounds for data requests following the typical workload pattern in the monitoring system.

2. In Chapter 3, we present a hybrid EDF packet scheduler for real-time distribute systems to increase the predictability of packet scheduling on end nodes [49]. This work enhances the storage system by realizing the urgency (i.e., deadlines of requests) and scheduling these tasks accordingly. Employing EDF scheduling in distributed real-time systems has many challenges. First, the network layer of a resource has to notify the task scheduler about the deadlines of received messages. The randomness of interrupts due to such notifications makes context switch costs unpredictable. Second, communication delay variances in a network increase the timing unpredictability of packets, e.g., when multiple resources transmit message bursts simultaneously. To address these challenges, we implement an EDF-based packet scheduler, which transmits packets considering event deadlines. Then, we employ bandwidth limitations on the transmission links of resources to decrease network contention and network delay variance. We evaluate this work on a cluster of nodes in a switched network environment resembling a distributed cyber-physical system to demonstrate the real-time capability of the hybrid scheduler.

3. In Chapter 4, we present a static routing algorithm and a packet scheduler on network devices to transmit real-time messages via the underlying networks [50]. First, the static routing algorithm derives forwarding paths for real-time packets to guarantee their hard deadlines. This algorithm employs a validation-based backtracking procedure, which considers both the demand of real-time message transmissions and the limitation of network resource capabilities. The algorithm can check whether this demand can be met on the network devices. Second, the packet scheduler transmits real-time messages according to our routing requirements. In addition, the packet scheduler guarantees that only packets with no temporal requirements can be dropped when network contention occurs. We evaluate this work in a local cluster to demonstrate the feasibility and effectiveness of our static routing algorithm and packet scheduler.

1.5 Hypothesis

Distributed computation environments provide real-time systems an opportunity to utilize large amounts of resources on different computing hardware. In addition, some real-time systems are naturally distributed systems, such as cyber-physical systems, which are rapidly being developed today. This imposes additional requirements on the communication techniques for real-time tasks located on different computing components to cooperate. Compared to the real-time systems running on single or multiprocessor platforms, distributed real-time systems also require the underlying network infrastructure to support their temporal requirements. Current Ethernet-based networks have the following shortcomings:

1. Unpredictability of transmission due to dropped packets when network contention occurs, which potentially increases the rate of deadline misses.

2. Coarse-grained quality of service support, which limits the capability of real-time systems.

3. Lack of support for temporal requirements when routing process is required to establish forwarding paths.

We attempt to extend the current network infrastructures to address these shortages in this dissertation. The hypothesis of this dissertation is as follows.

Distributed real-time systems can increase their predictability when the predictability of all involved system layers (i.e., end-nodes and connecting network infrastructure) is increased. The deployment of real-time task and packet schedulers provides a fine-grained support to temporal requirements of distributed real-time tasks on both end nodes and network devices to effectively increase this predictability.

1.6 Organization

This document is structured as follows. Chapter 2 introduces a real-time distributed hash table, which utilizes a real-time task scheduler to achieve probabilistic deadlines. Chapter 3 presents the hybrid EDF packet scheduler and bandwidth sharing policy running on end nodes to reduce network variance. Chapter 4 contributes the static routing algorithm and packet scheduler on network devices to guarantee real-time transmission deadlines when the network is in a stable state. Chapter 5 discusses the open problems for future work and summarizes this work.

Chapter 2

A Real-time Distributed Hash Table

2.1 Introduction

The North American power grid uses Wide-Area Monitoring System (WAMS) technology to monitor and control the state of the power grid [45, 38]. WAMS increasingly relies on Phasor Measurement Units (PMUs), which are deployed to different geographic areas, e.g., the Eastern interconnection area, to collect real-time power monitoring data per area. These PMUs periodically send the monitored data to a phasor data concentrator (PDC) via proprietary Internet backbones. The PDC monitors and optionally controls the state of the power grid on the basis of the data. However, the current state-of-art monitoring architecture uses one centralized PDC to monitor all PMUs. As the number of PMUs is increasing extremely fast nowadays, the centralized PDC will soon become a bottleneck [12]. A straight-forward solution is to distribute multiple PDCs along with PMUs, where each PDC collects real-time data from only the part of PMUs that the PDC is in charge of. In this way, the PDC is not the bottleneck since the number of PMUs for each PDC could be limited and new PDCs could be deployed to manage new PMUs.

However, new problems arise with such a distributed PDC architecture. In today's power grid, real-time control actuation relies in part on the grid states of multiple areas [42], but with the new architecture the involved PMUs could be monitored by different PDCs. The first problem is how to manage the mapping between PDCs and PMUs, so that a PDC can obtain PMU data from other PDCs. The second problem is how to communicate between these PDCs, so that the real-time bounds on control operation are still guaranteed. For simplification, we consider these PDCs as distributed storage nodes over a wide-area network, where each of them periodically generates records as long as the PDC periodically collects data from the PMUs in that area. Then, the problem becomes how to obtain these records from the distributed storage system with real-time bounded response times.

Our idea is to build a distributed hash table (DHT) on these distributed storage nodes to solve the first problem. Similar to a single node hash table, a DHT provides *put(key, value)* and *get(key)* API services to upper layer applications. In our DHT, the data of one PMU are not only stored in the PDC that manages the PMU, but also in other PDCs that keep redundant copies according to the distribution resulting from the hash function and redundancy strategy. DHTs are well suited for this problem due to their superior scalability, reliability, and performance over centralized or even tree-based hierarchical approaches. The performance of the service provided by some DHT algorithms, e.g., Chord [40] and CAN [52], decreases only insignificantly while the number of the nodes in an overlay network increases. However, there is a lack of research on real-time bounds for these DHT algorithms for end-to-end response times of requests. Timing analysis is the key to solve the second problem. On the basis of analysis, we could provide statistical upper bounds for request times. Without loss of generality, we use the term *lookup request* or *request* to represent either a *put request* or a *get request*, since the core functionality of DHT algorithms is to look up the node that is responsible for storing data associated with a given key.

It is difficult to analyze the time to serve a request without prior information of the workloads on the nodes in the DHT overlay network. However, in our research, requests follow a certain pattern, which makes the analysis concrete. For example, PMUs periodically send real-time data to PDCs [38], so that PDCs issue put requests periodically. At the same time, PDCs need to fetch data from other PDCs to monitor global power states and control the state of the entire power grid periodically. Using these patterns of requests, we design a real-time model to describe the system. Our problem is motivated by the power grid situation but our abstract real-time model provides a generic solution to analyze response time for real-time applications over network. We further apply queuing theory to stochastically analyze the time bounds for requests. Stochastic approaches may break with traditional views of real-time systems. However, cyber-physical systems rely on stock Ethernet networks with TCP/IP enhanced, so that new soft real-time models for different QoS notions, such as simplex models [10, 4], are warranted.

Contributions: We have designed and implemented a real-time DHT by enhancing the Chord algorithm. Since a sequence of nodes on the Chord overlay network are involved to serve a lookup request, each node is required to execute one corresponding job. Our real-time DHT follows a cyclic executive [34, 5] to schedule these jobs on each node. We analyze the response time of these sub-request jobs on each node and aggregate them to bound the end-to-end response time for all requests. We also use a stochastic model to derive the probability that our real-time DHT can guarantee deadlines of requests. We issue a request pattern according to the needs of real-time distributed storage systems, e.g., power grid systems, on a cluster of workstations to evaluate our model. In general, we present a methodology for analyzing the real-time response time of requests served by multiple nodes on a network. This methodology includes: 1) employing real-time executives on nodes, 2) abstracting a pattern of requests, and 3) using a stochastic model to analyze response time bounds under the cyclic executive and the given request pattern. The real-time executive is not limited to a cyclic executive. For example, we show that a prioritized extension can increase the probability of meeting deadlines for requests that have failed in their first time execution. This work, while originally motivated by novel control methods in the power grid, generically applies to distributed control of CPS real-time applications.

The rest of the paper is organized as follows. Section 2.2 presents the design and implementation details of our real-time DHT. Section 2.3 presents our timing analysis and quality of service model. Section 2.4 presents the evaluation results. Section 2.5 presents the related work. Section 2.6 presents the conclusion and on-going part of our research.

2.2 Design and Implementation

Our real-time DHT uses the Chord algorithm to locate the node that stores a given data item. Let us first summarize the Chord algorithm and characterize the lookup request pattern (generically and specifically for PMU data requests). After that, we explain how our DHT implementation uses Chord and cyclic executives to serve these requests.

2.2.1 Chord

Given a particular key, the Chord protocol locates the node that the key maps to. Chord applies consistent hashing to assign keys to Chord nodes organized as a ring network overlay [40, 25]. The consistent hash uses a base hash function, such as SHA-1, to generate an identifier for each node by hashing the node's IP address. When a lookup request is issued for a given key, the consistent hash uses the same hash function to encode the key. It then assigns the key to the first node on the ring whose identifier is equal to or follows the hash code of that key. This node is called the successor node of the key, or the target node of the key. Fig. 2.1 depicts an example of a Chord ring in the power grid context, which maps 9 PDCs onto the virtual nodes on the ring (labels in squares are their identifiers). As a result, PDCs issue requests to the Chord ring to periodically store and obtain PMU data from target PDCs. This solves the first problem that we have discussed in Section 2.1.

It is sufficient to locate the successor node of any given key by maintaining the nodes in a wrap-around circular list in which each node has a reference to its successor node (depicted as bold lines that connect the nodes in Fig. 2.1). The lookup request is passed along the list until it encounters one node whose identifier is smaller than the hash code of the key but the identifier of the successor node is equal to or follows the hash code of the key. The successor

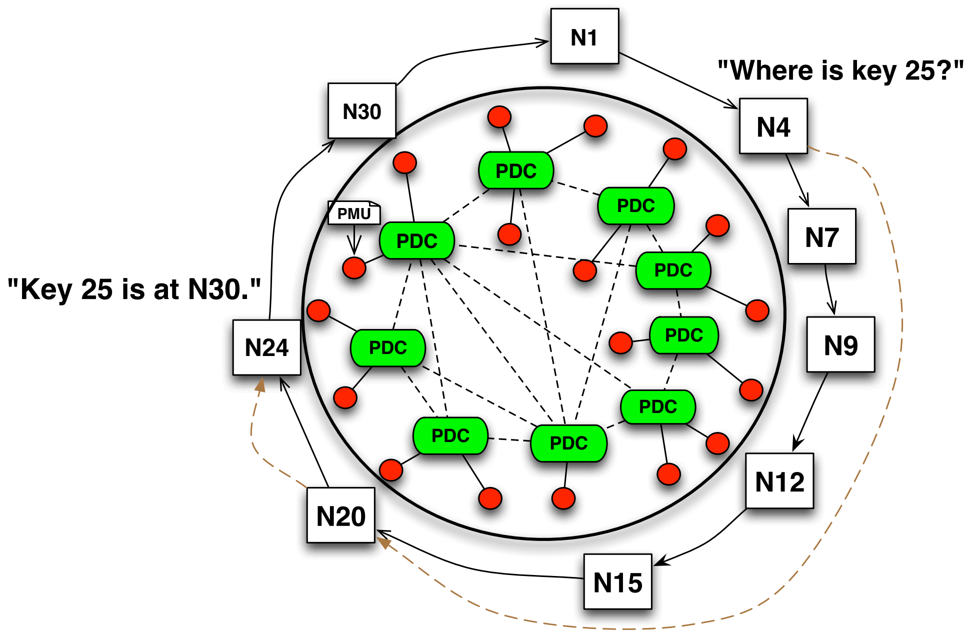

Figure 2.1: PDC and Chord Ring Mapping Example

of the node is the target node of the given key. Consider Fig. 2.1. The starting node $N4$ needs to locate key 25 (i.e., $K25$). By forwarding lookup messages along the circular list, $N4$ locates $N30$ as the target node of $K25$ via intermediate nodes *N7, N9, N12, N15, N20, N24*. However, the linear traversal requires *O(N)* messages to find the successor node of a key, where N is the number of the nodes.

This linear algorithm is not scalable with increasing numbers of nodes. In order to reduce the number of intermediate nodes, Chord maintains a so-called finger table on each node, which acts as a soft routing table, to decide the next hop to forward lookup requests to. For example, assuming 5-bit numbers are used to represent node identifiers in Fig. 2.1, the finger table for *N4* is in Table 2.1. The interval for the 5^{th} finger is (20, 36], which is (20, 4] as the virtual nodes are organized as a ring (modulo 32). In general, a finger table has $log(N)$ entries, where N is the number of bits of node identifiers. For node k, the start of the i^{th} finger table entry is $k + 2^{i-1}$, the interval is $(k + 2^{i-1}, k + 2^{i}]$, and the forward node is the successor of key $k + 2^{i-1}$ (i.e., the beginning of the interval). Each entry in the table indicates one routing rule: the next hop for a given key is the forward node in that entry if the entry interval includes the key. For example, the next hop for *K25* is *N20* as *25* is in *(20, 4]*.

To serve a lookup request for a given key, Chord first finds its predecessor, which is the first node in the counter-clockwise direction on the ring before the target node of that key, e.g., the predecessor of *K25* is *N24*. To find the predecessor, the start point *N4* forwards *K25* to

Table 2.1: Finger table for *N4*

#	**start**	**interval size**	**interval**	**forward node**
1	5	1	(5, 6]	N7
2	6	2	(6, 8]	N7
3	8	4	(8, 12]	N9
4	12	8	(12, 20]	N12
5	20	16	(20, 4]	N20

Table 2.2: Finger table for *N20*

#	**start**	**interval size**	**interval**	**forward node**
1	21	1	(21, 22]	N24
2	22	2	(22, 24]	N24
3	24	4	(24, 28]	N24
4	28	8	(28, 4]	N30
5	4	16	(4, 20]	N4

N20 according to its finger table (shown as the dotted line in Fig. 2.1). Next, *N20* forwards $K25$ to $N24$ according to $N20$'s finger table (depicted in Table 2.2). $N24$ considers itself the predecessor of *K25* since *K25* is between *N24* and its successor *N30.* Chord returns the successor *N30* of the predecessor *N24* as the target node of *K25.* In this example, only two intermediate nodes $N20$ and $N24$ are required to locate $K25$ via the finger tables. In general, each message forwarding reduces the distance to the target node to at least half that of the previous distance on the ring. Thus, the number of intermediate nodes per request is at most $log(N)$ with high probability [40], which is more scalable than the linear algorithm.

In addition to scalability, Chord tolerates high levels of churn in the distributed system. which makes it possible to provide PMU data with the possibility that some PDC nodes or network links have failed in the power grid environment. However, resilience analysis with real-time considerations is beyond the scope of this paper. Thus, our model in this paper derives the response times for requests assuming that nodes do not send messages in the background for resilience purpose.

2.2.2 The pattern of requests and jobs

To eventually serve a request, a sequence of nodes are involved. The node that issues a request is called the initial node of the request, and other nodes are called subsequent nodes. In the previous example, *N4* is the initial node; *N20, N24, N30* are subsequent nodes. As stated in Section 2.1, requests in the power grid system follow a certain periodic pattern, which makes timing analysis feasible. In detail, each node in the system maintains a list of get and put tasks together with the period of each task. Each node releases periodic tasks in the list to issue lookup requests. Once a request is issued, one job on each node on the way is required to be executed until the request is eventually served.

Jobs on the initial nodes are periodic, since they are driven by lists of periodic requests. However, jobs on the subsequent nodes are not necessarily periodic, since they are driven by the lookup messages sent by other nodes via the network. The network delay to transmit packets is not always constant, even one packet is enough to forward a request message as the length of PMU data is small in the power grid system. Thus, the pattern of these subsequent jobs depends on the node-to-node network delay. With different conditions of the network delay, we consider two models.

(1) In the first model, the network delay to transmit a lookup message is constant. Under this assumption, we use periodic tasks to handle messages sent by other nodes. For example, assume node A has a periodic PUT task $\tau_1(0, T_1, C_1, D_1)$. We use notation $\tau(\phi, T, C, D)$ to specify a periodic task τ, where ϕ is its phase, T is the period, C is the worst case execution time per job, and D is the relative deadline. From the schedule table for the cyclic executive, the response times for the jobs of τ_1 in a hyperperiod H are known [34]. Assume two jobs of this task, $J_{1,1}$ and $J_{1,2}$, are in one hyperperiod on node A, with $R_{1,1}$ and $R_{1,2}$ as their release times relative to the beginning of the hyperperiod and $W_{1,1}$ and $W_{1,2}$ as the response times, respectively.

Let the network latency be the constant δ. The subsequent message of the job $J_{1,1}$ is sent to node B at time $R_{1,1} + W_{1,1} + \delta$. For a sequence of hyperperiods on node A, the jobs to handle these subsequent messages on node B become a periodic task $\tau_2(R_{1,1} + W_{1,1} + \delta, H, C_2, D_2)$. The period of the new task is H as one message of $J_{1,1}$ is issued per hyperperiod. Similarly, jobs to handle the messages of $J_{1,2}$ become another periodic task $\tau_3(R_{1,2} + W_{1,2} + \delta, H, C_2, D_2)$ on node B.

(2) In the second model, network delay may be variable, and its distribution can be measured. We use aperiodic jobs to handle subsequent messages. Our DHT employs a cyclic executive on each node to schedule the initial periodic jobs and subsequent aperiodic jobs, as discussed in the next section.

We focus on the second model in this paper since this model reflects the networking properties of IP-based network infrastructure. This is also consistent with current trends in cyber-physical systems to utilize stock Ethernet networks with TCP/IP enhanced by soft real-time models for different QoS notions, such as the simplex model [10, 4]. The purely periodic first model is subject to future work as it requires special hardware and protocols, such as static routing and TDMA to avoid congestion, to support constant network delay.

2.2.3 Job scheduling

Table 2.3: Types of Messages passed among nodes

Type [1]	**Parameters** [2]	**Description**
PUT	:key:value	the initial put request to store (key:value).
PUT_DIRECT	:ip:port:sid:key:value	one node sends to the target node to store (key:value). [3]
PUT_DONE	:sid:ip:port	the target node notifies sender of successfully handling the PUT_DIRECT.
GET	:key	the initial get request to get the value for the given key.
GET_DIRECT	:ip:port:sid:key	one node sends to the target node to get the value.
GET_DONE	:sid:ip:port:value	the target node sends the value back as the feed back of the GET_DIRECT.
GET_FAILED	:sid:ip:port	the target node has no associated value.
LOOKUP	:hashcode:ip:port:sid	the sub-lookup message. [4]
DESTIN	:hashcode:ip:port:sid	similar to LOOKUP, but DESTIN message sends to the target node.
LOOKUP_DONE	:sid:ip:port	the target node sends its address back to the initial node.

[1] Message types for Chord *fix_finger* and *stabilize* operations are omitted.
[2] One message passed among the nodes consists of its type and the parameters, e.g, PUT:PMU-001:15.
[3] *(ip:port)* is the address of the node that sends the message. Receiver uses it to locate the sender. *sid* is unique identifier.
[4] Nodes use finger tables to determine the node to pass the LOOKUP to. *(ip:port)* is the address of the request initial node.

Each node in our system employs a cyclic executive to schedule jobs using a single thread. The schedule table $L(k)$ indicates which periodic jobs to execute at a specific frame k in a hyperperiod of F frames. L is calculated offline from the periodic tasks parameters [34]. Each node has a FIFO queue of released aperiodic jobs. The cyclic executive first executes periodic jobs in the current frame. It then executes the aperiodic jobs (up to a given maximum time allotment for aperiodic activity, similar to an aperiodic server). If the cyclic executive finishes all the aperiodic jobs in the queue before the end of the current frame, it waits for the next timer interrupt. Algorithm 1 depicts the work of the cyclic executive, where f is the interval of each frame.

In addition to jobs that handle request messages, our DHT uses a task to receive messages

```
Data: $L$, aperiodic job queue $Q$, current job
1  Procedure SCHEDULE
2      $current \leftarrow 0$
3      setup timer to interrupt every f time, execute TIMER_HANDLER when it interrupts
4      while true do
5          $current\ job \leftarrow L(current)$
6          $current = (current + 1)\%F$
7          if $current\ job \neq nil$ then
8              execute current job
9              mark current job done
10         end
11         while $Q$ is not empty do
12             $current\ job \leftarrow$ remove the head of $Q$
13             execute current job
14             mark current job done
15         end
16         $current\ job \leftarrow nil$
17         wait for next timer interrupt
18     end
19 Procedure TIMER_HANDLER
20     if $current\ job \neq nil$ and not done then
21         if current job is periodic then
22             mark the current job failed
23         else
24             save state and push current job back to the head of $Q$
25         end
26     end
27     jump to Line 4
```

Algorithm 1: Pseudocode for the cyclic executive

from other nodes. The cyclic executive schedules this receiving task as just another periodic task. This receiving task periodically moves the messages from the buffer of the underlying network stack to the aperiodic job queue. This design avoids the cost of context switches between the cyclic executive and another thread to receive messages, which could interrupt the cyclic executive at a random time whenever a message arrives. As a side effect, messages can only be scheduled in the next frame that the periodic receiving job is executed. This increases the response times of aperiodic jobs. Since the frame size is small, in the level of milliseconds, this delay is acceptable for our application domain.

2.2.4 Real-time DHT

Our real-time DHT is a combination of Chord and cyclic executive to provide predictable response times for requests following the request pattern. Our DHT adopts Chord's algorithm to locate the target node for a given key. On the basis of Chord, our DHT provides two operations: *get(key)* and *put(key, value)*, which obtains the paired value with a given key and puts one pair onto a target node, respectively. Table 2.3 describes the types of messages exchanged among nodes. Algorithms 2 and 3 depicts the actions for a node to handle these messages. We have omitted the details of failure recovery messages such as *fix_finger* and *stabilize* in Chord, also implemented as messages in our DHT. $node.operation(parameters)$ indicates that the operation with given parameters is executed on the remote node, implemented by sending a corresponding message to this node.

As an example, in order to serve the periodic task to *get K25* on node *N4* in Fig. 2.1, the cyclic executive on *N4* schedules a periodic job *GET* periodically, which sends a *LOOKUP* message to *N20*. *N20* then sends a *LOOKUP* message to *N24*. *N24* then sends a *DESTIN* message to *N30* since it determines that *N30* is the target node for *K25*. *N30* sends *LOOKUP_-DONE* back to the initial node *N4*. Our DHT stores the request detail in a buffer on the initial node and uses a unique identifier (sid) embedded in messages to identify this request. When *N4* receives the *LOOKUP_DONE* message from the target node, it releases an aperiodic job to obtain the request detail from the buffer and continue the request by sending *GET_DIRECT* to the target node *N30* as depicted in the pseudocode. *N30* returns the value to *N4* by sending a *GET_DONE* message back to *N4*. Now, *N4* removes the request detail from the buffer and the request is completed.

2.3 Analysis

As a decentralized distributed storage system, the nodes in our DHT work independently. In this section, we first explain the model that we use to analyze the response times of jobs on a single node. Then, we aggregate them to bound the end-to-end time for all requests.

