TOWARDS ADAPTIVE EDGE COMPUTING AND COMMUNICATION FOR DISASTER RESPONSE

A Dissertation

by

MENGYUAN CHAO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Radu Stoleru
Committee Members,	Anxiao Jiang
	Dilma Da Silva
	I-Hong Hou
Head of Department,	Scott Schaefer

August 2020

Major Subject: Computer Science

ABSTRACT

Effective disaster response depends on technologies that enable timely gathering and processing of data from different disaster sites in the disaster area. However, a large scale disaster like Hurricane Maria 2017 may totally destroy the infrastructure of the affected area, which makes both the Internet and cloud unavailable and the disaster response inefficient. To deal with this issue, Edge Computing and Communication (ECC), which performs data processing and data transmission at the edge of network, is applied to provide temporary computing and communication services to the first responders. By ECC, the first responders can process a large amount of sensing data at each disaster site and only send the processing result back to the Emergency Operation Center (EOC) via Disaster Response Networks (DRNs). However, the mobile devices carried by the first responders have limited computing resources, the wireless network connecting them is dynamic, and the applications used to analyze sensing data are computation-intensive and have diverse performance goals. Therefore, a lot of challenges exist for achieving high efficient ECC for disaster response.

In this dissertation, in order to address the aforementioned challenges, we propose an adaptive edge computing and communication framework for disaster response. This framework consists of a Distributed Mobile Stream Processing (DMSP) platform, a CNN-based multitask video processing system, and a user-customizable delay-tolerant routing protocol. Specially, the mobile stream processing platform allows first responders to perform computation-intensive stream processing on a cluster of mobile devices. It adopts feedback-based task configuration, resilient task assignment and adaptive stream grouping to deal with dynamic computing resources and network connectivity. The CNN-based multitask video processing system allows first responders to extract different Information of Interests (IoIs) from the on-body camera video stream by using different CNNs derived from the same base CNN. These CNNs can adaptively share different amount of common layers to trade off between the total computation cost and inference accuracy. Each of these CNNs can be adaptively divided into two separate parts to run on different mobile devices to meet the specific performance goals. The user-customizable delay-tolerant routing protocol enables first responders to send back different information obtained at each disaster site to EOC based on the specific Quality of Service (QoS) requirements. We evaluate the proposed framework through extensive real-world experiments and simulations, which demonstrate its effectiveness in enabling high efficient ECC for disaster response.

DEDICATION

To My Beloved Family.

ACKNOWLEDGMENTS

First, I would like to thank my advisor Dr. Radu Stoleru for his support, encouragement and guidance during my Ph.D. period. I still remember at our first 1-1 meeting, he said, "All you guys are smart, the only requirement is working hard." Now, at the end of my Ph.D. period, I feel this sentence is correct more than ever. I still remember at my early Ph.D. stage, he was so patient to guide me to find the correct direction and teach me the correct research methods. I still remember when I was stuck on a problem, he spent a lot of time on discussing with me and gave me precious advice to move forward. I still remember when I was under the deadline pressure, he made some jokes to help me relax. Without his generous help, I would not complete my Ph.D. work.

Second, I would like to thank my committee members, Dr. Anxiao Jiang, Dr. Dilma Da Silva and Dr. I-Hong Hou for their insightful comments and feedback. Their feedback at my prelim let me feel confident about my research topic and clarify the problem I am going to solve. Without their constructive comments, my later research work cannot be so smooth.

Third, I would like to thank my labmates Dr. Chen Yang, Dr. Wei Zhang, Dr. Ala Altaweel and Dr. Suman Bhunia for their helpful discussion and insightful suggestion during my Ph.D. period. I still remember that Dr. Yang spent a whole night with me at the lab to polish my first paper, I still remember that Dr. Zhang helped me solve some complicated cross-compling problems, and I still remember those unforgettable days I spent together with Dr. Altaweel and Dr. Bhunia to prepare for the exercise at Disaster City. Your carefulness and attention in research always inspire me.

Fourth, I would like to thank other LENSS members Qian, Congcong, Jay, Mahima, Yukun, Mohammed, Amran, Liuyi, Ben, Aaron, Ali and others. It is my honor to work with you and hear some useful feedback and interesting ideas from you! I also want to thank my friends Jiayi, Di, Zelun, Jay, Jasmine, etc. I really enjoy those happy moments with you!

Last but not the least, I would like to thank my parents and other family members for their supports. Specifically, I would like to thank my wife Ting and son Frank, their invaluable companionship is always the source of my motivation to move forward.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Radu Stoleru, Professor Anxiao Jiang and Professor Dilma Da Silva of the Department of Computer Science and Engineering at Texas A&M University, and Professor I-Hong Hou of the Department of Electrical and Computer Engineering at Texas A&M University.

All work for the dissertation was completed by student Mengyuan Chao, under the advisement of Professor Radu Stoleru of the Department of Computer Science and Engineering at Texas A&M University, and in collaboration with Professor Harsha Chenji of the Department of Electrical Engineering and Computer Science at Ohio University, Professor Evdokia Nikolova of