2.3.1 Job response time on a single node

Let us first state an example of job patterns on a single node to illustrate the problem. Let us assume the hyperperiod of the periodic requests is $30ms$, and the periodic receiving task is executed three times in one hyperperiod at $0ms$ (the beginning of the hyperperiod), $10ms$, and $20ms$, respectively. The timeline of one hyperperiod can be divided into three slots according to the period of the receiving task. In every slot, the cyclic executive utilizes the first 40% of its time to execute periodic jobs, and the remaining time to execute aperiodic jobs as long as

```
Procedure GET(key) and PUT(key, value)
    sid ← unique request identifier
    code ← hashcode(key)
    address ← ip and port of this node
    store the request in the buffer
    execute LOOKUP(address, sid, code)
Procedure LOOKUP(initial-node, sid, hashcode)
    next ← next node for the lookup in the finger table
    if next = target node then
        next.DESTIN(initial-node, sid, hashcode)
    else
        next.LOOKUP(initial-node, sid, hashcode)
    end
Procedure DESTIN(initial-node, sid, hashcode)
    address ← ip and port of this node
    initial-node.LOOKUP_DONE(address, sid)
Procedure LOOKUP_DONE(target-node, sid)
    request ← find buffered request using sid
    if request = get then
        target-node.GET_DIRECT(initial-node, sid, request.key)
    else if request = put then
        target-node.PUT_DIRECT(initial-node, sid, request.key, request.value)
    else if request = fix finger then
        update the finger table
    end
```

Algorithm 2: Pseudocode for message handling jobs (1)

the aperiodic job queue is not empty. Since aperiodic jobs are released when the node receives messages from other nodes, we model the release pattern of aperiodic jobs as a homogeneous Poisson process with $2ms$ as the average inter-arrival time. The execution time is $0.4ms$ for all aperiodic jobs. The problem is to analyze response times of these aperiodic jobs. The response time of an aperiodic job in our model consists of the time waiting for the receiving task to move its message to the aperiodic queue, the time waiting for the executive to execute the job, and the execution time of the job.

An M/D/1 queuing model [26] is suited to analyze the response times of aperiodic jobs if these aperiodic jobs are executed once the executive has finished all periodic jobs in that time slot. However, in our model, the aperiodic jobs that arrive at this node in one time slot could only be scheduled during the next time slot, as these aperiodic jobs are put into the job queue only when the receiving task is executed at the beginning of the next time slot. The generic M/D/1 queuing model cannot capture this property. We need to derive a modified model.

```
Procedure GET_DIRECT(initial-node, sid, key)
    address ← ip and port of this node
    value ← find value for key in local storage
    if $value = nil$ then
        initial-node.GET_FAILED(address, sid)
    else
        initial-node.GET_DONE(address, sid, value)
    end
Procedure PUT_DIRECT(initial-node, sid, key, value)
    address ← ip and port of this node
    store the pair to local storage
    initial-node.PUT_DONE (address, sid)
Procedure PUT_DONE, GET_DONE, GET_FAILED
    execute the associated operation
```

Algorithm 3: Pseudocode for message handling jobs (2)

Formally, Table 2.4 includes the notation we use to describe our model. We also use the same notation without the subscript for the vector of all values. For example, U is $(U_0, U_1, \ldots, U_K)$, C is $(C_0, C_1, \ldots, C_M)$. In the table, C is obtained by measurement from the implementation; H, K, ν, U are known from the schedule table; M is defined in our DHT algorithm. We explain λ in detail in Section 2.3.2.

Given a time interval of length x and arrivals g in that interval, the total execution time of aperiodic jobs $E(x, g)$ can be calculated with Equation 2.1. Without loss of generality, we use notation E and $E(x)$ to represent $E(x, g)$.

$$E(x,g) = \sum_{i=1}^{M} C_i * g_i \; with\ probability \prod_{i=1}^{M} P(g_i, \lambda_i, x) \tag{2.1}$$

We further define A_p as the time available to execute aperiodic jobs in time interval (ν_0, ν_{p+1}).

$$A_p = \sum_{i=0}^{p} (1 - U_i) * (\nu_{i+1} - \nu_i), 0 \leq p \leq K. \tag{2.2}$$

However, the executive executes aperiodic jobs only when periodic jobs are finished in a frame. We define $W(i, E)$ as the response time for an aperiodic job workload E if these jobs are scheduled after the i^{th} receiving job. For $1 \leq i \leq K$, $W(i, E)$ is a function of E calculated from the schedule table in three cases:

(1) If $E \leq A_i - A_{i-1}$, which means the executive can use the aperiodic job quota between $(\nu_i, \nu_{i+1}]$ to finish the workload, we can use the periodic job schedule details in the schedule

Table 2.4: Notation

Notation	**Meaning**
H	hyperperiod
K	number of receiving jobs in one hyperperiod
ν_i	time when the i^{th} receiving job is scheduled [1,2]
U_i	utilization of periodic jobs in time interval (ν_i, ν_{i+1})
M	number of different types of aperiodic jobs
λ_i	average arrival rate of the i^{th} type aperiodic jobs
C_i	worst case execution time of the i^{th} type aperiodic jobs
g_i	number of arrivals of i^{th} aperiodic job
$E(x, g)$	total execution time for aperiodic jobs that arrive in time interval of length x
$W(i, E)$	response time for aperiodic jobs if they are scheduled after i^{th} receiving job
$P(n, \lambda, x)$	probability of n arrivals in time interval of length x, when arrival is a Poisson process with rate λ

[1] ν_i are relative to the beginning of the current hyperperiod.
[2] For convenience, ν_0 is defined as the beginning of a hyperperiod, ν_{K+1} is defined as the beginning of next hyperperiod.

table to calculate $W(i, E)$.

(2) If $\exists p \in [i+1, K]$ so that $E \in (A_{p-1} - A_{i-1}, A_p - A_{i-1}]$, which means the executive utilizes all the aperiodic job quota between (ν_i, ν_p) to execute the workload and finishes the workload at a time between (ν_p, ν_{p+1}), then $W(i, E) = \nu_p - \nu_i + W(p, E - A_{p-1} + A_{i-1})$.

(3) In the last case, the workload is finished in the next hyperperiod. $W(i, E)$ becomes $H - \nu_i + W(0, E - A_K + A_{i-1})$. $W(0, E)$ means to use the aperiodic quota before the first receiving job to execute the workload. If the first receiving job is scheduled at the beginning of the hyperperiod, this value is the same as $W(1, E)$. In addition, we require that any workload finishes before the end of the next hyperperiod. This is accomplished by analyzing the timing of receiving jobs and ensures that the aperiodic job queue is in the same state at the beginning of each hyperperiod, i.e., no workload accumulates from previous hyperperiods except the last hyperperiod.

Let us assume an aperiodic job J of execution time C_m arrives at time t relative to the beginning of the current hyperperiod. Let $p+1$ be the index of the receiving job such that $t \in [\nu_p, \nu_{p+1})$. We also assume that any aperiodic job that arrives in this period is put into the aperiodic job queue by this receiving job. Then, we derive the response time of this job in different cases.

(1) The periodic jobs that are left over from the previous hyperperiod and arrive before ν_p

in the current frame cannot be finished before ν_{p+1}. Equation 2.3 is the formal condition for this case, in which HLW is the leftover workload from the previous hyperperiod.

$$HLW + E(\nu_p) - A_p > 0 \tag{2.3}$$

In this case, the executive has to first finish this leftover workload, then any aperiodic jobs that arrive in the time period $[\nu_p, t)$, which is $E(t-\nu_p)$, before executing job J. As a result, the total response time of job J is the time to wait for the next receiving job at ν_{p+1}, which puts J into the aperiodic queue, and the time to execute aperiodic job workload $LW(\nu_p) + E(t-\nu_p) + C_m$, which is $W(p+1, HLW + E(t) - A_p + C_m)$, after the $(p+1)^{th}$ receiving job. The response time of J is expressed in Equation 2.4.

$$R(C_m, t) = \begin{array}{l}(\nu_{p+1} - t) + \\ W(p+1, HLW + E(t) - A_p + C_m)\end{array} \tag{2.4}$$

(2) In the second case, the periodic jobs that are left over from the previous hyperperiod and arrive before ν_p can be finished before ν_{p+1}; the formal condition and the response time are given by Equations 2.5 and 2.6, respectively.

$$HLW + E(\nu_p) \leq A_p \tag{2.5}$$

$$R(C_m, t) = (\nu_{p+1} - t) + W(p+1, E(t-\nu_p) + C_m) \tag{2.6}$$

The hyperperiod leftover workload HLW is modeled as follows. Consider three continuous hyperperiod, H_1, H_2, H_3. The leftover workload from H_2 consists of two parts. The first part, $E(H - \nu_K)$, is the aperiodic jobs that arrive after the last receiving job in H_2, as these jobs can only be scheduled by the first receiving job in H_3. The second part, $E(H + \nu_K) - 2A_{K+1}$, is the jobs that arrive in H_1 and before the last receiving job in H_2 which have not been scheduled in all the aperiodic allotment in H_1 and H_2. We construct a distribution for HLW in this way. In addition, more previous hyperperiods considered results in more stable model for HLW. However, our evaluation shows that two previous hyperperiods is sufficient for the request patterns.

Now we have the stochastic model $R(C_m, t)$ for the response time of an aperiodic job of execution time C_m that arrives at a specific time t. By sampling t in one hyperperiod, we have the stochastic model for the response time of aperiodic jobs that arrive at any time.

2.3.2 End-to-end response time analysis

We aggregate the single node response times and network delays to transmit messages for the end-to-end response time of requests. The response time of any request consists of four parts: (1) the response time of the initial periodic job; this value is known by the schedule table; (2) the total response time of jobs to handle subsequent lookup messages on at most $logN$ nodes with high probability [40], where N is the number of nodes; (3) the response time of aperiodic jobs to handle LOOKUP_DONE, and the final pair of messages, e.g., PUT_DIRECT and PUT_DONE; (4) the total network delays to transmit these messages. We use a value δ based on measurement for the network delay, where $P(network\ delay \leq \delta) \geq T$, for a given threshold T.

To use the single node response time model, we need to know the values of the model parameters of Table 2.4. With the above details on requests, we can obtain H, K, v, U from the schedule table. λ is defined as follows: let T be the period of the initial request on each node, then $\frac{N}{T}$ new requests arrive at our DHT in one time unit, and each request at most $logN$ subsequent lookup messages. Let us assume that hash codes of nodes and keys are randomly located on the Chord ring, which is of high probability with the SHA-1 hashing algorithm. Then each node receives $\frac{logN}{T}$ lookup messages in one time unit. The arrival rate of LOOKUP and DESTIN messages is $\frac{logN}{T}$. In addition, each request eventually generates one LOOKUP_DONE message and one final pair of messages, so for these messages the arrive rate is $\frac{1}{T}$.

2.3.3 Quality of service

We define quality of service (QoS) as the probability that our real-time DHT can guarantee requests to be finished before their deadlines. Formally, given the relative deadline D of a request, we use the stochastic model $R(C_m, t)$ for single node response times and the aggregation model for end-to-end response times to derive the probability that the end-to-end response time of the request is equal or less than D. In this section, we apply the formula in our model step by step to explain how to derive this probability in practice.

The probability density function $\rho(d, C_m)$ is defined as the probability that the single node response time of a job with execution time C_m is d. We first derive the conditional density function $\rho(d, C_m|t)$, which is the probability under the condition that the job arrives at time t, i.e., the probability that $R(C_m, t) = d$. Then, we apply the law of total probability to derive $\rho(d, C_m)$. The conditional density function $\rho(d, C_m|t)$ is represented as a table of pairs $\pi(\rho, d)$, where ρ is the probability that an aperiodic job finishes with response time d. We apply the algorithms described in Section 2.3.1 to build this table as follows. Let $p + 1$ be the index of the receiving job such that $t \in [\nu_p, \nu_{p+1})$.

(1) We need to know when to apply Equations 2.4 and 2.6. This is determined by the

probability χ that condition $HLW + E(\nu_p, g) \leq A_p$ holds. To calculate χ, we enumerate job arrival vectors g that have significant probabilities in time $(0, \nu_p)$ according to the Poisson distribution, and use Equation 2.1 to calculate the workload $E(\nu_p, g)$ for each g. The term significant probability means any probability that is larger than a given threshold, e.g., 0.0001%. Since the values of HLW and $E(\nu_p, g)$ are independent, the probability of a specific pair of HLW and arrival vector g is given by simply their multiplications. As a result, we build a condition table $CT(g, \rho)$, in which each row represents a pair of vector g, which consists of the numbers of aperiodic job arrivals in time interval $(0, \nu_p)$ under the condition $HLW + E(\nu_p, g) \leq A_p$, and the corresponding probability ρ for that arrival vector. Then, $\chi = \sum \rho$ is the total probability for that condition.

(2) Build probability density table $\pi_2(\rho, d)$ for response times of aperiodic jobs under condition $HLW + E(\nu_p) \leq A_p$. In this case, we enumerate job arrival vectors g that have significant probabilities in time $(0, t - \nu_p)$ according to the Poisson distribution. and use Equation 2.1 to calculate its workload, use Equation 2.6 to calculate its response time for each g. Each job arrival vector generates one row in density table π_2.

(3) Build probability density table $\pi_3(\rho, d)$ for response times of aperiodic jobs under condition $HLW + E(\nu_p) > A_p$. We enumerate job arrival vectors g that have significant probabilities in time $(0, t)$ according to the Poisson distribution, and use Equation 2.4 to calculate response times. Since time interval $(0, t)$ includes $(0, \nu_p)$, arrival vectors that are in condition table CT must be excluded from π_3, because rows in CT only represent arrivals that results in $HLW + E(\nu_p) \leq A_p$. We normalize the probabilities of the remaining rows. By normalizing, we mean multiplying each probability by a same constant factor, so that the sum of the probabilities is 1.

(4) We merge the rows in these two density tables to build the final table for the conditional density function. Before merging, we need to multiply every probability in table π_2 by the weight χ, which indicates the probability that rows in table π_2 are valid. For the same reason, every probability in table π_3 is multiplied by $(1 - \chi)$.

Now, we apply the law of total probability to derive $\rho(d, C_m)$ from the conditional density functions by sampling t in $[0, H)$. The conditional density tables for all samples are merged into a final density table $\prod(\rho, d)$. The samples are uniformly distributed in $[0, H)$ so that conditional density tables have the same weight during the merge process. After normalization, table $\prod$ is the single node response time density function $\rho(d, C_m)$.

We apply the aggregation rule described in Section 2.3.2 to derive the end-to-end response time density function on the basis of $\rho(d, C_m)$. According to the rule, end-to-end response time includes the response time of the initial periodic job, network delays, and the total response time of $(logN + 3)$ aperiodic jobs of different types. In order to represent the density function of the total response time for these aperiodic jobs, we define the operation SUM on two density

```
1  **Procedure** UNIQUE ($\pi$)
2      $\pi_{des} \leftarrow$ empty table
3      **for** each row $(\rho, d)$ in $\pi$ **do**
4          **if** row $(\rho_{old}, d)$ exists in $\pi_{des}$ **then**
5              $\rho_{old} \leftarrow \rho_{old} + \rho$
6          **else**
7              add row $(\rho, d)$ to $\pi_{des}$
8          **end**
9      **end**
10     **return** $\pi_{des}$
11 **Procedure** SUM ($\pi_1$, $\pi_2$)
12     $\pi_3 \leftarrow$ empty table
13     **for** each row $(\rho_1, d_1)$ in $\pi_1$ **do**
14         **for** each row $(\rho_2, d_2)$ in $\pi_2$ **do**
15             add row $(\rho_1 * \rho_2, d_1 + d_2)$ to $\pi_3$
16         **end**
17     **end**
18     **return** UNIQUE($\pi_3$)
```

Algorithm 4: Density tables operations

tables π_1 and π_2 as in Algorithm 4. The resulting density table has one row $(\rho_1 * \rho_2, d_1 + d_2)$ for each pair of rows (ρ_1, d_1) and (ρ_2, d_2) from table π_1 and π_2, respectively. That is, each row in the result represents one sum of two response times from the two tables and the probability of the aggregated response times. The density function of the total response time for $(logN + 3)$ aperiodic jobs is calculated as Equation 2.7 (SUM on all π_i), where π_i is the density function for the i^{th} job.

$$\rho(d) = \sum_{i=1}^{logN+3} \pi_i \tag{2.7}$$

The maximum number of rows in density table $\rho(d)$ is $2(logN+3)H\omega$, where $2H$ is the maximum response time of single node jobs, and ω is the sample rate for arrival time t that we use to calculate each density table.

Let us return to the QoS metric, i.e., the probability that a request can be served within a given deadline D. We first reduce D by fixed value Δ, which includes the response time for the initial periodic job and network delays $(logN + 3)\delta$. Then, we aggregate rows in the density function $\rho(d)$ to calculate this probability $P(D - \Delta)$.

$$P(D - \Delta) = \sum_{(\rho_i, d_i) \in \rho(d), d_i \leq D - \Delta} \rho_i \tag{2.8}$$

2.4 Evaluation

In this section, we evaluate our real-time DHT on a local cluster with 2000 cores over 120 nodes. Each node features a 2-way SMP with AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node). Each node has 32GB DRAM and Gigabit Ethernet (utilized in this study) as well as Infiniband Interconnect (not used here). We apply workloads of different intensity according to the needs of the power grid control system on different numbers of nodes (16 nodes are utilized in our experiments), which act like PDCs. The nodes are not synchronized to each other relative to their start of hyperperiods as such synchronization would be hard to maintain in a distributed system. We design experiments for different intensity of workloads and then collect single-node and end-to-end response times of requests in each experiment. The intensity of a workload is quantified by the system utilization under that workload. The utilization is determined by the number of periodic lookup requests and other periodic power control related computations. The lookup keys have less effect on the workload and statistic results as long as they are evenly stored on the nodes, which has high probability in Chord. In addition, the utilization of aperiodic jobs is determined by the number of nodes in the system. The number of messages passing between nodes increases logarithmically with the number of nodes, which results in an increase in the total number of aperiodic jobs. We compare the experimental results with the results given by our stochastic model for each workload.

In the third part of this section, we give experimental results of our extended real-time DHT, in which the cyclic executive schedules aperiodic jobs based on the priorities of requests. The result shows that most of the requests that have tight deadlines can be finished at the second trial under the condition that the executive did not finish the request at the first time.

2.4.1 Low workload

In this experiment, we implement workloads of low utilizations for the cyclic executive to schedule. In detail, one hyperperiod (30ms) contains three frames of 10ms, each with a periodic followed by an aperiodic set of jobs. The periodic jobs include the receiving job for each frame and the following put/get jobs: In frame 1, each node schedules a *put* request followed by a *get* request; in frame 2 and 3, each node schedules a *put* request, respectively. In each frame, the cyclic executive schedules a *compute* job once the periodic jobs in that frame have been executed. This *compute* job is to simulate periodic computations on PDCs for power state estimation, which is implemented as a tight loop of computation in our experiments. As a result, the utilizations of periodic jobs in the three frames are all 40%. Any put/get requests forwarded to other nodes in the DHT result in aperiodic (remote) jobs. The execution time of aperiodic jobs is $0.4ms$. The system utilization is 66.7% (40% for periodic jobs and 26.7% for aperiodic jobs) when the workload is run with 4 DHT nodes.

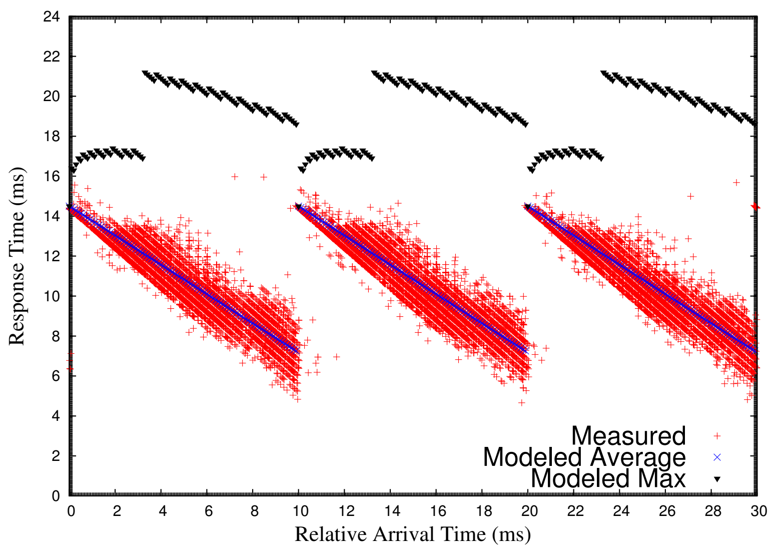

Figure 2.2: Low-workload Single-node Response Times (4 nodes)

Results are plotted over 3,000 hyperperiods. Fig. 2.2 depicts the single-node response times comparison. The red dots in the figure are measured response times of jobs. Since our stochastic model derives a distribution of response times for jobs arriving at every time instance relative to the hyperperiod, we depict the mathematical expectation and the maximum value for each time instance, which are depicted as blue and black lines in the figure, respectively. The figure shows that messages sent at the beginning of each frame experience longer delays before they are handled by corresponding aperiodic jobs with proportionally higher response times. The reason for this is that these messages spend more time waiting for the execution of the next receiving job so that their aperiodic jobs can be scheduled. The modeled average times (blue) follow a proportionally decaying curve delimited by the respective periodic workload of a frame (4ms) plus the frame size (10ms) as the upper bound (left side) and just the periodic workload as the lower bound (right side). The figure shows that the measured times (red) closely match this curve as the average response time for most of the arrival times.

Maximum modeled times have a more complex pattern. In the first frame, their response times are bounded by 18ms for the first 4ms followed by a nearly proportionally decaying curve (22ms-18ms response time) over the course of the next 6ms. The jump at 4ms is due to servicing requests that arrived in the first 4ms, including those from the previous hyperperiod. Similar jumps between frames exist for the same reason, where their magnitude is given by the density of remaining aperiodic jobs and the length of the periodic workload, which also accounts for the near-proportional decay. This results in aperiodic jobs waiting two frames before they execute when issued during the second part of each frame.

Fig. 2.3 depicts the cumulative distributions of single-node response times for aperiodic jobs. The figure shows that 99.3% of the aperiodic jobs finish within the next frame after they are released under this workload, i.e., their response times are bounded by $14.4ms$. Our model predicts that 97.8% of aperiodic jobs are finished within the $14.4ms$ deadline, which is a good match. In addition, for most of the response times in the figure, our model predicts that a smaller fraction of aperiodic jobs is finished within the response times than the fraction in the experimental data, as the blue curve for modeled response times is below the red curve, i.e., the former effectively provides a lower bound for the latter. This indicates that our model is conservative for low workloads.

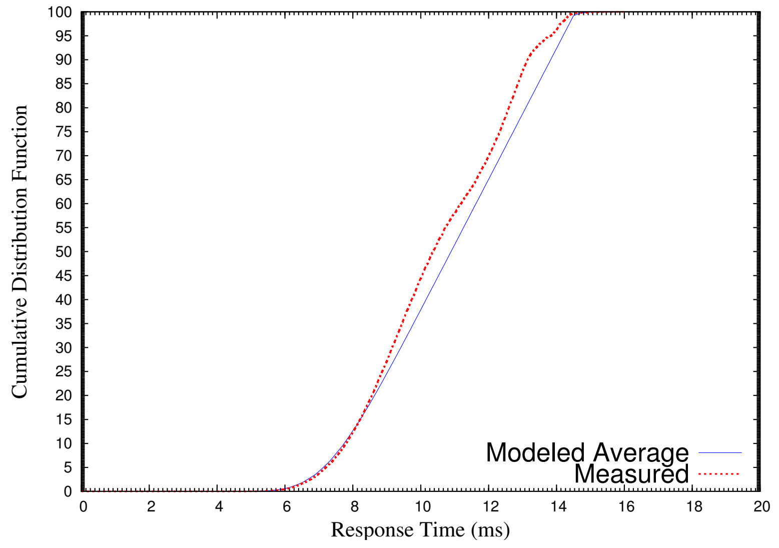

Figure 2.3: Low-workload Single-node Response Times Distribution (4 nodes)

Fig. 2.4 depicts the cumulative distributions of end-to-end response times for requests (solid lines for 4 nodes), i.e., the time between a put/get request and its final response after propagating over multiple hops (nodes) within the DHT. This figure shows that response times have three clusters centered around $35ms$, $45ms$, and $55ms$. Every cluster has a base response time (about $35ms$) due to the LOOK_DONE and the final pair of messages as discussed in Section 2.3.2. In addition, different clusters indicate that requests contact different numbers of nodes to eventually serve the requests. One more intermediate node contacted increases the response time by $10ms$ on average, which is the average single-node response time as depicted in Fig. 2.3. Approximately 10% of the requests are served without an intermediate node. This happens when the initial node of a request is also the target node. In this case, our current DHT

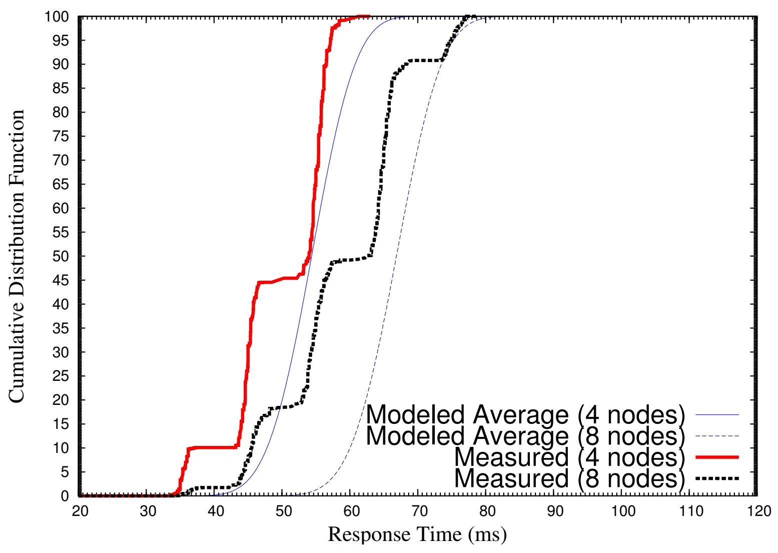

Figure 2.4: Low-workload End-to-end Response Times Distribution

implementation sends a LOOKUP_DONE message to itself directly. Our QoS model considers the worst case where $log(N)$ nodes are required, which provides an upper bound on response times given a specific probability. In addition, this figure shows that our QoS model is conservative. For example, 99.9% of the requests are served within $62ms$ in the experiment, while our model indicates 93.9% coverage for that response time.

In addition, we evaluate our model with the same request pattern with 8 DHT nodes. The system utilization becomes 72% as the utilization for aperiodic jobs increases because of the additional nodes in the DHT. Fig. 2.4 (dotted lines) depicts the cumulative distributions of end-to-end response times for requests. Four clusters of response times are shown in this figure (the base value is considered larger between $40ms$ and $45ms$ as the utilization has increased). One more intermediate node is contacted for a fraction of requests compared to the previous experiment with 4 DHT nodes.

2.4.2 High workload

In this experiment, we increase the utilization of the workloads for the cyclic executive. In detail, one hyperperiod (40ms) contains four frames of 10ms, each with a periodic followed by an aperiodic set of jobs. The periodic jobs include the receiving job and following two put/get jobs for each frame. Any put/get requests forwarded to other nodes in the DHT result in aperiodic jobs. The execution time of aperiodic jobs are $0.4ms$. The system utilization is 88.0% when the workload is run with 8 DHT nodes, which includes 40% for periodic jobs and 44.0% for aperiodic jobs.

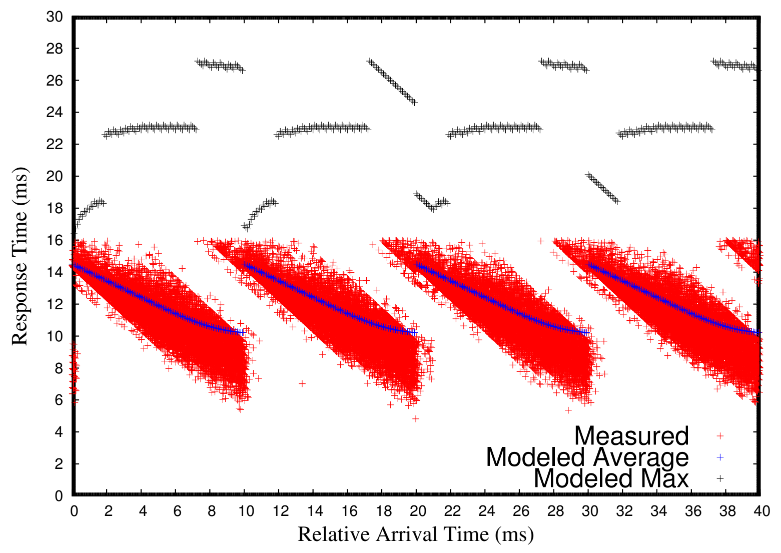

Figure 2.5: High-workload Single-node Response Times (8 nodes)

Fig. 2.5 depicts the single-node response times comparison. The red dots in the figure depict measured response times of jobs. The blue and black lines are the mathematical expectation and maximum response times for each sample instance given by our model, respectively. Compared with Fig. 2.2, a part of the red area at the end of each frame, e.g., between $8ms$ to $10ms$ for the first frame, moves up indicating response times larger than $14.4ms$. This is due to a larger fraction of aperiodic jobs during these time interval being scheduled after the next frame (i.e., in the third frame), due to the increase of system utilization. Our model also shows this trend as the tails of blue lines curve up slightly. Figures 2.6 and 2.7 depict the cumulative distributions of single-node and end-to-end response times under this workload, respectively. Compared with Figures 2.3 and 2.4, the curves for higher utilization move to the right. This suggests larger response times. Our model provides upper bound on response times of 99.9% of all requests. Fig. 2.7 (dotted lines) also depicts the cumulative distribution of end-to-end response times of requests running on 16 DHT nodes. Our model also gives reliable results for this case, as the curves for the experimental data are slightly above the curves of our model.

These experiments demonstrate that a stochastic model can result in highly reliable response times bounds, where the probability of timely completion can be treated as a quality of service (QoS) property under our model.

2.4.3 Prioritized queue extension

In this experiment, we extend our real-time DHT to use prioritized queues to schedule aperiodic jobs. Our current stochastic model does not take prioritized queues into consideration, so we

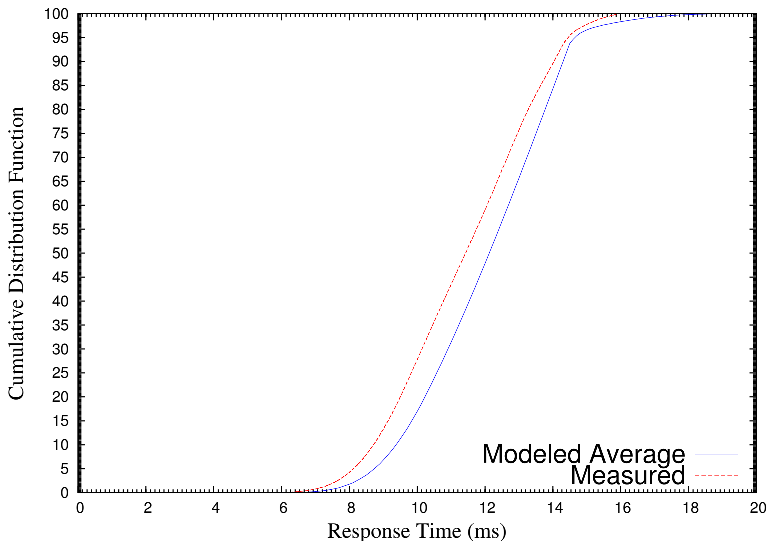

Figure 2.6: High-workload Single-node Response Times Distribution (8 nodes)

compare the results with that of the real-time DHT without prioritized queue for the low workload executed on 4 DHT nodes.

In this extension, the system assigns the lowest priority to jobs of requests that are released for the first time and sets a timeout at their deadline. If the system fails to complete a request before its deadline, it increases the priority of its next job release until that request is served before its deadline. To implement this, all the messages for a request inherit the maximum remaining response time, which is initially the relative deadline, to indicate the time left for that request to complete. This time is reduced by the single node response time (when the aperiodic job for that request is finished) plus the network delay δ if request is forwarded to the next node. When a node receives a message that has no remaining response time, the node simply drops the message instead of putting it onto aperiodic job queues. Cyclic executives always schedule aperiodic jobs with higher priority first (FCFS for the jobs of same priority).

In the experiment, we set the relative deadline of requests to $55ms$, which is the duration within which half of the requests with two intermediate nodes can be served.

The purple curve in Fig. 2.8 depicts the cumulative distribution of end-to-end response times of requests served by the prioritized implementation. Requests that require three intermediate nodes have higher probability to miss the $55ms$ relative deadline at first. By increasing the priorities of requests that failed the first time, we observe that prioritized requests never fail again. As a result of prioritization, the response times of requests with lower priorities increases. For example, compared to the red curve, which is for the implementation without prioritized queues (FIFO queues), a larger proportion of requests have response times between $40ms$ and

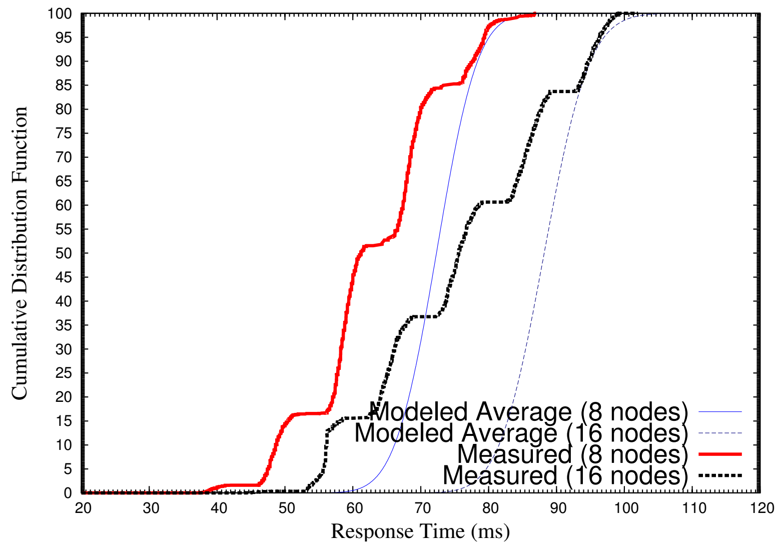

Figure 2.7: High-workload End-to-end Response Times Distribution

$50ms$. In addition, requests that have response times in the same cluster (e.g., centered around $45ms$) could have larger numbers of intermediate nodes (hops) in the prioritized implementation since prioritized requests have smaller response times per sub-request job.

2.5 Related work

Distributed hash tables are well known for their performance and scalability for looking up the node that stores a particular piece of data in a distributed environment. Many existing DHTs use the number of nodes involved in one lookup request as a metric to measure performance. For example, Chord and Pastry require $O(logN)$ node traversals using $O(logN)$ routing tables to locate the target node for the given data in a N-nodes network [40, 56], while D1HT [39] and ZHT [32] requires a single node traversal of the expense of $O(N)$ routing tables. However, this level of performance analysis is not suited for real-time applications, as these applications require detailed timing information to guarantee time bounds on lookup requests. In our research, we focus on analyzing response times of requests by modeling the pattern of job executions on the nodes.

We build our real-time DHT on the basis of Chord, so each node needs to maintain a small fraction of information of other nodes in its soft routing table. Compared with D1HT and ZHT, in which one node maintains the information of all other nodes, Chord requires less communication overhead to maintain its routing table when nodes or links have failed in the DHT. Thus, we believe a Chord-like DHT is more suitable for wide-area power grid real-time

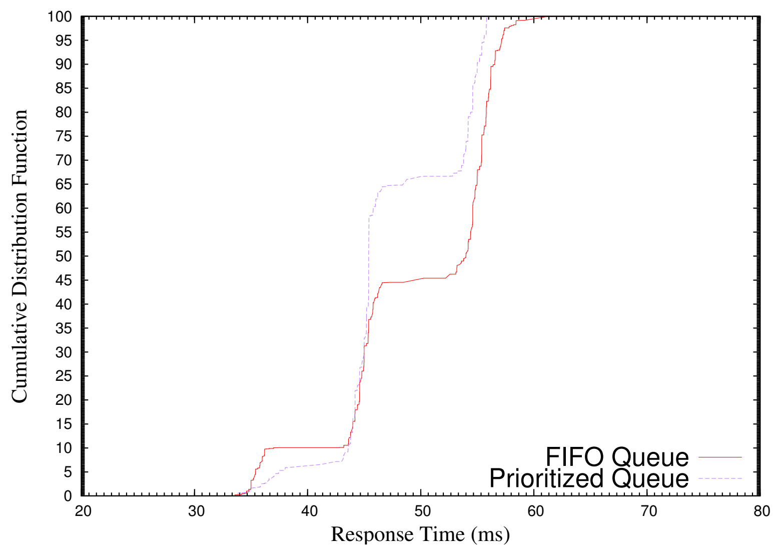

Figure 2.8: End-to-end Response Times Distribution Comparison

state monitoring as it is more scalable in the numbers of PDCs and PMUs.

In our timing analysis model, we assume that node-to-node network latency is bounded. Much of prior research changes the OSI model to smooth packet streams so as to guarantee time bounds on communication between nodes in switched Ethernet [30, 14]. Software-defined networks (SDN), which allow administrators to control traffic flows, are also suited to control the network delay in a private network [36]. Distributed power grid control nodes are spread in a wide area, but use a proprietary Internet backbone to communicate with each other. Thus, it is feasible to employ SDN technologies to guarantee network delay bounds in a power grid environment [7].

2.6 Conclusion

We have designed and implemented a real-time distributed hash table (DHT) on the basis of Chord to support a deadline-driven lookup service at upper layer control systems of the North American power grid. Our real-time DHT employs a cyclic executive to schedule periodic and aperiodic lookup jobs. Furthermore, we formalize the pattern of lookup workload on our DHT according to the needs of power grid monitoring and control systems, and use queuing theory to analyze the stochastic bounds of response times for requests under that workload. We also derive the QoS model that measures the probability that the deadlines of requests can be met by our real-time DHT. Our problem was motivated by the power grid system but the cyclic executive and the approach of timing analysis is generic to distributed storage systems in which the requests follow our pattern. Our evaluation shows that our model is suited to provide an upper bound on response times and that a prioritized extension can increase the probability of meeting deadlines for subsequent requests.

Our current analysis does not take node failures into consideration. With data replication (Chord stores data not only at the target node but in the nodes on the successor list [40]), requested data can still be obtained with high probability if node failures happen. Failures can be detected when jobs on one node fail to send messages to other nodes during the execution. In this case, the job searches the finger table again to determine the next node to send a data request message to, which increases the execution times of that job. In future work, we will generalize the modeling of executions times so that node failures can be tolerated.

Chapter 3

Hybrid EDF Packet Scheduling for Real-Time Distributed Systems

3.1 Introduction

When computational resource elements in a cyber-physical system are involved in the handling of events, understanding and controlling the scheduling algorithms on these resource elements become an essential part in order to make them collaborate to meet the deadlines of events. Our previous work has deployed a real-time distributed hash table in which each node employs a cyclic executive to schedule jobs that process data request events with probabilistic deadlines, which are resolved from a hash table [47]. However, cyclic executives are static schedulers with limited support for real-time systems where jobs have to be prioritized according to their urgency. Compared to static schedulers, EDF always schedules the job with earliest deadline first and has a utilization bound of 100% on a preemptive uniprocessor system [34, 33]. In theory, two major assumptions have to be fulfilled in real-time systems in order to achieve this utilization bound. First, the scheduler has to know the deadlines of all jobs in the system, so it can always schedule the job with the shortest deadline. Second, the cost of a context switch between different jobs is ignorable. To satisfy the second assumption, not only the cost of a single context switch, but also the number of context switches must be ignorable. This constrains the number of job releases in a period of time. Employing EDF schedulers in such a distributed system raises significant challenges considering the difficulty of meeting these assumptions in a cyber-physical system in which computational resource elements (i.e., nodes) communicate with each other by message passing.

First, to handle an event in a cyber-physical system, messages of the event are passed among nodes to trigger jobs on each node so that the event can be eventually processed. When messages arrive at a particular node, they are queued in the operating system network buffer until the job scheduler explicitly receives these messages and releases corresponding jobs to process them. The delays between the arrival and reception of messages provide a limitation on the capability of the job scheduler in terms of awareness of the deadlines of all current jobs on the node. One possible solution to address this problem is to let the operating system interrupt the job scheduler whenever a new message arrives at the node. In this way, the scheduler can always accept new messages at their arrival time so that it knows the deadlines of all current jobs. However, this may result in an unpredictable cost of switching between the job execution context and the interruption handling context that receives messages.

To address this problem, we combine a periodic message receiving task with the EDF scheduler. This receiving task accepts messages from the system network buffer and releases corresponding jobs into job waiting queues of the EDF scheduler. Considering the fact that message jobs can only be released when the receiving task is executing, this design makes the scheduler partial-EDF because of the aforementioned delays. However, it increases the predictability of the system in terms of temporal properties including period, worst case execution time, and relative deadline of the receiving task, which also makes context switch cost more predictable. In addition, our partial-EDF adopts a periodic message sending task to regulate the messages sent by events so that the inter-transmission time of messages has a lower bound. We have theoretically studied the impact of the temporal properties of these transmission tasks on the schedulability of partial-EDF.

The second challenge is relevant to the deadlines carried in event messages. Since clocks of nodes in cyber-physical systems are not synchronized globally considering the cost of global clock synchronization [28], deadlines carried in messages sent by different nodes lose their significance when these messages arrive at a node. To address this problem, our scheduler maintains a time-difference table on each node consisting of the clock difference between the senders and this receiving node. Hence, the deadlines carried in messages can be adjusted to the local time based on the information in this time-difference table. Thus, the EDF order of jobs can be maintained. These time-difference tables are built based on the Network Time Protocol (NTP) [37], and they are periodically updated so that clock drifts over time can be tolerated.

The third challenge is to optimize the underlying network that the cyber-physical system utilizes to transmit event messages between nodes to provide a bounded network delay. We assume that even in a real-time distributed system where all nodes release distributed jobs periodically, the number of corresponding messages received by the receiver nodes cannot be guaranteed to be periodic due to the inconsistent delay of the underlying network. Since the jobs that handle these messages could send messages to further nodes (when distributed jobs require more than two nodes to handle them), the variance in the numbers of messages can potentially increase the burstiness of the network traffic, which decreases the predictability of network delay. Our evaluation in Section 3.4.3 proves this hypothesis. We address this challenge in two ways. First, we propose an EDF-based packet scheduler that works on the system network layer on end nodes to transmit packets in EDF order. Second, since network congestion increases the variance of the network delay especially in communication-intensive cyber-physical systems (considering the fact that multiple nodes may drop messages onto the network simultaneously and that bursty messages may increase the packet queuing delay on intermediate network devices, which may require retransmission of message by end nodes), we employ a bandwidth sharing policy on end nodes to control the data burstiness in order to bound the worst-case network delay of message transmission. This is essentially a network resource sharing problem. In terms of resource sharing among parallel computation units, past research has focused on shaping the resource access by all units into well defined patterns. These units collaborate to reduce the variance of resource access costs. E.g., MemGuard [63] shapes memory accesses by each core in modern multi-core platforms in order to share memory bandwidth in a controlled way so that the memory access latency is bounded. D-NoC [13] shapes the data transferred by each node on the chip with a (σ, ρ) regulator [11] to provide a guaranteed latency of data transferred to the processor. We have implemented our bandwidth sharing policy based on a traffic control mechanism integrated into Linux to regulate the data dropped onto the network on each distributed node.

To demonstrate the feasibility and capability of our partial-EDF job scheduler, the EDF-based packet scheduler and the bandwidth sharing policy, , we integrated them into our previous real-time distributed hash table (DHT) that only provides probabilistic deadlines [47]. The new real-time distributed system still utilizes a DHT [40] to manage participating storage nodes and to forward messages between these nodes so that data requests are met. However, each participating node employs our partial-EDF job scheduler (instead of the cyclic executive in our previous paper) to schedule data request jobs at the application level. Furthermore, each node employs our EDF-based packet scheduler to send packets in EDF order at the system network level. In addition, all participating nodes follow our bandwidth sharing policy to increase the predictability of network delay. We evaluated our system on a cluster of nodes in a switched network environment.

In summary, the contributions of this work are: (1) We combine periodic message transmission tasks with an EDF scheduler to increase the predictability of real-time distributed systems; we study the impact of period, relative deadline, and WCET of transmission tasks on the EDF job scheduler. (2) We propose an EDF-based packet scheduler running at the operating system level to make the network layer aware of message deadlines. (3) We utilize a Linux traffic control facility to implement our bandwidth sharing policy to decrease the variance of network delays. (4) We implement our comprehensive EDF scheduler and bandwidth sharing policy in a real-time distributed storage system to demonstrate the feasibility and capability of our work.

The rest of the paper is organized as follows. Section 3.2 presents the design of our partial-

EDF job scheduler, packet scheduler, and traffic control policy. Section 3.3 details our real-time distributed storage system implementation. Section 3.4 discusses the evaluation results. Section 3.5 contrasts our work with related work. Section 3.6 presents the conclusion and on-going research.

3.2 Design

This section first presents the design of our partial earliest deadline first scheduler (partial-EDF) that employs periodic tasks to transmit messages and release jobs, and schedules these jobs in EDF order. It then presents the time-difference table maintained on each node so that the absolute deadlines carried in messages can be adjusted to local clock time. Then, this section presents the design of our EDF packet scheduler and bandwidth sharing policy at the network transport layer.

3.2.1 Partial-EDF Job Scheduler

In distributed real-time systems, messages that contain the deadlines of the event are passed among nodes so that the event can be eventually processed. When a message arrives at the operating system network buffer, the scheduler has to acknowledge the arrival of the message so that it can release a corresponding job, which inherits the absolute deadline of the message and can be executed later to process the message. However, when the sender transmits messages periodically, the scheduler on the receiving side cannot guarantee that the release of jobs to process these messages be periodic considering the variance of network delays between the sender and receiver, which causes deviations in message arrival times.

One possible solution is to let the operating system notify the scheduler when a message arrives. For example, a scheduler implemented on a Linux system may utilize a thread to accept messages. This thread is waiting on the network socket until the kernel notifies the thread of message arrivals. The scheduler, once interrupted, releases sporadic jobs to process messages. In this way, the scheduler can process the deadlines of messages immediately when they arrive at the node. Let us call this scheduler *global-EDF* if it executes the job in an EDF order, considering the fact that this scheduler has the knowledge of all deadlines of the corresponding jobs in the system so that it can always execute jobs with the shortest deadlines first. The drawback of the solution is that it is hard to provide a reliable service. Since the operating system may interrupt the scheduler as well as the job executed at a sporadic time, context switches between the scheduler and the interrupt handler may reduce the predictability of the system.

To eradicate the randomness of context switches, we combine the EDF scheduler with a static schedule table, which includes temporal information of a periodic message sending task and a periodic message receiving task. The sending task is to guarantee inter-transmission time of messages has a lower bound. We will describe its details when we present the schedulability test of this EDF scheduler. The receiving task, once executed, accepts messages from the operating system network buffer, processes deadlines contained in these messages, and releases corresponding jobs into the EDF job queue. The receiving task introduces delays between message arrivals and receptions, since messages that have arrived at a node have to wait in the network buffer until the next execution of the transmission task. As a result, the scheduler only has the deadline information of the jobs in the job queue (but no deadline information of the messages waiting in the network buffer). Thus, we name this scheduler the *partial-EDF* job scheduler.

Fig. 3.1 illustrates the effects of the partial-EDF job scheduler on a node that receives and processes messages of 3 distributed tasks. The green blocks on the first line (τ') represent the execution of the periodic receiving jobs while blocks on the other lines (τ_1, τ_2, τ_3) represent the execution of the corresponding jobs for these tasks, respectively. Four messages arrive at time 2, 3, 6, and 8. Since the corresponding jobs of the messages can only be released the next time a transmission job is executing, jobs $\tau_{1,1}, \tau_{1,2}, \tau_{2,1}, \tau_{3,1}$ are released at time 7, 7, 7, and 13, as shown in the figure. The numbers in the blocks indicate the absolute deadlines of the jobs inherited from messages. As the figure shows, both messages, $\tau_{2,1}$ and $\tau_{3,1}$, have arrived at this node at time 8. However, $\tau_{2,1}$ is executed first even though $\tau_{3,1}$ has a tighter deadline since the message of $\tau_{3,1}$ is still waiting in the network buffer and the partial-EDF scheduler has no knowledge of its arrival at time 8.

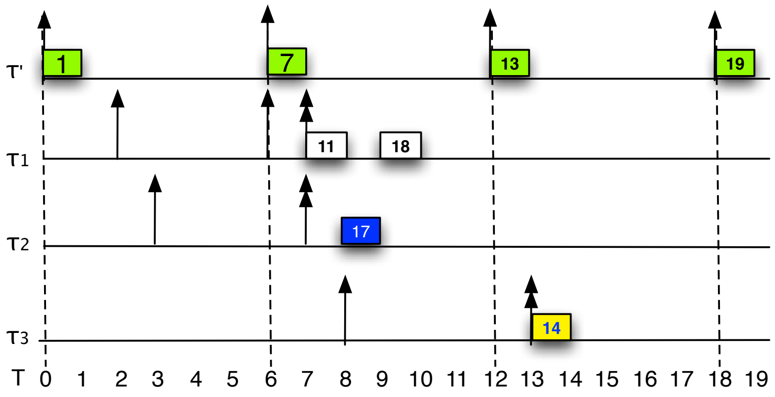

Figure 3.1: Partial-EDF Job Scheduler Example (1)

The system density-based schedulability test [33] for global-EDF scheduling is not sufficient for our partial-EDF. As an example, assume the absolute deadline of $\tau_{3,1}$ was 10 in Fig. 3.1. The global-EDF scheduler can execute all jobs before their deadlines ($\tau_{3,1}$ is scheduled at time 8) as the system density is no more than 100%. However, under partial-EDF, $\tau_{3,1}$ cannot execute before time 10 because of the delay between the message arrival at time 8 and its reception within τ' at time 12.

To derive the schedulability test for our partial-EDF scheduler, we consider a set τ of N tasks that are scheduled in a distributed system of M nodes. A local task $\tau_l = (T_l, C_l, D_l)$ is characterized by its period T_l, worst case execution time (WCET) C_l, and relative deadline D_l. A distributed task $\tau_i = (T_i, C_i, D_i) \xrightarrow{s_i, r_i} (T'_i, C'_i)$ is characterized by a node s_i that releases jobs of the task with period T_i, relative deadline D_i, and WCET C_i of the jobs. The message sending task on node s_i regulates the message traffic of task τ_i so that the minimum inter-transmission time is T'_i, although the EDF scheduler could reduce the time between the executions of consecutive jobs of task τ_i. However, we require that $T_i \geq T'_i$. The receiving task on the receiver r_i releases jobs with WCET C'_i to process these messages. In addition, we use $\tau_i^s = (T_i^s, C_i^s, D_i^s)$ to denote the sending task on the i^{th} node, and we use $\tau_i^r = (T_i^r, C_i^r, D_i^r)$ to denote the receiving task.

Fig. 3.2 depicts an example that has one distributed task τ_1 and two nodes, (i.e., $N = 1$, $M = 2$). The number in a block represents the absolute deadline of that job. The red blocks on the first line represent the execution of the jobs of τ_1 on the first node in EDF order. $\tau_1 = (3, 1, 21) \xrightarrow{1,2} (3, 1)$ indicates that the message sending task on the first node $\tau_1^s(3, 1, 1)$ guarantees that the inter-transmission time of messages of τ_1 is at least 3. The second receiving job (shown as a block with absolute deadline 11) on the second node $\tau_2^r(5, 1, 1)$ accepts the first three messages of τ_1 and releases three jobs to process them, respectively.

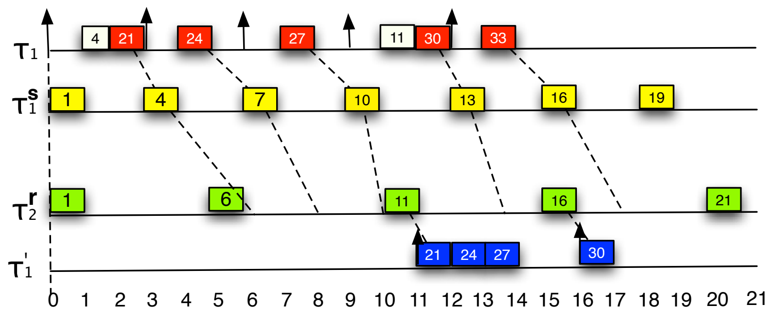

Figure 3.2: Partial-EDF Job Scheduler Example (2)

A sufficient schedulability test for our partial-EDF is to prove that a task set always has a system utilization of no more than 100% during any time interval. Let us consider the worst case. Task τ_i generates a flow of periodic messages (period T'_i) from s_i to r_i. Equation 3.1 expresses the arrival time of the j^{th} message of task τ_i on the receiver r_i, where $\delta_{s_i,r_i,j}$ is the network delay between the sender and receiver when this message is transmitted and O_{s_i,r_i} is the clock time difference between the sender and receiver. Equations 3.2 and 3.3 express the constraints of the number of messages k_i of task τ_i that can be queued in the network buffer together with the j^{th} message of the same task before their reception at the receiver r_i (Equation 3.3 is derived from the left-most terms of Equation 3.2). In this worst case, messages j, $j+1$, ..., $j+k_i-1$ all arrive at the receiver during $xT^r_{r_i}$, the start time of the x^{th} receiving job on r_i, and $(x+1)T^r_{r_i}$, the start time of the $(x+1)^{th}$ receiving job (assuming messages of the same event are transmitted by the underlying network in FIFO order).

$$A(i,j) = jT'_i + \delta_{s_i,r_i,j} + O_{s_i,r_i} \tag{3.1}$$

$$xT^r_{r_i} < A(i,j) \leq A(i,j+k_i-1) \leq (x+1)T^r_{r_i} \tag{3.2}$$

$$x = \lfloor \frac{A(i,j)}{T^r_{r_i}} \rfloor \tag{3.3}$$

Equation 3.4 is derived from the right-most terms of Equation 3.2 substituted with Equation 3.1. It presents the upper bound of the number of messages k_i. Let $\delta^-_{s_i,r_i}$ be the best case network delay between the sender and receiver, and $\delta^+_{s_i,r_i}$ be the worst case network delay. Equation 3.5 presents the number of messages for the i^{th} task that can be queued in the operating system network buffer at the receiver in the worst case, derived from substituting x in Equation 3.4 according to Equations 3.3 and 3.1.

$$k_i \leq \frac{(x+1)T^r_{r_i} - \delta_{s_i,r_i,j+k_i-1} - O_{s_i,r_i}}{T'_i} - j + 1 \tag{3.4}$$

$$\begin{aligned} k_i = 1 + \lfloor \frac{T^r_{r_i}}{T'_i} + \frac{\lfloor \frac{jT'_i+\delta^+_{s_i,r_i}+O_{s_i,r_i}}{T^r_{r_i}} \rfloor T^r_{r_i} - jT'_i - \delta^-_{s_i,r_i} - O_{s_i,r_i}}{T'_i} \rfloor \\ \leq 1 + \lfloor \frac{T^r_{r_i}}{T'_i} + \frac{\delta^+_{s_i,r_i} - \delta^-_{s_i,r_i}}{T'_i} \rfloor \\ \mathit{with}\ \delta_{s_i,r_i,j} \leq \delta^+_{s_i,r_i}\ \mathit{and}\ \delta_{s_i,r_i,j+k_i-1} \geq \delta^-_{s_i,r_i}. \end{aligned} \tag{3.5}$$

The intuition of Equation 3.5 is that in order to maximize the number of queued messages for an event between two receiving jobs, the first message queued for that event has to arrive with the worst network delay and the last message queued before the receiving job has to arrive with the best network delay.

The first three jobs of τ_1 in Fig. 3.2 illustrate this worst case. Assume the network delay between the two nodes is in the range $[1, 3]$. The message of the first job $\tau_{1,1}$ arrives after the first receiving job of the second node is invoked because of the large network delay (i.e., 3). After that, $\tau_{1,2}$ and $\tau_{1,3}$ arrive, which results in the worst-case number of message arrivals from τ_1 (i.e., $k_1 = 3$). Equation 3.5 with $T^r_{r_i} = 5$, $T'_i = 3$, $\delta^+_{s_i,r_i} = 3$, and $\delta^-_{s_i,r_i} = 1$ derives exactly this worst case.

When the s^{th} queued message is eventually accepted by the $(x+1)^{th}$ receiving job, the scheduler releases a job to process the message. The relative deadline of the job should be the difference of the absolute deadline carried in the message adjusted by the clock time difference, $(j + s)T_i + D_i + O_{s_i,r_i}$, and the time when the job enters the EDF queue, $(x + 1)T^r_{r_i} + C^r_{r_i}$. Equation 3.6 shows the relative deadline, $D'_{i,s}$, of this job (substitute x from Equations 3.3 and 3.1). A negative job deadline indicates that the absolute deadline carried in the message is shorter than the worst-case time $(T^r_{r_i} + C^r_{r_i} + \delta^+_{s_i,r_i})$, i.e., the time required for the partial-EDF on the receiver to release a corresponding job to process the message. In this case, this message cannot be processed before its deadline.

$$\begin{aligned}
D'_{i,s} &= ((j+s)T_i + D_i + O_{s_i,r_i}) - ((x+1)T^r_{r_i} + C^r_{r_i}) \\
&\geq ((j+s)T'_i + D_i + O_{s_i,r_i}) - ((x+1)T^r_{r_i} + C^r_{r_i}) \\
&\geq (sT'_i + D_i) - (T^r_{r_i} + C^r_{r_i} + \delta^+_{s_i,r_i}), \\
&\qquad \textit{where } \delta_{s_i,r_i,s} \leq \delta^+_{s_i,r_i} \textit{ and } 0 \leq s \leq k_i - 1. \qquad (3.6)
\end{aligned}$$

If these jobs can be executed before the $(x + 2)^{th}$ receiving job, the task set is schedulable. Equation 3.7 defines the total density of periodic tasks on node m (the density of local periodic tasks are included). Equation 3.8 defines the total density of jobs that process messages sent to node m in this case. $L(i, m)$, $S(i, m)$, and $R(i, m)$ are indicator functions. $L(i, m) = 1$ iff τ_i is a local periodic task on node m. $S(i, m) = 1$ iff node m is the sender of distributed task τ_i. $R(i, m) = 1$ iff node m is the receiver of distributed task τ_i. Otherwise, $L(i, m)$, $S(i, m)$ and $R(i, m)$ are 0. The density test on node m for this case is expressed in Equation 3.9. This density test has to pass for all M nodes.

$$\rho_m = \sum_{i=1}^{N}(L(i,m)+S(i,m))\frac{C_i}{min\{T_i, D_i\}} + (\frac{C_m^r}{T_m^r} + \frac{C_m^s}{T_m^s}),$$

$$L(i,m) = \begin{cases} 1, & if\ \tau_i\ is\ local\ on\ m, \\ 0, & otherwise. \end{cases}, S(i,m) = \begin{cases} 1, & if s_i = m, \\ 0, & otherwise. \end{cases} \tag{3.7}$$

$$\rho'_m = \sum_{i=1}^{N} R(i,m) \sum_{s=0}^{k_i-1} \frac{C'_i}{min\{T_m^r, D'_{i,s}\}},$$

$$R(i,m) = \begin{cases} 1, & if\ r_i = m, \\ 0, & otherwise. \end{cases} \tag{3.8}$$

$$\rho_m + \rho'_m \leq 1 \tag{3.9}$$

Our schedulability test accuracy depends on the variance of network delays as shown in Equation 3.5. We will introduce our bandwidth sharing policy in Section 3.2.3, which reduces contention and network delay variance. In addition, absolute deadlines carried in messages have to be adjusted at the receiver to compensate for pairwise time differences across nodes. Otherwise, our partial-EDF cannot schedule corresponding jobs in EDF order. We will introduce the time-difference table maintained on each node in Section 3.2.2 to address this problem.

3.2.2 Time-difference Table

Our scheduler depends on the task deadlines carried in the messages to schedule the jobs to process these messages in EDF order. Since the messages received on a node could come from different senders and the absolute deadlines carried in these messages are derived from the local clock of the senders (i.e., different reference frames), the receiving node must have adequate knowledge of the time difference to maintain the EDF order of job executions. Global clock synchronization could solve this problem but is costly [28]. Instead of clock synchronization, our scheduler maintains a time-difference table on each node. This table contains the difference between the clocks on nodes that send messages to each other. When one node receives messages from another node, it uses the time-difference table to adjust the absolute deadlines carried in the messages, so that the deadlines in all messages are adjusted to the local clock of this receiver.

We utilize the algorithm of the Network Time Protocol (NTP) [37] to derive the time difference between each pair of sender and receiver. We assume the sender s and receiver r of a task τ to be known a priori. In our system, the receiver r transmits a synchronization message to s and s transmits a feedback message to r. On the basis of the timestamp exchanged via this pair of messages, the receiver r can derive the round trip delay $\delta_{s,r}$ and clock offset $O'_{s,r}$ The constraint of the true time offset for the sender relative to the receiver, $O_{s,r}$, is expressed in Equation 3.10. Following the NTP design, the data-filtering algorithm is performed on multiple pairs of $(\delta_{s,r}, O'_{s,r})$ collected from timestamps exchanges to derive the three time offset. Each node in our distributed system has a periodic task to transmit time synchronization messages and to calculate its local time-difference table dynamically.

$$\delta_{s,r} - \frac{O'_{s,r}}{2} \leq O_{s,r} \leq \delta_{s,r} + \frac{O'_{s,r}}{2} \tag{3.10}$$

With the time-difference table, the absolute deadline carried in messages can be adjusted to the local time on the receiver (i.e., $D'_r = D_s + O_{s,r}$).

3.2.3 EDF Packet Scheduler and Bandwidth Sharing

We utilize a traffic control mechanism provided by Linux to regulate the message traffic that is transmitted via the network in order to reduce network contention and variance of network delay. By limiting the bandwidth per node and by reducing the burstiness of the traffic, network contention can be reduced. In addition, we extend Linux traffic control by implementing a deadline-driven queue discipline, the *EDF packet scheduler*, to compensate for the shortcoming of our partial-EDF scheduler that it may transmit the messages for tasks with longer deadlines earlier than the messages for tasks with shorter deadlines.

Let us assume that each node in the distributed system is connected by a single full-duplex physical link to a switch and the network interface works in work-conserving mode. A node transmits messages to other nodes via the same link. We configure the network interface to use the Hierarchical Token Bucket (HTB) [20] algorithm to transmit messages. HTB has two buckets: The *real-time bucket* is utilized to transmit packets of messages that serve real-time tasks and the *background traffic bucket* is utilized to transmit background traffic. This design requires that the IPs and ports of receiving nodes for real-time tasks are known a priori, so that their messages can be differentiated from other traffic and put into the real-time bucket. We configure HTB to have a (σ, ρ) regulator [11] for each bucket. This regulator ensures that the total size of packets transmitted from that bucket during the time interval $[s, e]$ is bounded by $\sigma + \rho * (s - e)$, where σ determines the burstiness of the traffic, and ρ determines the upper bound of the long-term average rate.

In order to configure HTB so that it benefits real-time systems, we propose several policies (1) to prioritize buckets, (2) to schedule packets in the real-time bucket, and (3) to share bandwidth among nodes. First, the real-time bucket has a higher priority than that of the background traffic bucket. Task packets are transmitted before background traffic packets when both buckets are legitimate to send packets according to the (σ, ρ) rule.

Second, the real-time bucket employs an EDF packet scheduler to schedule packets. Since the Linux traffic control layer does not understand task deadlines embedded in data messages, which are encapsulated by the application layer, we extend the data structure for IP packets in Linux by adding new fields to store timestamps and extend the *setsockopt* system call by supplying these timestamps. These extensions provide the capability of specifying message deadlines for real-time tasks (applications). Then, our EDF packet scheduler utilizes the message deadlines to transmit packets in EDF order. By employing the EDF packet scheduler, we compensate for shortcomings of our partial-EDF job scheduler. As mentioned in Section 3.2.1, because of the delay between message arrivals and receptions introduced by the receiving task in our partial-EDF scheduler, messages that arrive at a node but have not been accepted into the job queue may be processed later than the jobs already in the queue, even if these messages have shorter deadlines. As a result, messages carrying short deadlines are put into the real-time bucket later than packets with longer deadlines. By considering the deadlines carried in the messages at the transport layer, our EDF packet scheduler compensates for this shortcoming by rescheduling the messages. In contrast, the background traffic bucket simply employs a FIFO queue to transmit background traffic.

Third, we consider the underlying network utilized by nodes as a whole system and propose two strategies of bandwidth sharing among nodes. Let w_m be the proportion of bandwidth for node m and M be the number of nodes. The first strategy is fair-share (i.e., $w_m = \frac{1}{M}$). This strategy is simple to understand but does not consider the different transmission demands of nodes. Consider a scenario where two periodic tasks on node 1 transmit messages to node 2, of which only one periodic task transmits messages back to node 1. Node 1 could benefit from a larger fraction of bandwidth since it has a larger transmission demand. Thus, the second strategy shares bandwidth proportionally considering the number of tasks on each node and the periods of these tasks. Let us assume the task sets on nodes were known a priori so that the denominator in Equation 3.11 represents the aggregate of transmission demands on all nodes and the numerator the transmission demand of node m.

$$w_m = \frac{\sum_{i=1}^{N} S(i,m)\frac{1}{T'_i}}{\sum_{i=1}^{N} \frac{1}{T'_i}} \tag{3.11}$$

The experimental evaluation results in Section 3.4 show that our EDF packet scheduler can decrease the deadline miss rate of messages significantly and the bandwidth sharing policy can decrease the network delay deviation in a local cluster of nodes. In Section 3.3, we will elaborate on the implementation of our real-time distributed storage system, which employs the hybrid EDF scheduler to schedule jobs and packets.

3.3 Real-Time Distributed Storage System

In this section, we summarize our prior contributions [47], i.e., we present the features of our real-time distributed storage system and its message routing algorithm, which provides the basis of our novel contributions in this work. We then present our integration of a hybrid EDF scheduler to the RT-DHT.

3.3.1 DHT

As a storage system, our RT-DHT provides *put(key, data)* and *get(key)* API services to upper layer applications. This RT-DHT manages a group of storage nodes connected over the networks and implements a consistent hashing algorithm [25] to map a key onto a node that stores the data. When a *put* or *get* request is sent to any node in the system, a *lookup(key)* message is forwarded among these nodes following a particular forwarding path until it arrives at the node to which this key is mapped.

We adopt the Chord algorithm [40] to determine the forwarding paths for requests. Chord organizes storage nodes in a ring-shaped network overlay [40, 25]. It uses a base hash function, such as SHA-1, to generate an identifier for each node by hashing the node's information, e.g., IP address and port. It also uses the same hash function to generate an identifier for the key of a request. The purpose of using the same hash function is to generate the identifiers of nodes and keys in the same domain in which ordering is defined. If node A is the first node after node B clockwise along the ring, A is called the predecessor of B, and B is called the successor of A. Similarly, a key also has a successor node. For example, Fig. 3.3 depicts 9 storage nodes (green) that are mapped onto the Chord ring (labels in squares are their identifiers), which has 5-bit identifiers for nodes and keys. *N30* is the successor of *N24* and the predecessor of *N1*. The successor node of *K25* is *N30*. The Chord algorithm defines the successor node of a key as its destination node, which stores the corresponding data of the key.

Chord maintains a so-called finger table on each node, which acts as a soft routing table to decide the next node to forward lookup messages to. Each entry in that table is a tuple of (start, interval, forward node), as shown in Table 3.1 of *N4*'s finger table. Each entry in the table represents one routing rule: the next hop for a given key is the forward node in that entry if the entry interval includes the key. For example, the next hop for *K25* is *N20* as *25* is in *(20, 36]*. Since 5 bits are used to represent node id, (20, 36] equals to (20, 4] (modulo 32). In general, for a node with identifier k, its finger table has $logL$ entries, where L is the number of bits of identifiers generated by the hash function. The start of the i^{th} finger table entry is $k + 2^{i-1}$, the interval is $(k + 2^{i-1}, k + 2^{i}]$, and the forward node is the successor node of key $k + 2^{i-1}$. Because of the way a finger table is constructed, each message forwarding reduces the distance to its destination node to at least half that of the previous distance on the ring. The

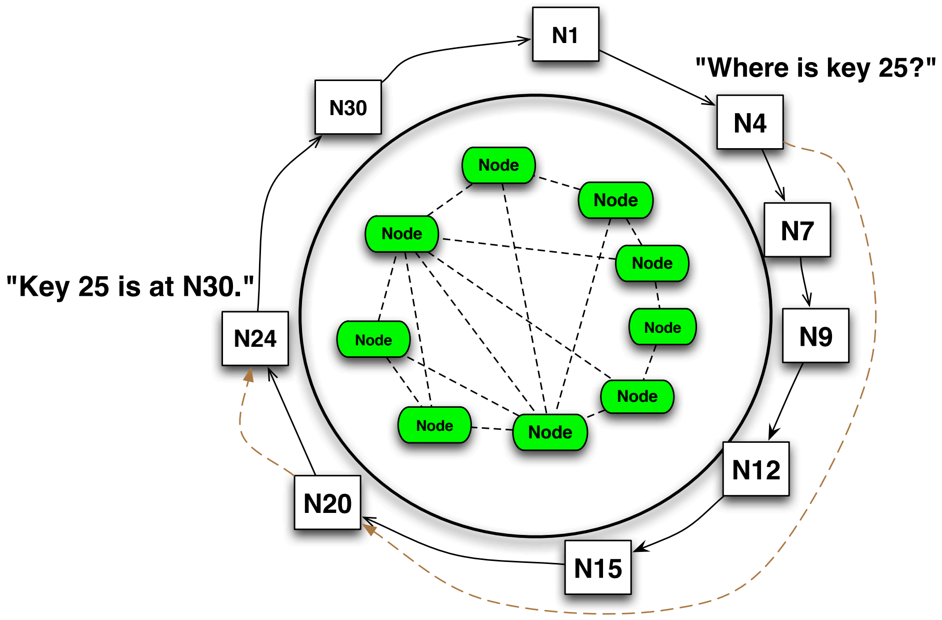

Figure 3.3: Chord Ring Example

Table 3.1: Finger table for *N4*

#	**start**	**interval**	**forward node**
1	5	(5, 6]	N7
2	6	(6, 8]	N7
3	8	(8, 12]	N9
4	12	(12, 20]	N12
5	20	(20, 4]	N20

number of intermediate nodes per request is at most $logM$ with high probability [40], where M is the number of nodes in the system.

Chord provides high efficiency and deterministic message forwarding in the stable state (i.e., no node failures, node joins, and network failures). This is important to our RT-DHT since based on this property, we can apply the bandwidth sharing policy and perform a schedulability test on a given task set to determine whether the deadlines of the requests in the task set can be met by our partial-EDF scheduler.

3.3.2 Hybrid EDF Scheduler Integration

We have implemented our RT-DHT in Linux. Each storage node in the system employs a partial-EDF job scheduler using a single thread. As described in Section 3.2.1, the static schedule table includes the temporal information of periodic tasks. Based on this temporal information, the scheduler sets up a periodic timer. When the signal handler (at the application level) is triggered by the timer interrupt, the scheduler, by executing the handler, releases the jobs of periodic tasks into a job queue according to the table. After that, the scheduler executes the jobs waiting in the queue in EDF order.

This work extends the Linux system by providing support for the RT-DHT storage node for message deadlines and for transmitting messages in EDF order (see Section 3.2.3). Below are the most significant changes we made to Linux to support this functionality.

(1) We added a new field, *deadline*, of type *ktime_t* in the kernel data structure *sock* so that deadline of a socket provided by the application can be stored.

(2) We extended the kernel function *sock_setsockopt* and added a new option $SO\_DEADLINE$ so that the application can utilize *setsockopt* (i.e., the corresponding user mode function) to configure the message deadlines when the application attempts to transmit messages. *sock_set-sockopt* keeps the value of the deadline in the *deadline* field of the *sock* structure.

(3) When the application transmits a message, the kernel creates instance(s) of the *sk_buff* structure to store the data of the message. We added a new field, *deadline*, in *sk_buff* so that the *deadline* of a socket can be saved when the kernel creates the data structure. After this, the deadline of the message in *sk_buff* is passed down to the transport layer.

(4) We implemented an EDF packet scheduler, which provides the standard interfaces of Linux traffic control queue disciplines [20]. The EDF packet scheduler utilizes a prioritized queue to maintain the instances of *sk_buff* in a min-heap data structure. We utilize the *cb* field, the control buffer in *sk_buff*, to implement a linked list-based min-heap. This linked list-based implementation does not have a limit on the number of messages that can be queued in the min-heap, which an array-based implementation of min-heap would have.

We combine Linux HTB with our EDF packet scheduler and associate the hierarchical queue with the network interface that connects the storage node with the DHT overlay as described in Section 3.2.3. Since the EDF packet scheduler implements the standard interfaces of queue disciplines, the storage system can utilize the Linux traffic control command (i.e., *tc*) to configure the queue when the system boots up.

Section 3.2.1 presented the schedulability test for the partial-EDF job scheduler. However, more than two storage nodes may be involved to serve the same data request of a distributed task in the RT-DHT. For example, to serve the request that looks up the data for key 15 on $N4$ in Fig. 3.3, $N4$ first executes a job to transmit the request message to $N12$ following the finger table. Then, $N12$ executes a corresponding job to process the message and transmits another message to $N15$ via its sending task. $N15$ obtains the data from its storage and transmits a feedback message back to $N4$. We adjust the task set in the schedulability test to accommodate this as follows.

Let us use τ to represent the original task set, in which the message forwarding path for τ_i is $P_i = (n_1, n_2, ..., n_{m_i})$, where m_i is the length of the path. Let us denote $\tau_i = (T_i, C_i, D_i) \xrightarrow{n_1,n_2} (T'_i, C'_i)$ to represent task τ_i that initializes data requests on node n_1 periodically. We first replace τ_i with τ_i^1 as shown in Equation 3.12. The partial-EDF scheduler on n_1 schedules τ_i^1 with its new absolute deadline $\frac{D_i}{m_i}$. We share the end-to-end deadline D_i evenly among m_i nodes to simplify the representation of our schedulability test model. Past work has provided sophisticated protocols for the subtask deadline assignment problem (SDA) [24, 53, 54], which can be adopted by our schedulability model. When the receiving job on n_2 releases the corresponding job to process the message of τ_i, it increases the deadline of the job by $\frac{D_i}{m_i}$. Replacing τ_i by τ_i^1 only changes Equation 3.7 in Section 3.2.1 if $m_i = 2$ (i.e., $D_i \rightarrow \frac{D_i}{2}$).

$$\tau_i^1 = (T_i, C_i, \frac{D_i}{m_i}) \xrightarrow{n_1,n_2} (T'_i, C'_i) \tag{3.12}$$

On node n_2, the jobs created by τ_i^1 transmit messages to n_3. To reflect this new task on n_3 in the schedulability test, we define a virtual task $\tau_i^2 = (*, 0, \frac{D_i}{m_i}) \xrightarrow{n_2,n_3} (T'_i, C'_i)$. The density of the jobs (on sender n_2) of τ_i^2 is already considered in Equation 3.8 since they are the very jobs that process the received messages of τ_i^1. The density of the jobs (on receiver n_3) of τ_i^2 is considered in Equation 3.8 for node n_3. This task is virtual since the scheduler on n_2 does not actually schedule any jobs of the task. We set its WCET to 0 so that it can be integrated into Equation 3.7. As indicated in Equations 3.4 to 3.8, the density of the jobs on receiver n_3 does not depend on the period of τ_i^2. Thus, the period of τ_i^2 is set to *. In general, Equation 3.13 expresses the virtual task τ_i^j for each node n_j on the path.

$$\tau_i^j = (*, 0, \frac{D_i}{m_i}) \xrightarrow{n_j,n_{j+1}} (T'_i, C'_i), where\ 2 \leq j < m_i. \tag{3.13}$$

Finally, we perform the schedulability test on the new task set, which is the union of τ and all virtual task, with Equations 3.5 - 3.9 by adjusting Equation 3.6 ($D_i \rightarrow \frac{2D_i}{m_i}$) and Equation 3.7 ($D_i \rightarrow \frac{D_i}{m_i}$).

To simplify the presentation of these equations, we assume that the jobs to process messages of the same task have the same WCET even when they are executed on different storage nodes (i.e., C'_i is inherited by the virtual task). However, the schedulability test does not depend on inheritance of WCETs.

3.4 Experimental Results and Analysis

In this section, we evaluate our hybrid EDF scheduler on a local cluster. The experiments are conducted on 8 nodes, each of which features a 2-way SMP with AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node). Each node has 32GB DRAM and

Gigabit Ethernet (utilized in this study). These nodes are connected via a single network switch. Each node runs a modified version of Linux 2.6.32, which includes the EDF packet scheduler.

In the first experiment, we evaluate the contribution of the bandwidth sharing policy in terms of decreases in network delay variance. Then, we evaluate the impact on task deadline miss rate when employing our EDF packet scheduler. Next, we evaluate the partial-EDF job scheduler by running different task sets on the RT-DHT.

3.4.1 Bandwidth Sharing Results

In this experiment, we use the network benchmark, Sockperf, to measure the application-to-application network delays in ping-pong mode. We utilize 8 nodes, each of which runs a Sockperf server and a client. Each Sockperf client periodically transmits messages as TCP packets to the servers and a server sends a feedback packet back to the client immediately. The clients calculate network delays based on the time of sending the message and receiving the feedback. Sockperf clients can only transmit a fixed number of messages (specified with *burst* option) within a period of time. However, a real-time application could have different message transmission demands in different frames. We extend the Sockperf implementation to transmit varying numbers of messages during different time periods. In our experiment, the varying numbers are drawn round robin from a population of bursts derived from *burst* option, and the destination (server) of the message is chosen from the 8 servers in a round robin fashion. As a result, the number of messages transmitted (i.e., burstiness) in different frames can be different. We compare the network delay variance with and without bandwidth sharing policy. For the experiments of bandwidth sharing, we configure the network interface of server and client to send messages into two token buckets: one for the client to transmit messages and the other one for the server to transmit feedback messages. A third token bucket is added to transmit unclassified (background) traffic including the packets of other applications running on the same node.

Fig. 3.4 depicts the network delay interval on one node (x-axis) over the network delay (round-trip delay) in milliseconds (y-axis). The blue clusters depict the network delays for the bandwidth sharing policy. The interval of blue clusters extends from 0.14 to 3.69 (i.e., a range of 3.55) altogether, which is larger than 0.97, the range without bandwidth sharing policy. However, the large variance is not only due to network delays. Since the token bucket implementation of Linux is driven by kernel timer events, once a message is pending in the queue (i.e., not transmitted by the network interface immediately due to the bandwidth sharing policy), the message has to wait at least one timer interrupt interval until the next time the network interface considers to transmit messages. The timer interrupt interval is $1ms$ (i.e., $HZ = 1000$). Considering the cost of the token bucket implementation, we divide the measured application-to-application delays into different clusters and calculate the network delay variance in each cluster. As depicted in Fig. 3.4, each blue cluster represents a cluster of network delays with the variance marked on the x-axis. The range of all blue clusters is less than the 0.97 range of the red cluster, which illustrates the benefit of our bandwidth sharing policy.

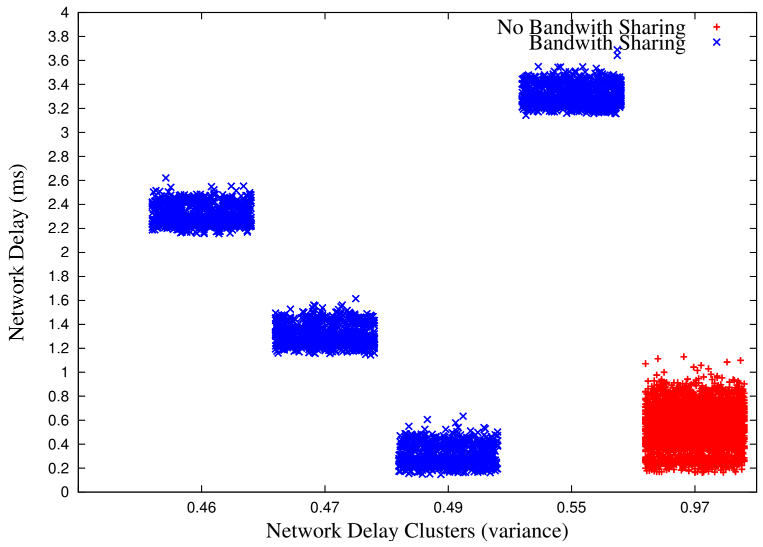

Figure 3.4: Network Delay Comparison (1) for Delay Clusters (ms)

Fig. 3.5 depicts the network delays for all 8 nodes after the token bucket costs are removed in the measured data. We observe that the bandwidth sharing policy reduces network delay variance on all nodes. The cost of bandwidth sharing contributes significantly to application-to-application network delay since we utilize a Gigabit network switch and NICs in our experiment. However, we believe that the proportion of the cost will reduce in complex network topologies, where cumulative network queuing delays on intermediate network devices could change significantly without bandwidth sharing policy, i.e., our reference without bandwidth sharing in the experiments is the best case in a distributed system.

The challenge of applying bandwidth sharing is to determine the limit of the packet transmission rate according to the transmission demand of the applications and the transmission capability of the underlying network. If the transmission rate is too large, packets are transmitted into the network immediately and network contention could be significant resulting in large network delay variance. If the rate is too small, packets could wait in the bucket for a long time resulting in large application-to-application delays. In Section 3.2.3, we proposed to share the total rate proportionally according to the demand of an individual node. However, we did not provide a theoretic method to determine the total rate. In the experiment for Fig. 3.4

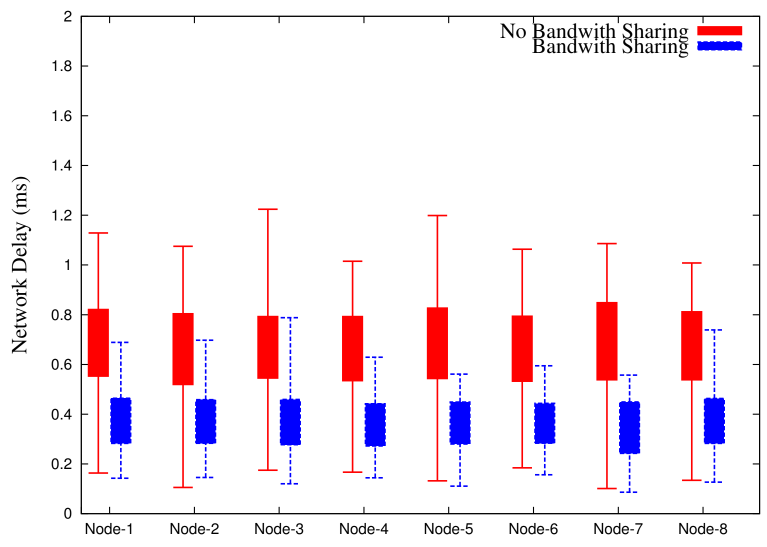

Figure 3.5: Network Delay Comparison (2)

and 3.5, we configure Sockperf to simulate a $10ms$ frame size (set option mps to 100) and in each frame, the number of messages are chosen from $[5, 10]$ in a round robin fashion (set option $burst$ to 10 for an the interval of $[\frac{burst}{2}, burst]$). The message size is 400 bytes. Then, we tune the rate experimentally and select $2.45Mbps$ as the client transmission rate and $19.60Mbps$ as the server transmission rate.

3.4.2 EDF Packet Scheduler Results

In this section, we extend the network benchmark, Sockperf, so that messages sent by Sockperf clients carry deadline information. Sockperf clients specify the deadline (by using the *setsockopt* system call) before sending a message and Sockperf servers immediately send a feedback message with a specified message deadline. The feedback message inherits the deadline of the original message sent by the client. A Sockperf client compares the arrival time of a feedback message and the deadline carried in that message to determine if this message has met its deadline. In each experiment, we first run Sockperf without the EDF packet scheduler on 8 nodes (one client and one server on each node), then we run Sockperf in playback mode, which transmits the same sequence of messages with the EDF packet scheduler installed on the same nodes. We compare their deadline miss rates.

The workload is controlled by Sockperf options $(mps, burst)$. The number of messages transmitted in a frame by a client is chosen from an interval $[\frac{burst}{2}, burst]$ in a round robin fashion. Each message is 400 bytes. In addition, the deadline of a message is the current time when the message is created, increased by a random value uniformly drawn from the interval $[D, 2D]$, where D is a configurable parameter in different experiments.

Fig. 3.6 depicts the deadline miss rates on one node under different workloads and deadline setups. Line *0.8-NoEDF* depicts the change of deadline miss rates when the workload changes from $mps = 100, burst = 40$ to $mps = 5000, burst = 200$. D is $0.8ms$, and no EDF packet scheduler is attached to the network interface of the nodes. The line indicates that before workload $(4400, 40)$, the deadline miss rate is stable (about 5%). After that, the deadline miss rate increases significantly when the workload increases. This is explained as follows: When the workload is small, the network interface can transmit messages immediately into the network. In this case, the deadline misses are caused by the network delay. When the workload increases to large values (i.e., larger than $(4400, 40)$ in our experiment), message packets start to be queued in the network buffer, which increases their transmission time. Thus, deadline misses increase significantly. In addition, the probability of a message being queued depends on the burst size, since a larger burst size suggests that the Sockperf client attempts to transmit more messages in a frame.

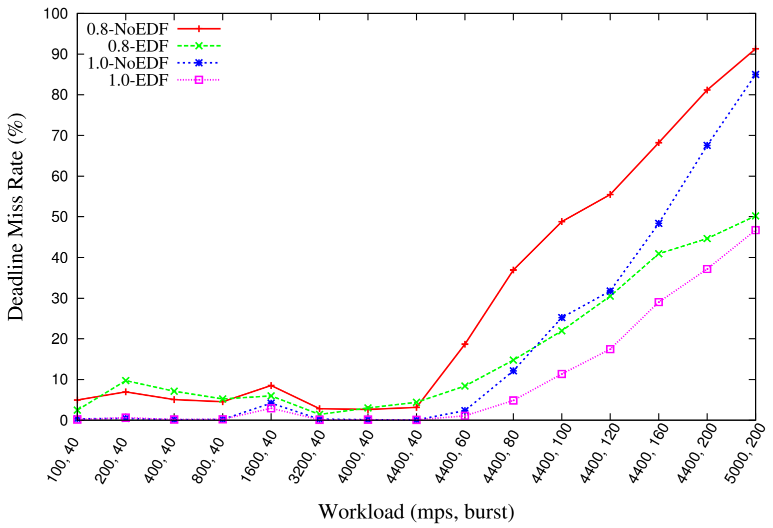

Figure 3.6: Deadline Miss Rate Comparison

As a comparison, line *0.8-EDF* depicts the change of deadline miss rates under the same setup, except that each node adopts an EDF packet scheduler. We observe a similar trend of deadline miss rates that change with the workload as for line *0.8-NoEDF*. However, due to the capability of re-scheduling packets, the EDF packet scheduler decreases the deadline miss rates when the network interface is dominated by bursty data. Lines *1.0-NoEDF* and *1.0-EDF* depict the comparison of deadline miss rates when D is $1ms$, which suggest a similar trend as that of $D = 0.8ms$.

3.4.3 Partial-EDF Job Scheduler Results

We next experiment with our RT-DHT storage system to measure the value of k_i defined in Equation 3.5, which is the worst number of messages that can arrive at one node for task τ_i, since k_i is the basis of the schedulability test of the partial-EDF scheduler. Then, we compare the measurement with the value calculated by our formula. To simplify the analysis, we predefine the identifiers of nodes and keys on the chord ring instead of using a hash function to calculate these identifiers. We utilize 8 storage nodes in this experiment.

In the first experiment, we run a single distributed put-request task $\tau_1 = (10, 1, 100) \xrightarrow{0,1} (5, 1)$ on node 0, which only involves nodes 0 and 1 (all times are expressed in ms). The partial EDF scheduler on node 0 releases jobs of τ_1 with a period of $10ms$ and the sending task guarantees that the inter-transmission time for the message of τ_1 is at least $5ms$. The frame size of node 0 is $5ms$ (we require that $T_0 \geq T'_0$). In addition, a local periodic task $\tau_2 = (60, 50, 60)$ is scheduled on node 0. We busy wait to make the real execution time of a task match its WCET. The receiving task on node 1 has a period of $10ms$. We observe that the worst-case number of message arrivals on node 1 is 3, which matches the value calculated by Equation 3.5 (i.e., $k_1 = 1 + \lfloor \frac{10}{5} + \frac{\delta^+ - \delta^-}{5} \rfloor$), since both δ^+ and δ^- are small in our local cluster compared to the message sending period ($5ms$). However, we cannot simply decrease the period of τ_1 to the level of network delay considering the minimal time slices of Linux ($1ms$ in our experiment). Instead, we implement a traffic control utility to simulate large network delays. We add a $10 \pm 5ms$ network delay to node 0, which results in a delay range of $[5.25ms, 15.89ms]$. (The delay range is measured via the ping-pong mode of Sockperf. It changes over time but the changes are in $\pm 0.5ms$). With this setup, the worst-case number of message arrivals is 4 (i.e., smaller than the upper bound 5 calculated from Equation 3.5).

Then, we update τ_1 so it releases *put* requests that require nodes 0, 3 and 4. In this case, the message of τ_1 is first forwarded to node 3 and from there forwarded to node 4. Both nodes 3 and 4 have a $10ms$ message receiving period and a $5ms$ message sending period. We observe the same worst-case number of message arrivals, i.e., 4.

However, the temporal properties of the receiving task in our partial-EDF scheduler have to be tuned to suit different workloads. For example, the WCET of the receiving task determines the most number of messages it can copy from the kernel. In one experiment, each node has four distributed tasks and each of them involves three nodes. The periods of these tasks are $40ms$ each. The receiving task of $1ms$ WCET cannot always copy all messages from the kernel and release the corresponding jobs to process these messages. This leaves as an open question:

How should one theoretically determine the proper WCET of the receiving task as well as the sending task?

3.5 Related work

Past research has put forth methodologies for adjusting the interrupt-handling mechanisms of operating systems to provide predictable interrupt-handling mechanisms for real-time application tasks [65, 15, 31]. These approaches improves the predictability of the kernel-level interrupt handling, while our partial-EDF job scheduler focuses on improving the predictability of the interrupt handling at user level. Compared to other structured user-level real-time schedulers that provide predictable interrupt management [43], our partial-EDF job scheduler simply adopts a static scheduling mechanism (i.e., cyclic executive [5]) with an EDF scheduler. This simplification allows us to use a density-based schedulability test on a task set by refining the temporal properties of tasks in the table (for example, the timer interrupt handler, the message sending task, and the message receiving task in our system). Thus, these tasks can be modeled as periodic tasks in the density-based schedulability test. This also allows us to perform a schedulability test on a set of distributed tasks, in which we consider the temporal dependency among tasks on different distributed nodes.

Resource sharing significantly affects the predictability of real-time systems where multiple real-time tasks may attempt to access the shared resource simultaneously. Past work has focused on shaping the resource access patterns to provide predictable response times to concurrent real-time tasks [63, 13, 55, 30]. For example, bursty memory accesses were delayed so that the number of memory accesses by a task meets a bound during a period of time. Shaping is also utilized to improve the schedulability of fixed-priority real-time distributed systems that contain periodic tasks with a release jitter. [46]. We apply a similar methodology, which utilizes Linux traffic control mechanisms to shape the out-going traffic on distributed nodes in order to improve the predictability of the network delay (i.e., a decrease in variance of network delay). In contrast to past work [46], our experimental results have shown the benefit of applying shaping, which decreases the network delay variance by reducing network congestion in general in distributed systems, where tasks not only experience release jitter, but the number of tasks can also vary over time. However, these aforementioned resource-sharing mechanisms are *passive* since they only reduce the probability of resource contention instead of eradicating contention altogether by only controlling the resource access of end nodes. Real-time tasks may benefit more from *active* resource sharing mechanisms, which perform access control via resource controllers [2] or real-time state-ful communication channels [23]. These mechanisms require intermediate devices (e.g., network routers) to cooperate. Today, software-defined networks (SDN) [36] allow intermediate network devices (i.e., SDN capable switches and routers) to control traffic flows explicitly, which may reduce the network delay variance even for complex network topologies.

We implement an EDF packet scheduler that transmits packets in EDF order at end nodes. Previous research has also proposed to transmit real-time traffic over entire networks [18]. However, this requires modifications to the IP packets and end nodes to establish stateful communication channels before transmitting data. In addition, network devices have to maintain channel states and be aware of the deadlines carried in IP packets.

3.6 Conclusion

We have presented a hybrid EDF scheduler for distributed real-time systems. At the application level, our partial-EDF job scheduler utilizes periodic tasks to transmit messages over networks to decrease the impact of application-level interrupt handling on system predictability, while the scheduler schedules tasks in EDF order. We have proposed a density-based schedulability test on a task set for our partial-EDF job scheduler. In the transport layer, we extend the Linux network stack to support message deadlines and implement an EDF packet scheduler to transmit messages in EDF order. We also propose to utilize Linux traffic control mechanisms to decrease network delay variance, which may increase the predictability of distributed tasks. Our experimental results showed that the EDF packet scheduler can decrease the deadline miss rate and the traffic control mechanism can decrease network delay variance in a local cluster. In addition, we demonstrated the effectiveness of these techniques by integrating them into our RT-DHT storage system so that data requests can be served in EDF order.

Future work includes experimenting the traffic control mechanism in complex network topologies and performing fine-grained network resource control, for example, adopting software-defined network techniques to prioritize messages on intermediate network devices according to message deadlines. In doing so, we wish to study the impact of our hybrid EDF scheduler in a large-scale environment, where more nodes could process real-time distributed tasks simultaneously. In addition, we wish to find a better way to evaluate our partial-EDF job scheduler.

Chapter 4

A Linux Real-Time Packet Scheduler for Reliable Static Routing

4.1 Introduction

In a distributed computing environment, multiple compute nodes share communication resources to transmit data in order to collaborate with each other. In such systems, employing an effective resource sharing mechanism is essential to meet the real-time requirements of tasks. These mechanisms can be divided into two categories. First, the compute nodes control their own behavior of how to utilize shared resources. Past research has studied mechanisms of shaping the resource access pattern to increase timing predictability, e.g., memory sharing based on limiting the memory bandwidth for different cores [62, 63] and network bandwidth limitation on compute nodes connected via Ethernet [49]. These mechanisms are *passive* since they can only reduce the probability of resource contention instead of preventing contention in the first place. Thus, passive mechanisms usually guarantee probabilistic deadlines. The second category includes *active* resource sharing mechanisms, which either assign the resource to exactly one task at a time to prevent contention (e.g., TDMA-based bus sharing on a multiprocessor platform [55]), or provide mechanisms on the shared resources to control its usage directly [2, 23]. One benefit of active mechanisms is that by controlling how resources are shared, one can build up a mathematical model to estimate the resource access time. We believe this model is a necessary condition to guarantee hard deadlines.

In our previous work, we implemented a cyclic executive based task scheduler on distributed nodes to guarantee probabilistic task deadlines [47] and a hybrid earliest-deadline-first (EDF) task and packet scheduler to guarantee hard deadlines [49]. Furthermore, we have implemented a real-time distributed hash table (RT-DHT) to support the scalability and resilience requirements of distributed wide-area measurement systems, which estimate power grid oscillation modes based on real-time power state data [48, 41]. A passive mechanism (bandwidth limitation on DHT nodes) was adopted to reduce variations of the network delay in order to increase the schedulability of our hybrid scheduler [49]. However, when the RT-DHT has to share network resources with other systems whose network utilization patterns are unpredictable, this passive mechanism is not enough since it cannot restrict the network access of other systems. In this work, we actively control network devices (e.g., switches) by enforcing a packet scheduling algorithm on the devices to guarantee hard message transmission deadlines for real-time packets, even when these devices are subject to unpredictable background traffic.

One of the challenges of adopting the active mechanism is to determine an accurate resource sharing policy that considers both the demands of the real-time tasks and the capacities of the shared resources. For example, an active memory controller needs to know the bandwidth demands of the running tasks to proactively reserve bandwidth for each of them. This challenge becomes even harder for a network since the topology of the network can be complex (e.g., network devices have to collaborate in order to guarantee the deadline). We need to derive an effective static routing algorithm to determine the forwarding paths for all real-time packets so that the end-to-end delay for such a packet through the corresponding path never exceeds its deadline. Without loss of generality, we use term *packet* and *message* interchangeably in this chapter.

One possible approach to address this problem is to express it as an integer linear programming (ILP) problem if the objective function and constraints are linear. In our case, each real-time message could have multiple forwarding paths with different costs for each path. For example, one cost is the size of the buffer the message will occupy on every network device along the path. Then the problem becomes to assign a forwarding path for each real-time message under the condition that the constraints on each shared device can be met (in this case, the total size of messages is no more than the buffer size of the shared device at any time). The effectiveness of this model is based on the accuracy of the cost function. However, for the routing problem, the cost function on each device is dependent on the exact traffic that goes through that device (i.e., dependent on the assignment results). This cyclic dependency makes this a hard problem. Previous work has proposed to utilize different oracles to break the cyclic dependency [21]. For example, assuming an oracle that determines the transmission time on any device for any message at any time exists, the assignment problem then can be solved with ILP techniques. In this chapter, we enforce an active message scheduling algorithm on network devices so we can derive the cost function for each message and each device on the path for all traffic that goes through the device. As a result, we use a validation-based backtracking algorithm to determine the forwarding path for each message (detailed in Section 4.2.3).

The objective of our scheduling algorithm is to avoid dropping real-time messages even when network congestion occurs, e.g., due to too many real-time messages and background traffic (considered as non real-time). First, the routing algorithm needs to guarantee that when multiple real-time messages share a device on their forwarding paths, the aggregate message size does not exceed the buffer size of that device. Thus, the scheduling algorithm needs to control the timing when a message is transmitted from one device to the next. The transmission time is planned statically so that the aggregate size of real-time messages on a device can be calculated in our static routing algorithm. Second, the scheduling algorithm needs to drop background traffic when the available buffer on a device is insufficient for real-time messages. In this way, our scheduling algorithm guarantees real-time message deadlines while providing a best-effort service to background traffic.

In summary, the contributions of this work are: (1) We design a static routing algorithm to find forwarding paths for multiple real-time messages to guarantee the hard deadlines of messages. (2) We propose a deadline-based scheduling algorithm, which actively controls the time at which real-time messages are processed at each device and only drops background packets when the device is congested. (3) We implement the packet scheduler on virtual SDN switches and conduct experiments to evaluate our approach.

The rest of chapter is organized as follows. Section 4.2 presents the design of our static routing algorithm and the active device scheduling algorithm. Section 4.3 presents the implementation details of the scheduling algorithm on virtual switches. Section 4.4 discusses the evaluation setup and results. Section 4.5 contrasts our work with related work. Section 4.6 presents the conclusion and on-going research.

4.2 Design

This section first presents the system model and the objectives. It then contributes the static routing algorithm that determines forwarding paths for multiple pairs of source and destination of real-time messages. After that, it contributes the scheduling algorithm that runs on network devices to guarantee that real-time messages are transmitted in a time predictable fashion.

4.2.1 System Model and Objectives

Network Model: We use notation (V, E, B, Pr) to represent the underlying network utilized by compute nodes in distributed real-time systems to exchange messages. V denotes a finite set of compute nodes and the networking hardware, which forwards messages between compute nodes. Without loss of generality, we use the term *node* to represent either a networking device or a compute node. Let B be buffer sizes on nodes, i.e., the aggregate size of messages that can be stored in the queue on each node. If a burst of messages arriving at a node exceeds the buffer size, a fraction of these messages will be dropped. Thus, one objective of our system is

Table 4.1: Notation

Name	Meaning
V	set of nodes
$B[v]$	buffer size on node v
$E[v_1, v_2]$	constant interface speed matrix for real-time traffic between nodes v_1 and v_2
$Pr[v_1, v_2]$	constant propagation delay matrix on links between v_1 and v_2
D_m	relative deadlines of a message flow m
T_m	periods of a message flow m
S_m	size of messages in flow m
$R[v]$	expected message response time on node v
$A[v]$	latest message transmission time on node v
$\delta[v]$	runtime message response time variance on node v
$\Delta[v]$	response time variance bound on node v, determined by all message flows on the node

to guarantee that only background traffic (i.e., messages without deadline requirements) can be dropped in such a case. Matrix E represents the real-time links between nodes in the graph. Nodes v_1 and v_2 have a physical connection for real-time traffic iff $E[v_1, v_2] > 0$, where $E[v_1, v_2]$ is the interface speed. Matrix Pr represents the propagation delays on these links. Table 4.1 summarizes the notation. When a network runs in stable state, its nodes and links do not change over time. Thus, V, E, B, and Pr are constant. We require that clock times on nodes are synchronized. This can be achieved via the Network Time Protocol [37], running at system startup time and periodically as a real-time task, or hardware support (e.g., GPS of phasor measurement units in the power grid).

Node Architecture: We consider a store-and-forward node model. A packet arriving at a node is first put into the input buffer. Then, the node processor (i.e., forwarding engine) processes the packet to determine the forwarding rule for that packet (e.g., time to forward the packet and output link) and forwards the packet to the corresponding output queue at scheduled time (e.g., Cisco Calalyst Switches [1]). This is further detailed in Section 4.2.2.

Message Release Model: We consider two types of messages. First are messages due to background traffic without deadlines. Second are real-time messages released periodically by real-time tasks running on one end node, which are subsequently transmitted to another node. These messages can be classified as message flows, where a flow contains all messages released by the same real-time task. Thus, messages in the same flow have the same source and destination nodes, same relative deadline, and are released periodically. We define a set of message flows as $M = (D, T, S)$, where D_m is the relative deadline of the message in flow m, T_m is the period, and S_m is its size. The flow id and the release time of a message are embedded in the message header upon transmission. Nodes use the flow id to look up the forwarding policy for that flow from their routing table and use the release time to calculate the exact forwarding time for a specific message in that flow.

Objectives: Given the configuration of real-time flows, our static routing algorithm shall derive a forwarding path for each flow, such that (1) the transmission time of a real-time message shall never exceed its relative deadline, (2) the aggregate size of real-time messages on any network device shall never exceed the buffer size of that device, (3) when multiple real-time messages share a network device on their paths, the network device shall have the computing capability to process them before their local deadlines. To achieve these goals, the static routing algorithm derives offline the expected response time R by which each node must have processed a real-time message. The goal for the scheduler is to guarantee end-to-end message deadlines by enforcing the expected response time on each node and considering the runtime response time variance δ caused by the interference due to scheduling and queueing at the output interface. Section 4.2.3 proves that the scheduler can adopt an EDF-based algorithm that uses R as local deadlines to achieve this goal.

Message Delay Model:

We focus on real-time messages to be transmitted before their deadlines. Thus, given any real-time message flow, the objective is to find a forwarding path for which the end-to-end delay does not exceed the relative deadline for the flow. More formally, let the forwarding path for a flow be $(v_0, v_1, ..., v_k)$, where k is the path length and $v_i \in V(0 \le i \le k)$ are the nodes on the forwarding path. Let $t[v_i]$ be the time when the message arrives at node v_i. $t[v_0]$ is the message release time, which is determined by the property of the corresponding message flow. $t[v_i]$, $0 < i \le k$, can be formalized as a function of the path and the underlying network as shown in Eq. 4.1, where $R[v_i]$ is the expected message response time on node v_i and $\delta[v_i]$ is the runtime variance for the message response time (i.e., node v_i starts to transmit the message at time $t[v_i] + R[v_i]$ and finishes the transmission at time $t[v_i] + R[v_i] + \delta[v_i]$). The first sum in Eq. 4.1 is the accumulated propagation delay on links from v_0 to v_i. The second sum is the accumulated message response time on nodes v_0 to v_{i-1}. The subscript m for the message flow is abbreviated when the equations are suitable for every message flow (e.g., $R[v_j]$ instead of $R_m[v_j]$).

$$\begin{aligned} t[v_i] &= t[v_{i-1}] + R[v_{i-1}] + \delta[v_{i-1}] + Pr[v_{i-1}, v_i] \\ &= t[v_0] + \sum_{j=0}^{i-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{i-1} (R[v_j] + \delta[v_j]). \end{aligned} \tag{4.1}$$

Let $A[v_i]$ be the worst case (i.e., the latest time) that node v_i has to start the transmission.

On the first node, $A[v_0] = t[v_0] + R[v_0]$. We assume that $\delta[v_i] \geq 0$ on any node v_i. $\delta[v_i]$ can be bounded by a value $\Delta[v_i]$, which is determined by all message flows on node v_i and is the same for any individual message on this node. Then, the latest transmission time $A[v_i]$ $(0 < i \leq k)$ occurs when the message transmissions on all nodes before v_i have experienced the worst variation, which is expressed in Eq. 4.2. Thus, $A[v_i]$ can be calculated for a message once R and Δ are derived. Sections 4.2.2 and 4.2.3 detail the approach to derive R and Δ offline.

$$\begin{aligned} A[v_i] &= max\{t[v_i] + R[v_i]\} = max\{t[v_i]\} + R[v_i] \\ &= t[v_0] + \sum_{j=0}^{i-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{i} R[v_j] + \sum_{j=0}^{i-1} \Delta[v_j]. \end{aligned} \tag{4.2}$$

As a result, the static routing algorithm needs to find a forwarding path for every message flow so that the end-to-end delay in the worst case is bounded by its relative deadline as expressed in Eq. 4.3.

$$A[v_k] + \Delta[v_k] - t[v_0] \leq D. \tag{4.3}$$

4.2.2 Message Scheduler

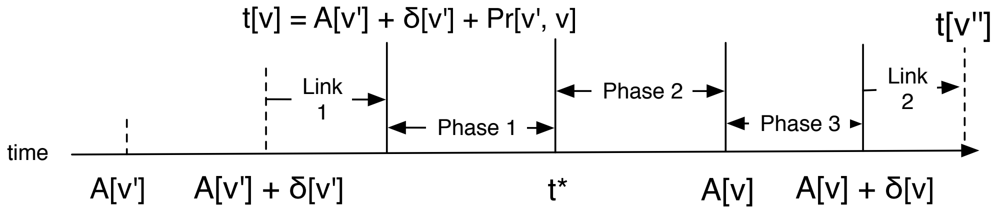

Figure 4.1: Message Phases on Node

A real-time message experiences three phases on one node. Fig. 4.1 illustrates these phases on node v, where the message is sent by node v' and received by node v, and then forwarded to node v''. Before the message is sent to node v, the processor on node v' has calculated the latest message transmission time for the next node (i.e., $A[v]$) from the forwarding table. In the first phase, the message arrives from link 1 at time $t[v]$ and is put in the input buffer. We assume that the time for an interface to put a packet into the input buffer can be ignored. In the second phase, the message is processed and moved to the intermediate queue. During processing, the latest message transmission time for the next node (i.e., $A[v'']$) is calculated and the output interface for the message is determined according to the forwarding table. The message stays in the intermediate queue during the second phase. In the third phase, the message is forwarded to the corresponding interface at time $A[v]$ and then finally transmitted via link 2 at time $A[v] + \delta[v]$. The intuition of this design is to constrain the message arrival time at each node v to a time interval $(A[v'] + Pr[v', v], A[v'] + Pr[v', v] + \Delta[v'])$. In contrast, a background traffic packet is moved from the input buffer to the corresponding interface directly when no real-time messages need to be processed.

To guarantee that the processor can start the transmission in the third phase at time $A[v]$, the processor has to finish processing the message before $A[v]$ in the second phase (at time t^* in Fig. 4.1). Thus, our packet scheduler adopts an EDF-based algorithm to process the message before its local relative deadline $A[v]-t[v]$. Since $A[v] = A[v']+Pr[v', v]+\Delta[v']+R[v]$, which is derived from Eq. 4.2, the local relative deadline $A[v]-t[v]$ is no less than the expected response time $R[v]$ as derived in Eq. 4.4. If the processor has the computation capability to process every real-time message before its expected response time $R[v]$, the processor can forward it on time.

$$\begin{aligned} A[v] - t[v] &= (A[v'] + Pr[v', v] + \Delta[v'] + R[v]) - t[v] \\ &= (A[v'] + Pr[v', v] + \Delta[v'] + R[v]) \\ &\quad - (A[v'] + Pr[v', v] + \delta[v']) \\ &= R[v] + \Delta[v'] - \delta[v'] \\ &\geq R[v]. \end{aligned} \tag{4.4}$$

Algorithm 5 depicts the packet scheduler. The input buffer is organized into a real-time message section and a background traffic section. Each section stores the front and tail side of the corresponding queue. These sections are stored in a circular fashion. Thus, the growing of one side of a section can reach the other side of the other section. An incoming real-time message is stored on the side with sufficient space. If neither side has sufficient space, background packets at the tail of the background section are dropped until the space is sufficient for the real-time message. An incoming background packet is dropped immediately if the input buffer has insufficient space. Otherwise, if the background traffic section has insufficient space at its tail, real-time messages are moved from the head to the tail side of the real-time section. An incoming background packet is always stored at the tail of the background traffic section to support an FCFS service. The scheduler scans the real-time messages in the input buffer and processes the message with the smallest $A[v]$ but $t_{current} \geq A[v] - R[v]$. The second condition is to guarantee that a message will not be processed before its latest release time (see lines 4 to 25). In the algorithm, $\bar{A}$ represents the earliest transmission time of all real-time messages in the intermediate queue. The scheduler compares $\bar{A}$ with the current time to determine whether the transmission time of any message has passed. The scheduler transmits all messages whose transmission times have passed (see lines 26 to 30).

```
**Message Scheduler** *(node v)*
    **Input**: Packet Input Buffer Q
    $\bar{A} \leftarrow \infty$
    **while** *true* **do**
        $t_{current} \leftarrow$ current time
        target real message $x \leftarrow nil$
        target background packet $y \leftarrow nil$
        **for** *each packet m in Q* **do**
            **if** *m is a real-time message* **then**
                **if** $t_{current} \geq A_m[v] - R_m[v]$ *and* ($x = nil$ *or* $A_m[v] < A_x[v]$) **then**
                    $x \leftarrow m$
                **end**
            **else if** $y = nil$ **then**
                $y \leftarrow m$
            **end**
        **end**
        **if** $x \neq nil$ **then**
            **if** $\bar{A} > A_x[v]$ **then**
                $\bar{A} \leftarrow A_x[v]$
            **end**
            $v' \leftarrow$ next node for $x$
            calculate latest transmission time for next node $A_x[v']$.
            move message $x$ into intermediate queue.
        **else if** $y \neq nil$ **then**
            move $y$ into output queue
        **end**
        $t_{current} \leftarrow$ current time
        **while** $t_{current} >= \bar{A}$ **do**
            forward message with transmission time $\bar{A}$ to output interface.
            update $\bar{A}$.
        **end**
        $T_s \leftarrow \bar{A} - t_{current}$
        **if** *Q is empty* **then**
            sleep ($T_s$) or wake up by packet arrival interrupt.
        **end**
    **end**
```

Algorithm 5: Pseudocode for message scheduler

Our scheduler belongs to the class of non-preemptive EDF scheduling algorithms. The real-time messages can be considered as jobs of periodic tasks (i.e., real-time message flows) and instructions (lines 16 to 30) can be considered as the non-preemptive execution of the jobs. Since the expected response time R is not required to be equal to the message flow period, we utilize existing processor-demand analysis [3, 22] to perform a schedulability test for the message flows on each node to determine whether the real-time messages can be transmitted on time with the corresponding expected response time. This schedulability test is adopted by the static routing algorithm when deriving message forwarding paths in Section 4.2.3.

The runtime response time variation, δ, for a particular message consists of two parts. One part is the time to execute at most one iteration of the while loop (lines 4 to 25). The other part is the time to transmit other messages that have earlier transmission times in the intermediate queue than the transmission time of that particular message (lines 26 to 30). Thus, the worst case variation, Δ, is expressed by Eq. 4.5, where c is the worst case execution time in the first part, $B[v]$ is the buffer size of node v, and $E[v, v']$ is the bandwidth of the link that transmits any real-time messages from node v to node v'. The second part in Eq. 4.5 represents the time for the data to be transmitted on node v via the slowest link.

$$\Delta[v] = c + \frac{B[v]}{min\{E[v, v']\}}, \qquad (4.5)$$

for all nodes v' linked to v for real-time messages.

4.2.3 Routing Algorithm

Two significant differences between the objectives of our routing algorithm and other routing algorithms (see Section 4.5) are: (1) Our algorithm finds forwarding paths for multiple pairs of source and destination, and (2) it bounds the end-to-end delay for every pair by a predefined value (relative deadline of the flow) instead of minimizing the delay for a particular flow. Thus, we want to increase the predictability of the node behavior so that the message response time on each node is controlled and no real-time messages will be dropped.

Let us first describe the validation algorithm that checks if a set of real-time message flows with their corresponding paths is schedulable. Given a set of flows, a set of corresponding paths (one path for each flow) is schedulable if the following criteria can be met on every node in the network: (1) When a real-time message arrives at node v, the node is able to process it before time $A[v]$ (i.e., latest message response time). In addition, the node is able to transmit the message at a time in the range $(A[v], A[v] + \Delta[v])$. This is to guarantee that the message arrival time at the next node on the path can be modeled by Eq. 4.1 since this criterion infers $0 \leq \delta[v] \leq \Delta[v]$, which is the assumption for Eq. 4.1. This criterion is checked for the given message flows and the corresponding paths by the schedulability test described in Section 4.2.2.

(2) The residual buffer size of every node (i.e., the node buffer size minus the aggregate size of real-time messages that are resident on that node) cannot be negative at any time. Let us first consider the worst case for one message flow m on any node v. The longest time that any message in the flow is queued on this node is $\Delta[v'] + R_m[v] + \Delta[v]$, where v' is the predecessor of node v on the path. Let $\Delta[v'] = 0$ if v is the first node on the path. Thus, the number of messages of this flow on node v in the worst case is $\lceil \frac{\Delta[v']+R_m[v]+\Delta[v]}{T_m} \rceil$. Then, the aggregate size of all message flows in the worst case must not exceed the buffer size of node v as expressed in Eq. 4.6. This buffer size limitation has to be validated on every node.

$$\sum_{\textit{all flows } m \textit{ on } v} \lceil \frac{\Delta[v'] + R_m[v] + \Delta[v]}{T_m} \rceil * S_m \leq B[v] \tag{4.6}$$

With this validation algorithm, our routing algorithm derives the forwarding paths for message flows with the following backtracking procedure. Assuming a schedulable forwarding path set is found for the first i flows, the routing algorithm checks every path candidate for flow $i+1$ until it finds a candidate that will result in a schedulable path set for all first $i+1$ flows according to the validation algorithm. If no such forwarding path is found for flow $i+1$, the algorithm backtracks to flow i and continues the process with the next candidate for flow i. A forwarding path candidate for a message flow is defined as a sequence of nodes that can transmit this flow from its source node to its destination with the expected response time on each node (i.e., $(v_0, R[v_0]), (v_1, R[v_1]), ..., (v_k, R[v_k])$). Section 4.2.4 describes the algorithm to find path candidates for message flows.

This algorithm always finds a schedulable forwarding path for each message flow if such a path exists since it checks all permutations. The complexity of this backtracking algorithm is exponential with the number of message flows (i.e., the same as an ILP algorithm) but this has no impact on real-time messaging as the calculations are performed offline. Nonetheless, we reduce the actual search time by optimizing the path search order. When checking the path candidates for flow $i+1$, the intuition is to give higher preference to the candidate that results in a larger residual buffer on the path if that candidate is chosen to be the forwarding path for flow $i+1$. The residual buffer size of a path is defined as the minimum residual buffer size of all nodes on the path. Since the residual buffer size of a node determines the size of background traffic that can be kept on the node, this strategy decreases the chance that certain nodes become the bottleneck for the background traffic. If two candidates result in the same residual buffer size, we give higher preference to the candidate with a shorter length. Furthermore, we give higher preference to the candidate with a shorter end-to-end delay if two candidates have the same length. The backtracking algorithm checks candidate paths for flow $i+1$ from highest preference to lowest.

4.2.4 Forwarding Path Candidate

Our routing algorithm requires that forwarding path candidates for message flow i are known when the algorithm attempts to find the path for that flow. A forwarding path candidate includes the nodes on the path and the expected message response time R on each node. First, we perform a breadth-first-search algorithm on the network graph to find all acyclic paths that connect the source and destination of the message flow. Then, for each path, we need to assign the expected message response time to each node based on the following three considerations.

First, the buffer size restriction described in Eq. 4.6 must be met on every switch. Second, the overall end-to-end delay in the worst case based on the expected message response time assignment must not exceed the relative deadline of the message flow (as shown in Eq. 4.3), which can also be expressed as:

$$\begin{gathered}\sum_{j=0}^{k-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{k} R[v_j] + \sum_{j=0}^{k} \Delta[v_j] \leq D. \\ Thus, \sum_{j=0}^{k} R[v_j] \leq D - \sum_{j=0}^{k-1} Pr[v_j, v_{j+1}] - \sum_{j=0}^{k} \Delta[v_j].\end{gathered} \tag{4.7}$$

Third, the assigned expected response time must be long enough on every node v_j on the path so that these messages can be processed with their local relative deadline $R[v_j]$ as required by the validation algorithm. Since the message scheduler is non-preemptive, we use the worst case execution time for the node to process one message (i.e., notation c as described in Section 4.2.2) as the unit to calculate the expected response time. Thus, $R[v_j] = c * x$, where x is at least 1 and upper bounded by Eq. 4.6 and 4.7. The assignment algorithm enumerates all values x in its range and performs processor-demand analysis for EDF scheduling to check if the corresponding response time assignment for the chosen x can be met on the network devices. As a result, multiple forwarding path candidates can be generated for each acyclic path found by the breadth-first-search algorithm.

4.3 Implementation

To support the three-phases message transmission (see Fig. 4.1), this section first presents the message structure and the extension to the Linux network stack on end nodes to specify the flow id and release time of real-time messages. It then presents the structure of forwarding tables on network devices and a scheduler implementation on virtual switches based on Open vSwitch [44, 9], a software-defined network (SDN) simulation infrastructure.

4.3.1 Message Structure and Construction

To support the scheduler in Algorithm 5, real-time messages need to carry the flow id, the latest transmission time for the current node $A[v]$, and the latest transmission time for the next node $A[v_{next}]$. We utilize the *Type of Service (ToS) field (i.e., bits 8-15)* and *Options field* in the standard IP Header to store them as illustrated in Fig. 4.2. For real-time messages, *bit* 8 is set to 1 and *bits* $9 - 15$ are set to the message flow id. Then, a 16-bytes option (*bytes* $20 - 35$) is used to carry the two time values in milliseconds. The type of the option (*byte* 20) is set to a value that has not been used by other protocols to prevent conflicts. For non real-time traffic, *bit* 8 is set to 0, which is the default value for the ToS field.

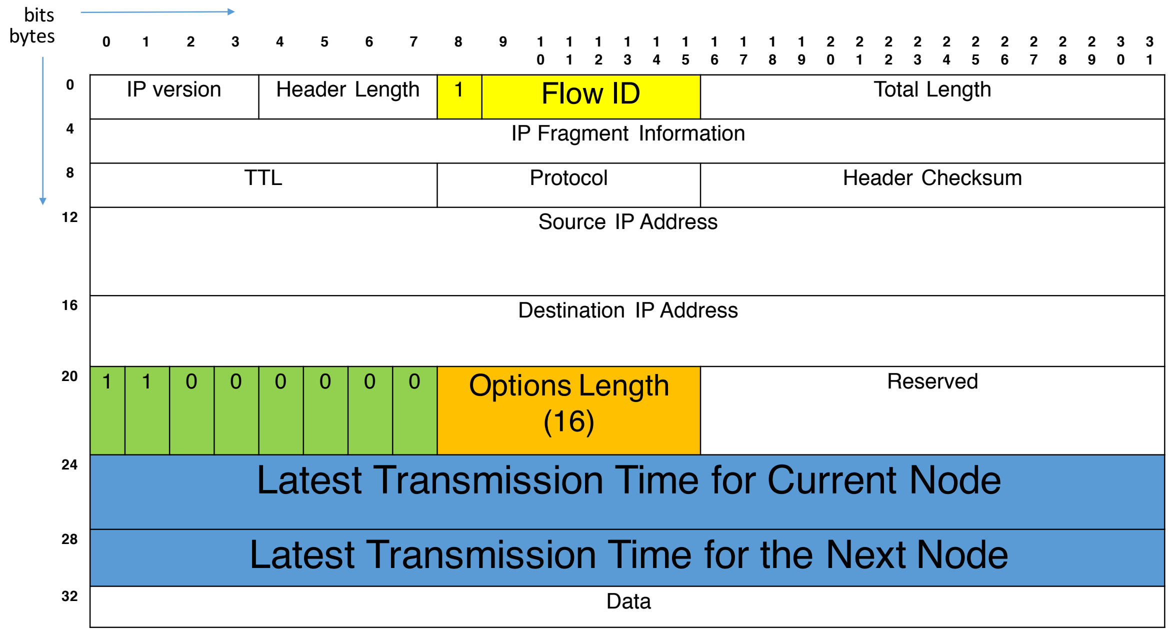

Figure 4.2: Message Header

When a task running in user mode attempts to transmit a real-time message, it needs to specify the flow id to the network stack of Linux. The network stack software of the node searches in the forwarding table to calculate the transmission times and determines the network interface for the transmission. Below are the most significant changes we made to Linux to support this functionality.

(1) We added a new field, fid, of type *char* in the kernel data structure *sock* so that the flow id of a socket provided by the task can be stored.

(2) We extended function *sock_setsockopt* of the Linux kernel and added a new option *SO_FLOW* so that the system call *setsockopt* assigns the message flow id when the task attempts to transmit a real-time message. *sock_setsockopt* keeps the value of the flow id in the fid field of the *sock* structure. The fid field is initialized to 0 when the *sock* structure is created.

(3) When the application transmits the real-time message, the kernel creates instance(s) of the sk_buff structure to store the data of the message. We added a new field, fid, in sk_buff so that the flow id of the message can be copied and passed down to the network layer. In addition, we added another field, *release_time* of type *ktime_t* in sk_buff, to store the current system time as the message release time.

(4) When the network header structure *network_header* in sk_buff is constructed in the network layer, we use fid to search in the forwarding table (see Section 4.3.2) and calculate the latest transmission time for the current node and for the next node. Then, fid and the transmission times are stored in the network header as shown in Fig. 4.2. *Bit* 8 is set to 1 to indicate real-time messages. If fid is 0, *bit* 8 is set to 0.

(5) We implemented a delay queue, which provides the standard interfaces of the Linux traffic control queue disciplines [20], so that real-time messages can be transmitted at its latest transmission time as required by our routing algorithm. The delay queue contains two components. The first one is a linked-list based FIFO queue to store non real-time packets. For real-time packets, we utilize the *cb* field, the control buffer in sk_buff, to implement a linked list-based min-heap. This linked list-based implementation does not have a limit on the number of messages that can be queued in the min-heap, which an array-based implementation of min-heap would have.

(6) When the clock reaches the latest transmission time of a real-time message, the queue discipline moves the value of the latest transmission time for the next node (i.e., *bytes* $28 - 31$) to *bytes* $24 - 27$. Then, the message is forwarded to the network interface for transmission. In this way, when the message arrives at the next node, *bytes* $24 - 27$ denote the latest transmission time for the new node.

4.3.2 Forwarding Table Structure

We utilize the default forwarding table for background traffic. The default forwarding table contains the mapping between the destination network IDs (i.e., network destination and network mask) and the local interfaces that reach the destinations. For real-time messages, we use the flow IDs to indicate the destinations in the forwarding table. The forwarding table contains the expected message response time ($R[v]$) on the current node v. In addition, it contains the sum of the worst case message transmission variation $\Delta[v]$ on the current node, the propagation delay ($Pr[v, v']$) on the link to the next node v', and the expected message response time ($R[v']$) on the next node. Table 4.2 depicts the table structure, where *Next Aggregate Delay* is the sum of $\Delta[v]$, $Pr[v, v']$, and $R[v']$. $Interface$ is the interface to forward messages in each flow.

Table 4.2: Forwarding Table

fid	Expected Response Time (ms)	Next Aggregate Delay (ms)	Interface
1	7	10	eth0
2	12	16	eth1

As a result, the time variables of real-time messages required by the scheduler (see Algorithm 5) can be calculated from the data carried in the header and stored in the forwarding table. Table. 4.3 depicts these relations.

Table 4.3: Message Scheduler Variables

Variable	**Source**
$A[v]$	Message header (*bytes* $24 - 27$)
$R[v]$	Forwarding table
$A[v']$	$A[v]$ + *Next Aggregate Delay* in forwarding table

4.3.3 Virtual Switch Implementation

We extend the traffic control [20] and the OpenFlow implementation in Open vSwitch [44, 9] on Linux to implement our message scheduler. Our virtual switch implementation includes three components: (1) An ingress queue based on the traffic control mechanism (i.e., the input queue in our node model). When this queue is enabled, the incoming packets on related interfaces are put into this queue. Meanwhile, the packet arrival events trigger the scheduler to invoke the *dequeue* function if the scheduler is idle. The *dequeue* function finds the packet with the earliest expected response time and sends it to the downstream OpenFlow actions (lines 4 to 15 of Algorithm 5). (2) A new flow table structure (see Section 4.3.2) is constructed for real-time messages based on the OpenFlow hierarchy and the corresponding forwarding actions to execute (in lines 16 to 25). The scheduler invokes the forwarding actions for every packet. It then forwards real-time messages to the downstream intermediate queue and forwards background traffic to the corresponding interface. (3) A delay queue based on the traffic control mechanism (i.e., the intermediate queue in our node model). This queue has the same structure as the delay queue at the end nodes (see Section 4.3.1). The scheduler forwards any real-time messages removed from the delay queue with the *dequeue* function for the corresponding interface.

The buffer to store packets on a virtual switch is shared by the ingress queue and the delay queue. When an interface attempts to put a background packet into the ingress queue by invoking the *enqueue* function, we check if the available buffer is sufficient for the packet. If it is not, that background packet is dropped. When an interface attempts to put a real-time message into the ingress queue and the available buffer is insufficient for that message, the *enqueue* function drops background packets that are already in the queue so that the real-time message can be stored. In addition, as one of the objectives of our routing algorithm (see the second criterion of the validation algorithm described in Section 4.2.3), the algorithm guarantees that the buffer always has enough space to store real-time messages if the forwarding path is schedulable. The size of the buffer is a configurable parameter in our implementation, which is set to different values as part of our experimental assessment.

4.4 Evaluation

We evaluate our virtual switch on a local cluster. The experiments are conducted on 6 nodes, each of which features a 2-way SMP with AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node). Each node has 32GB DRAM and Gigabit Ethernet. These nodes are connected via a single network switch in the physical network topology, and we use different numbers of nodes as virtual SDN switch nodes in different experiments. Each node runs a modified version of Linux 2.6.32 and Open vSwitch 2.3.2, which includes the virtual switch implementation. The clocks on these nodes are synchronized with a centralized NTP server when we conduct the experiments. In the first part of this section, we present the experimental results to demonstrate the capabilities of the message scheduler in two aspects: (1) Under intense network congestion, the message scheduler on a single node can meet the hard deadline of every real-time message. (2) When the forwarding paths of real-time messages consist of multiple switches, our message schedulers on these switches can collaborate to guarantee end-to-end deadlines of all real-time messages. In the second part of this section, we present a demonstration for the routing algorithms.

4.4.1 Single Virtual SDN Switch Result

Background: To demonstrate the effectiveness of the packet scheduler, we measures the deadline miss rate for real-time messages and packet drop rate for background traffic on a single virtual SDN switch with different configurations.

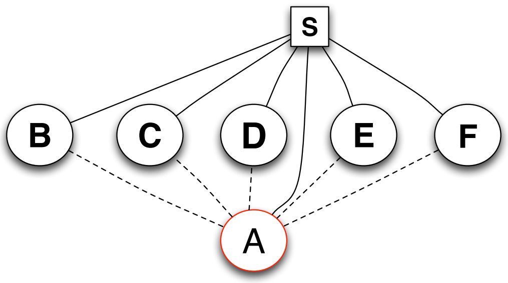

Figure 4.3: Experiment Network Setup (1)

All 6 nodes (marked as A to F in Fig. 4.3) are connected via a single physical switch (marked as S; physical links are indicated by solid lines). We use node A as the virtual switch. The traffic is transmitted through node A via the virtual links (dashed lines). To support this, a node generates test packets using the IP address of node A as the destination. When a test packet arrives at virtual switch A via the physical links, A uses the flow id carried in the packet header to determine its real destination. Thus, we add an extra column in the forwarding table to indicate the real destination of a message flow. In addition, the ToS field in background packets generated for test purposes is used to indicate their real destination. Virtual switch A fills the real destination of any packet into the IP header of the packet and then forwards it via the physical link to the central switch. In this way, test packets are transmitted to its destination via the virtual switch. This modification is due to the limitation of our experimental environment, which would be unnecessary if the virtual switch were deployed directly onto a physical switch.

We use the network benchmark Sockperf to generate test workloads. A Sockperf client transmits messages as UDP packets to the corresponding Sockperf server via the virtual switch. The client generates a log record to indicate the transmission time of a message while the server generates a log record to indicate the message arrival time. To simplify measurements, the data section for a real-time message contains the flow id and the sequence id of a particular message in that flow. The data section is padded by *0*s so that its total size is 1000 bytes. A packet has been dropped by the virtual switch if no corresponding record exists on the server side after the experiment. We assume that no packet drop occurs in the network stack software of the end nodes or on the physical network links. Our platform meets this assumption since we do not limit the buffer size on end nodes and the physical network is reliable in our cluster.

Table 4.4 depicts the workload in the first experiment. The workload is controlled by Sockperf parameters $(mps, burst)$. mps indicates the number of frames per second, which affects the frame size. $burst$ defines the number of packets transmitted in each frame by the client. For example, $(250, 1)$ indicates 250 frames per second (i.e., a frame size of $4ms$) with 1 message transmitted per frame. Real-time message flows have fixed parameters to make them strictly periodic. The actual burst of a background flow is a random value uniformly drawn from the interval $[\frac{burst}{2}, burst]$ in each frame. If a real-time message does not arrive at the server (i.e., no corresponding record exists in the server logs), we consider it a deadline miss in this experiment. No re-transmission is performed. In addition, we calculate the end-to-end delay for real-time messages even if they arrive at the server side. If the delay of a message is longer than its deadline, we also consider it as a deadline miss. We set the expected response time on the virtual switch to $20ms$ for both real-time flows, which is the relative deadline of the message flows ($24ms$) minus the aggregate propagation delay and expected response time variations on the path (estimated as $4ms$). The case study in Section 4.4.3 demonstrates this calculation.

Table 4.4: Test Workload in Single Switch Experiment

fid	**Type**	**Source**	**Destination**	**(mps, burst)**	**Relative Deadline**
1	Real Time	B	C	(250, 1)	24ms
2	Real Time	C	D	(200, 1)	24ms
3	Background	D	E	(200, 400)	NA
4	Background	E	F	(200, 400)	NA
5	Background	F	D	(200, 400)	NA

Analysis: Fig. 4.4 depicts the result. We adjust the buffer size and compare both deadline miss rate and packet drop rate when the message scheduler is turned on and off on the virtual switch. When the message scheduler is turned off, the virtual switch does not differentiate real-time packets from background packets and forwards them to the destination directly. The x-axis in Fig. 4.4 is the buffer size on the virtual switch. The y-axis is the packet drop rate for background traffic and the deadline miss rate for real-time flows.

The black and green lines depict the deadline miss rate for real-time messages when the scheduler is turned off and on, respectively. With $20ms$ as the expected response time, the message flows are schedulable under all buffer size configurations. Both message flows have a 0% deadline miss rate when the scheduler is on.

Observation 1: The packet scheduler can meet the deadline requirements of all real-time messages.

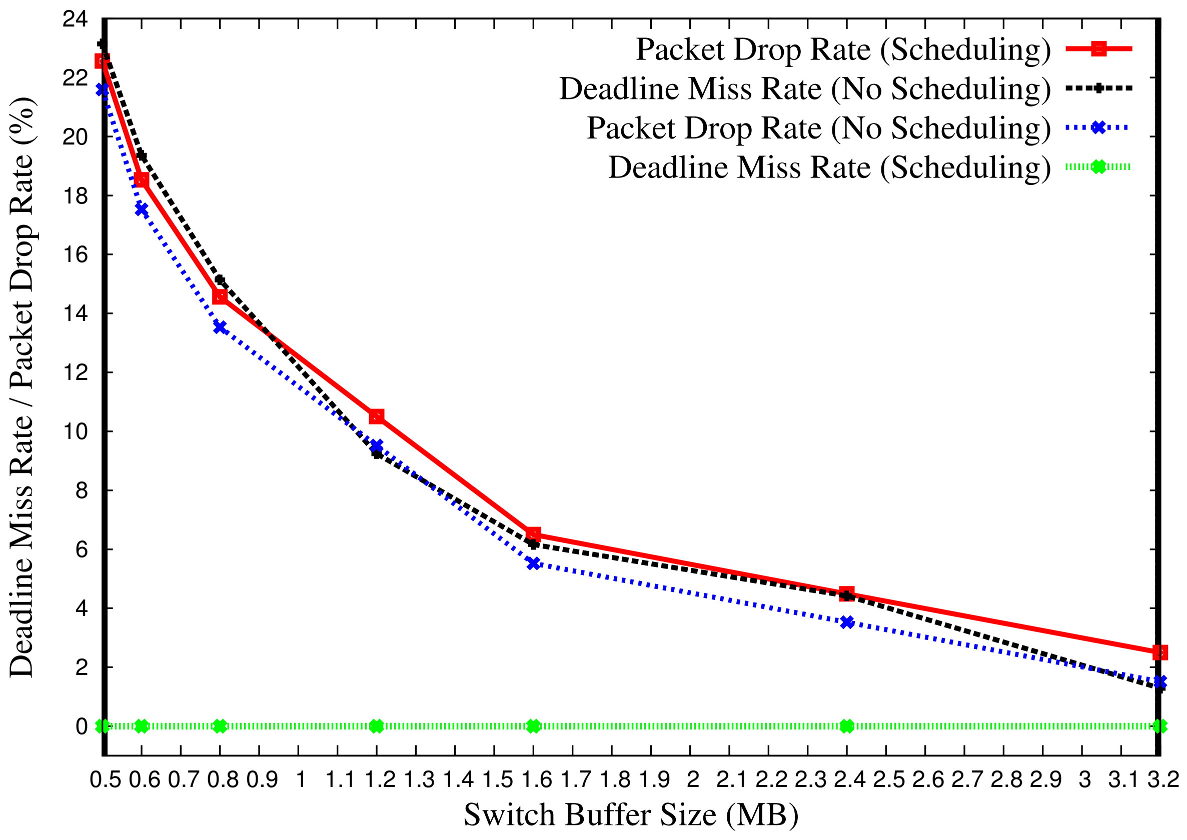

Figure 4.4: Deadline Miss Rate and Message Drop Rate

The black and blue lines depict the deadline miss rate and packet drop rate when the scheduler is turned off. Since real-time packets and background packets are processed in the same way by the switch, the deadline miss rate and packet drop rate are close for all buffer size configurations. The virtual switch has an increasing tolerance to the burstiness of packets with a larger buffer size. As a result, both rates decrease when the buffer size is increased as depicted by the declining lines.

Observation 2: When the scheduler is off, a larger buffer size can decrease both the deadline miss rate for real-time messages and the packet drop rate for background traffic.

When the scheduler is on, the expected message response time on the virtual switch is $20ms$, which means every real-time message is delayed in the switch buffer for $20ms$ (no delay is added in the end nodes where the real-time messages are released and received). Due to this delay, the available buffer size for background traffic shrinks. As a result, the packet drop rate for background traffic increases slightly, e.g., from 17.53% (blue line) to 18.53% (red line) for a buffer size of $0.6MB$. Table 4.5 shows an overall trend of an increasing drop rate for background traffic when the expected response time for real-time messages is increased from $10ms$ to $320ms$ for a buffer size of $1.2MB$. The results indicate that the longer real-time messages are delayed in the switch buffer, the smaller the available buffer is for background traffic. As a result, the packet drop rate increases when the delay time for real-time messages is increased.

Observation 3: Our scheduler increases the packet drop rate for background traffic.

Table 4.5: Expected Response Time vs Message Drop Rate

Expected Response Time (ms)	**Message Drop Rate (%)**
10	10.80
20	10.89
40	11.05
80	11.46
160	11.81
320	12.33

The measurement of end-to-end delays for real-time messages indicates that the message scheduler does not introduce a significant variation to the message transmission time (Δ is small). The end-to-end delay includes the latency due to transmission from the source node to the virtual switch, the time the message spent on the virtual switch, and the transmission latency from the virtual switch to the destination node. When we set the buffer size to $1.2MB$ and the expected response time to $10ms$, the shortest end-to-end delay is $10.9ms$ and the longest is $12.1ms$ (the median value is $11.6ms$), of which $10ms$ are due to the software-induced delay on the virtual switch. Since messages are transmitted via a physical switch (and not via direct links in our system model), the physical switch also contributes to the delay variation.

4.4.2 Multiple Virtual Switches Result

Background: In the second experiment, we construct a virtual switch chain to transmit real-time messages. Nodes $A - D$ are configured as virtual switches with a buffer size of $1.2MB$. Node E uses a Sockperf client to transmit four message flows to the Sockperf server on node F. Fig. 4.5 depicts the forwarding path for each message flow. The periods of message flows are $1ms$. The expected response time is $5ms$ and the next aggregate delay is $7ms$ for each flow on these virtual switches. This setup suggests that the sum of the worst case response time variation, Δ, and the propagation delay, Pr, on the link between two virtual switches (i.e., two physical links connected via the central physical switch) is $2ms$. We measure the end-to-end delay for each message.

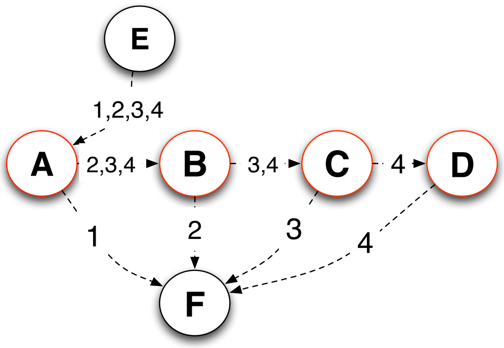

Figure 4.5: Experiment Network Setup (2)

Our experimental results show that messages in all four flows experience a similar end-to-end delay variation ($[0.8ms, 1.8ms]$). The value of next aggregate delay includes the worst case response time variance on the previous node (as described in Eq. 4.3.2). The scheduler only considers a real-time message once the current time is greater or equal to the worst case response time (see line 9 in Algorithm 5). As a result, only the response time variation on the last virtual switch (plus the propagation delay variation on the physical link from the last virtual switch to the receiver in our experiment) contributes to the measured variation of the end-to-end delays even though the forwarding path has multiple virtual switches. This indicates the effectiveness of the message scheduler in terms of variance control.

Observation 4: Real-time messages do not accumulate response time variation (jitter) when forwarded by a chain of virtual switches.

4.4.3 Routing Algorithm Demonstration

We demonstrate our routing algorithm to find forwarding paths for real-time message flows. Three message flows with attributes indicated in Table 4.6 are considered. These flows can be either forwarded via intermediate switch B or switches C and D as depicted in Fig. 4.6, where the numbers in the switch circles are the buffer size of that switch. For simplicity, we consider that it takes $1ms$ for the message scheduler on every switch to process one message (i.e., $c = 1ms$ in Eq. 4.5). We consider a response time variation of $2ms$ on every node or switch and a propagation delay of $1ms$ ($Pr = 1ms$) on every link (i.e., $\Delta = 2ms$, $Pr = 1ms$).

We use the same notation as described in Section 4.2.3 to express forwarding path can-

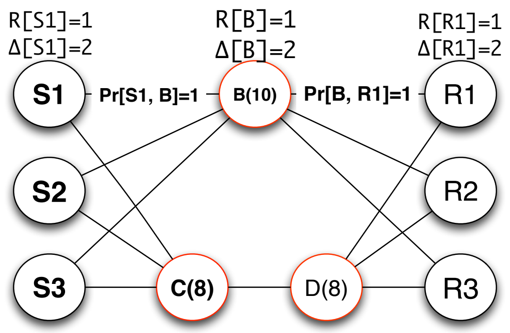

Figure 4.6: Routing Algorithm Demonstration

Table 4.6: Real-time Message Flows in Demonstration

fid	**Source**	**Destination**	**Period (T)**	**Relative Deadline (D)**	**Message Size (S)**
1	S1	R1	12	11	1
2	S2	R2	1	15	1
3	S3	R3	12	12	9

didates. For example, candidate ((S1, 1), (B, 1), (R1, 1)) for flow 1 means that a real-time message of flow 1 is forwarded via nodes $S1$, B, and $R1$ with expected message response times of $1ms$, $1ms$, and $1ms$ on each node. We determine the schedulable routing paths for this setup in the following steps.

(1) Consider forwarding path candidate ((S1, 1), (B, 1), (R1, 1)) for flow 1 (see 1st row in of Table 4.7). The worst case end-to-end delay of this path candidate is $R[S1] + \Delta[S1] + Pr[S1, B] + R[B] + \Delta[B] + Pr[B, R1] + R[R1] + \Delta[R1] = 11ms$. Thus, this is the only forwarding path candidate for flow 1 according to the deadline constraint of Eq. 4.7. If the expected response time on any node of the path were increased, the end-to-end delay would exceed the relative deadline of flow 1, which is $11ms$.

The size of messages in flow 1 is 1. The maximum number of messages in flow 1 that could be on switch B at any time is $\lceil \frac{\Delta[S1]+R[B]+\Delta[B]}{T_1} \rceil = 1$. As a result, the residual buffer size of node B becomes 9, which is the buffer size of node B (10) minus the maximum buffer that could be occupied by messages in flow 1.

(2) The forwarding path candidates for flow 2 are shown with ID 2-6 in Table 4.7, where columns B, C, and D depict the residual buffer size on the corresponding nodes if flow 2 is forwarded. Candidates 2, 3, and 4 are considered first since the residual buffer size on these paths is 4 (i.e., $B - \lceil \frac{\Delta[S2]+R[B]+\Delta[B]}{T_2} \rceil * S_2 = 9 - 5 = 4$) if flow 2 is scheduled, which is larger than the candidates 5 and 6. However, switch B cannot schedule both flow 1 and flow 2, since the utilization of flow 2 is $\frac{c}{T_2} = 100\%$. Other path candidates for flow 2 with $R[B] = 1ms$,

which are not shown in Table 4.7, are rejected for the same reason.

(3) Candidate 5 is considered next for flow 2 since it is a shorter path compared to candidate 6 even though they have the same residual buffer size on their paths. However, if flow 2 is transmitted via candidate 5, the residual buffer size of node B becomes 3. In this case, flow 3 has no forwarding path candidate since flow 3 requires that the buffer size of any node on the path is at least 9, the size of the message in flow 3, since the other forwarding path (via switches C and D) only has a residual buffer size of 8. Other path candidates for flow 2 with $R[B] \geq 2ms$, which are not shown in Table 4.7, are rejected for the same reason. Thus, the algorithm has to consider candidate 6 in Table 4.7, which is chosen.

Table 4.7: Forwarding Path Candidates

ID	**Flow**	**Forwarding Path Candidate**	**End-to-end Delay**	**Residual Buffer**[1] B	C	D	**Result**
1	1	(S1, 1), (B, 1), (R1, 1)	11	9	8	8	✔
2	2	(S2, 1), (B, 1), (R2, 1)	11	4	8	8	×
3	2	(S2, 2), (B, 1), (R2, 1)	12	4	8	8	×
4	2	(S2, 1), (B, 1), (R2, 2)	12	4	8	8	×
5	2	(S2, 1), (B, 2), (R2, 1)	12	3	8	8	×
6	2	(S2, 1), (C, 1), (D, 1), (R2, 1)	15	9	3	3	✔
7	3	(S3, 1), (B, 1), (R3, 1)	11	0	3	3	✔
8	3	(S3, 1), (B, 2), (R3, 1)	12	0	3	3	×
9	3	(S3, 2), (B, 1), (R3, 1)	12	0	3	3	×
10	3	(S3, 1), (B, 1), (R3, 2)	12	0	3	3	×

[1] After the corresponding flow is scheduled on the path.

(4) Candidates 7-10 for flow 3 meet the buffer size requirement and the relative deadline of flow 3. Candidate 7 has a higher preference since it has a shorter end-to-end delay. In addition, node B can schedule both flow 1 and 3, both with an expected response times of $1ms$. In summary, we found paths for all three flows.

4.5 Related Work

Our packet scheduler adopts per-node traffic control to guarantee transmission deadlines similar to other switched real-time Ethernet mechanisms, e.g., rate-controlled service disciplines (RCS) [17, 64], token-bucket traffic shaping [35], and EDF scheduling [18, 66]. Unlike these mechanisms, our scheduler considers multiple constraints imposed by network devices, namely link speed, computation speed, and buffer capacity. Table 4.8 details the comparison of the

Table 4.8: Assumptions Comparison for Real-time Packet Schedulers

Mechanism	**Buffer Capacity**	**Computation Capability**	**Link Speed**
RCS [17, 64]	infinite	infinite	defined
EDF [18, 66]	infinite	infinite	defined
Traffic shaping [35]	defined	infinite	defined
D^3 [60], D^2TCP [59]	defined	infinite	defined
Our work	defined	defined	defined

assumptions of these mechanisms. $defined$ in the table indicates that the corresponding mechanism considers the restriction in its model. Our work does not assume infinite buffer capacity and computation capacity. Our assumptions are more realistic on commodity switches connected by modern high speed links.

Past work has proposed different mechanisms to establish communication channels that recognize real-time requirements of packet flows at run time [16, 18, 57, 35]. In contrast, our work focuses on forwarding path planning (via a static routing algorithm) and forwarding policy enforcement (via packet scheduling). Other mechanisms guarantee soft deadlines of real-time packets while providing high throughput to other traffic [59], or adopt congestion avoidance algorithms when network contention occurs (e.g., buffer occupancy exceeds a certain threshold) [60]. Our work guarantees hard deadlines by dropping only non-realtime packets upon network congestion.

Our schedulability model does not depend on statistics as metrics of the network, e.g., bandwidth. Such metrics are dependent on multiple primitive factors of the switch: the packet scheduling algorithm, buffer size, computing capability, and the state of other flows on the switch [19]. Instead, we have specifically considered these factors in two ways: (1) We formalize the dynamics of the network delay by introducing response time variations (δ) in our analysis and we aggressively control the variation by the message scheduler. (2) Our routing algorithm considers multiple constraints on switches and links when finding forwarding paths.

In contrast to TDMA-based real-time Ethernet channel implementations [8, 27], which rely on custom hardware to respond to control data frames, our scheduler is designed and implemented on Linux-compatible virtual switches and can be deployed on modern commodity SDN-compatible switches, which is more suitable for wide-area deployment.

Past work has studied routing algorithms to find packet forwarding paths with QoS support in different situations. This includes Dijkstra's algorithm to find the single-source path with shortest end-to-end delay under the assumption that per switch delay is known as priori [21], Suurballe's algorithm to find multiple disjoint paths with minimal total end-to-end delay [58], and methods to find multiple disjoint paths with the minimized delay of the shortest path [61].

Others have studied the routings to find an optimal forwarding path for one particular message flow under certain network assumptions [21]. Past work has also proposed to transmit messages over multiple paths simultaneously while the total time of the flow is constrained under the assumption that actual bandwidths are known [51]. Network Calculus has also been adopted to derive the forwarding path for a particular flow under the additional assumptions that no cross-over traffic exists or the pattern of cross-over traffic is known as priori [6]. Our work differs as follows from these prior work. It derives forwarding paths for multiple message flows to guarantee the hard deadline of every message. Our analysis depends on the link speed matrix, E, as described in Section 4.2. However, when the implementation is deployed on physical switches, E can be quantified by the medium speed, which is independent of the network traffic and scheduling algorithms. In addition, our routing algorithm belongs to the class of constraint-based path selection algorithms, where multiple constraints are considered on the transmission links [29]. We extend that by adding the constraint of switch buffers and formalizing the cost and evaluation functions for switch buffers and message scheduling, which are essential when applying any constraint-based path selection algorithm.

To find forwarding paths for multiple packets, past work has proposed to minimize the average packet delay [21]. However, the objective of real-time message transmission is to meet the deadline of each message flow (i.e., not to minimize the average delay). Our routing algorithm considers the hard deadline of every real-time flow when assigning a forwarding path.

4.6 Conclusion

We have presented a routing algorithm to determine forwarding paths for real-time message flows in a distributed computing environment and a scheduler to actively enforce a message forwarding policy on network devices. Our routing algorithm considers both the deadlines and the network resource demands of real-time messages. As a result, no real-time messages can be dropped due to network contention when handled by our message scheduler and no real-time messages miss their deadlines. We have implemented the scheduler on virtual switches and conducted experiments on a local cluster to prove the effectiveness of the scheduler. Our experimental results showed that deadline misses of real-time messages dropped to 0 when the message scheduler was turned on.

Future work includes porting the message scheduler implementation to physical network devices (e.g., physical switches). Modern SDN-compatible switches support customized packet forwarding protocols (e.g., OpenFlow), which could be utilized to run our virtual switch implementation based on Open vSwitch. Since physical switches have hardware that is dedicated to packet processing, we expect a better performance than the virtual switches. In addition, in the case where a set of real-time message flows cannot be scheduled on a network environment due to hardware limitations, we plan to extend our routing algorithm to produce suggestive information, e.g., the required increase in buffer size of a switch before the set of flows becomes schedulable. Furthermore, we plan to extend the routing algorithm to find disjoint forwarding paths for real-time messages to handle network device failures.

Chapter 5

Conclusion and Future Work

In this chapter, we present the conclusions drawn from our work and the scope of the future work.

5.1 Conclusion

The work presented in this document first detailed our design and implementation for the real-time distributed hash table (DHT) on the basis of the Chord protocol to support a deadline-driven lookup service at upper layer control systems of the North American power grid in Chapter 2. This DHT provides $put(key, value)$ and $get(key)$ API services, which store the given key/value pair and fetch the value associated with a given key in the system. This DHT employs a cyclic executive to schedule periodic and aperiodic lookup jobs for data requests. Furthermore, we formalize the pattern of a lookup workload on our DHT according to the needs of power grid monitoring and control systems, and use queueing theory to analyze the stochastic bounds of response times for requests under that workload. We also derive the quality of service model to measure the probability that the deadlines of requests can be met by our real-time DHT. This problem was motivated by the power grid system but the cyclic executive and the approach of timing analysis generically applies to distributed storage systems when requests follow our pattern. Our evaluation shows that our model is suited to provide an upper bound on response times of real-time tasks.

This provides application level support for the predictability requirements of distributed storage systems. Since the underlying network infrastructure is transparent to our real-time DHT, this implementation can only provide probabilistic deadline guarantees to real-time tasks, which is suitable for soft real-time systems. To support real-time systems with hard deadline requirements, we enhance the predictability of the network transmission by adopting two packet schedulers on respective end-nodes and network devices.

Thus, this work introduces a hybrid EDF packet scheduler in Chapter 3. This packet scheduler runs on end nodes and includes two components. First, at the application layer, the partial-EDF job scheduler utilizes periodic tasks to transmit messages over networks to decrease the impact of application-level interrupt handling on system predictability. To increase the system predictability, we propose a density-based schedulability test on a task set for our partial-EDF job scheduler. Second, at the end node network stack layer, we extend the Linux network stack to support message deadlines and implement an EDF packet scheduler to transmit messages in EDF order. As a result, the packets of urgent tasks can be transmitted first. Futhermore, we propose to utilize Linux traffic control mechanisms to decrease network delay variance, which may increase the predictability of distributed tasks. Our experimental results showed that the EDF packet scheduler can decrease the deadline miss rate and the traffic control mechanism can decrease network delay variance in a local cluster. For example, under a large workload of real-time packets in our experiment, the EDF packet scheduler can drop the deadline miss rate from 38.0% to 15.0%. Since the packets with shorter deadlines are transmitted first, the system has a higher probability to guarantee deadlines for all real-time packets.

The first packet scheduler is considered *passive* since it can only reduce the probability of resource contention on the network instead of preventing contention. As shown in the experimental results, the packet scheduler cannot avoid deadline misses under large workloads. It is only sufficient to realize soft deadlines of distributed real-time tasks. To address this problem, this work details the second packet scheduler in Chapter 4. This packet scheduler runs on network devices to *actively* control the behavior of network devices and avoid network contention for real-time packets. Given the network topology and capacity of devices, this work contributes a novel approach to avoid packet loss and control the queueing delay for real-time packets to guarantee their transmission deadlines.

The active packet scheduling mechanism includes two components. First, the static routing algorithm can derive forwarding paths for real-time message flows in a distributed computing environment. This static routing algorithm considers both the deadline and buffer size requirements of the real-time flows and multiple constraints imposed by network devices such as link speed, computation speed, and buffer capacity. Second, the packet scheduler enforces the message forwarding policy derived by the routing algorithm on network devices. As a result, no real-time messages can be dropped due to network contention when handled by our message scheduler and no real-time messages miss their deadlines.

We provide a schedulability test model to increase the predictability of the system. This schedulability model does not depend on statistics of network metrics such as bandwidth, which are dependent on multiple primitive factors of the switch: the packet scheduling algorithm, switch buffer size, computing capability, and the impact of other flows on the switch. Instead, we have specifically considered these factors in two ways: (1) We formalize the dynamics of the network delay by introducing response time variations in our analysis and we aggressively control the variation via the message scheduler. (2) Our routing algorithm considers multiple constraints on switches and links when finding forwarding paths for real-time flows with hard deadlines. Our evaluation demonstrates the effectiveness of the packet scheduler. Our experimental results show that deadline misses of real-time messages drop to 0 when the message scheduler is turned on for different capacities of network devices.

In summary, we analyze the challenges in distributed real-time systems and provide an end-to-end solution to meet the temporal requirements of distributed tasks in the work. We show that the hypothesis holds, i.e., the deployment of both a real-time task scheduler and a packet scheduler across all layers of the system increases the end-to-end predictability of distributed real-time systems.

5.2 Future Work

This work can be extended in several directions.

1. Our predictability model for the distributed real-time systems requires that the underlying network experiences no failures. When network failures occur, the system cannot guarantee the deadlines for the real-time tasks whose forwarding paths are affected. Thus, we intend to develop a failure recovery mechanism, which dynamically establishes new forwarding paths for the affected tasks. This mechanism must collaborate with the existing packet scheduler to ensure sustained schedulability of real-time packets. In addition, this mechanism needs to prioritize dynamic routing packets according to the deadlines of their tasks. When network resource conflicts occur in the failure recovery, the mechanism must effectively reserve resources for tasks with higher priority.

2. Our implementation of packet schedulers uses virtual platforms instead of physical network devices. Compared to physical devices, the virtual platforms have a deeper software stack, which potentially decreases the efficiency of the packet schedulers. One example is that the non real-time packets in the network have a significantly higher loss rate as shown in the experimental results in Chapter 4. The implementation is compatible to commercial network switches and our experimental results show the trends of predictability improvement for distributed real-time systems. Thus, we intend to deploy our work on physical network devices to show its full potential.

3. Our work decouples distributed real-time tasks into subtasks that are processed on different layers, including the application layer, the end-node network layer, and the shared network layer. We propose separate techniques to increase predictability with the deadline requirements on each layer known priori. However, certain distributed tasks only

have end-to-end deadline requirements. Such loose constraints increase the flexibility of distributed systems. We intend to study the impact of this constraint and derive a solution for the deadline assignment problem, which decomposes an end-to-end deadline into deadlines for each layer.

REFERENCES

[1] Catalyst switch architecture. http://www.cisco.com/networkers/nw03/presos/docs/RST-2011.pdf.

[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256. ACM, 2007.

[3] Karsten Albers and Frank Slomka. An event stream driven approximation for the analysis of real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 187–195. IEEE, 2004.

[4] S. Bak, D.K. Chivukula, O. Adekunle, Mu Sun, M. Caccamo, and Lui Sha. The system-level simplex architecture for improved real-time embedded system safety. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 99–107, 2009.

[5] T.P. Baker and Alan Shaw. The cyclic executive model and Ada. Technical Report 88-04-07, University of Washington, Department of Computer Science, Seattle, Washington, 1988.

[6] Anne Bouillard, Bruno Gaujal, Sébastien Lagrange, and Éric Thierry. Optimal routing for end-to-end guarantees using network calculus. *Performance Evaluation*, 65(11):883–906, 2008.

[7] A. Cahn, J. Hoyos, M. Hulse, and E. Keller. Software-defined energy communication networks: From substation automation to future smart grids. In *IEEE Conf. on Smart Grid Communications*, page (accepted), October 2013.

[8] Gonzalo Carvajal, Luis Araneda, Alejandro Wolf, Miguel Figueroa, and Sebastian Fischmeister. Integrating dynamic-tdma communication channels into cots ethernet networks. *IEEE Transactions on Industrial Informatics*, 12(5):1806–1816, 2016.

[9] Martin Casado, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Rethinking packet forwarding hardware. In *HotNets*, pages 1–6. Citeseer, 2008.

[10] T.L. Crenshaw, E. Gunter, C.L. Robinson, Lui Sha, and P.R. Kumar. The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures. In *IEEE Real-Time Systems Symposium*, pages 400–412, 2007.

[11] Rene L Cruz. A calculus for network delay. i. network elements in isolation. *Information Theory, IEEE Transactions on*, 37(1):114–131, 1991.

[12] J. E. Dagle. Data management issues associated with the august 14, 2003 blackout investigation. In *IEEE PES General Meeting - Panel on Major Grid Blackouts of 2003 in North Akerica and Europe*, 2004.

[13] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhies, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[14] J. D. Decotignie. Ethernet-based real-time and industrial communications. *Proceedings of the IEEE*, 93(6):1102–1117, 2005.

[15] Tullio Facchinetti, Giorgio Buttazzo, Mauro Marinoni, and Giacomo Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 98–105. IEEE, 2005.

[16] Domenico Ferrari and Dinesh C Verma. A scheme for real-time channel establishment in wide-area networks. *Selected Areas in Communications, IEEE Journal on*, 8(3):368–379, 1990.

[17] Leonidas Georgiadis, Roch Guérin, Vinod Peris, and Kumar N Sivarajan. Efficient network qos provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking (TON)*, 4(4):482–501, 1996.

[18] Hoai Hoang, Magnus Jonsson, Anders Kallerdahl, and Ulrik Hagström. Switched real-time ethernet with earliest deadline first scheduling-protocols and traffic handling. *Parallel and Distributed Computing Practices*, 5(1):105–115, 2002.

[19] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125. IEEE, 2008.

[20] Bert Hubert, Thomas Graf, Greg Maxwell, Remco van Mook, Martijn van Oosterhout, P Schroeder, Jasper Spaans, and Pedro Larroy. Linux advanced routing & traffic control. In *Ottawa Linux Symposium*, page 213, 2002.

[21] Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a delay tolerant network. SIGCOMM '04, pages 145–158. ACM, 2004.

[22] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.

[23] DD Kandhlur, Kang G Shin, and Domenico Ferrari. Real-time communication in multihop networks. *Parallel and Distributed Systems, IEEE Transactions on*, 5(10):1044–1056, 1994.

[24] Ben Kao and Hector Garcia-Molina. Deadline assignment in a distributed soft real-time system. *Parallel and Distributed Systems, IEEE Transactions on*, 8(12):1268–1274, 1997.

[25] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[26] L. Kleinrock. *Queueing systems. volume 1: Theory.* 1975.

[27] Hermann Kopetz and Günter Grünsteidl. Ttp-a time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 524–533. IEEE, 1993.

[28] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Computers, IEEE Transactions on*, 100(8):933–940, 1987.

[29] Fernando Kuipers, Piet Van Mieghem, Turgay Korkmaz, and Marwan Krunz. An overview of constraint-based path selection algorithms for qos routing. *IEEE Communications Magazine, 40 (12)*, 2002.

[30] S. K. Kweon and K. G. Shin. Achieving real-time communication over ethernet with adaptive traffic smoothing. In *in Proceedings of RTAS 2000*, pages 90–100, 2000.

[31] Luis E Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional pc hardware. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 14–23. IEEE, 2006.

[32] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[33] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[34] J. Liu. *Real-Time Systems.* Prentice Hall, 2000.

[35] Jork Loeser and Hermann Haertig. Low-latency hard real-time communication over switched ethernet. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 13–22. IEEE, 2004.

[36] N. McKeown. Software-defined networking. INFOCOM keynote talk, 2009.

[37] David L Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[38] W. A. Mittelstadt, P. E. Krause, P. N. Overholt, D. J. Sobajic, J. F. Hauer, R. E. Wilson, and D. T. Rizy. The doe wide area measurement system (wams) project demonstration of dynamic information technology for the future power system. In *EPRI Conference on the Future of Power Delivery*, 1996.

[39] L. Monnerat and C. Amorim. Peer-to-peer single hop distributed hash tables. In *in GLOBECOM'09*, pages 1–8, 2009.

[40] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[41] Seyedbehzad Nabavi, Jianhua Zhang, and Aranya Chakrabortty. Distributed optimization algorithms for wide-area oscillation monitoring in power systems using interregional pmu-pdc architectures. *Smart Grid, IEEE Transactions on*, 6(5):2529–2538, 2015.

[42] M. Parashar and J. Mo. Real time dynamics monitoring system: Phasor applications for the control room. In *42nd Hawaii International Conference on System Sciences*, pages 1–11, 2009.

[43] Gabriel Parmer and Richard West. Predictable interrupt management and scheduling in the composite component-based system. In *Real-Time Systems Symposium, 2008*, pages 232–243. IEEE, 2008.

[44] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[45] A. G. Phadke, J. S. Thorp, and M. G. Adamiak. New measurement techniques for tracking voltage phasors, local system frequency, and rate of change of frequency. *IEEE Transactions on Power Apparatus and Systems*, 102:1025–1038, 1983.

[46] Linh TX Phan and Insup Lee. Improving schedulability of fixed-priority real-time systems using shapers. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 217–226. IEEE, 2013.

[47] T. Qian, F. Mueller, and Y. Xin. A real-time distributed hash table. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

[48] Tao Qian, Aranya Chakrabortty, Frank Mueller, and Yufeng Xin. A real-time distributed storage system for multi-resolution virtual synchrophasor. In *Power & Energy Society General Meeting*. IEEE, 2014.

[49] Tao Qian, Frank Mueller, and Yufeng Xin. Hybrid edf packet scheduling for real-time distributed systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 37–46, July 2015.

[50] Tao Qian, Frank Mueller, and Yufeng Xin. A linux real-time packet scheduler for reliable static sdn routing. In *Real-Time Systems (ECRTS), 2017 29th Euromicro Conference on*, June 2017.

[51] Nageswara SV Rao and Stephen G Batsell. Qos routing via multiple paths using bandwidth reservation. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 11–18. IEEE, 1998.

[52] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *in Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[53] Juan Rivas, J Gutierrez, J Palencia, and M Gonzalez Harbour. Deadline assignment in edf schedulers for real-time distributed systems.

[54] Juan M Rivas, J Javier Gutiérrez, J Carlos Palencia, and M González Harbour. Optimized deadline assignment for tasks and messages in distributed real-time systems. In *Proceedings of the 8th International Conference on Embedded Systems and Applications, ESA*. Citeseer, 2010.

[55] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 49–60. IEEE, 2007.

[56] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[57] Rui Santos, Moris Behnam, Thomas Nolte, Paulo Pedreiras, and Luís Almeida. Multi-level hierarchical scheduling in ethernet switches. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 185–194. ACM, 2011.

[58] John W Suurballe and Robert Endre Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984.

[59] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[60] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 50–61. ACM, 2011.

[61] Dahai Xu, Yang Chen, Yizhi Xiong, Chunming Qiao, and Xin He. On finding disjoint paths in single and dual link cost networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1. IEEE, 2004.

[62] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308. IEEE, 2012.

[63] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.

[64] Hui Zhang and Domenico Ferrari. Rate-controlled service disciplines. *J. High Speed Networks*, 3(4):389–412, 1994.

[65] Yuting Zhang and Richard West. Process-aware interrupt scheduling and accounting. In *RTSS*, volume 6, pages 191–201, 2006.

[66] Kai Zhu, Yan Zhuang, and Yannis Viniotis. Achieving end-to-end delay bounds by edf scheduling without traffic shaping. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1493–1501. IEEE, 2001.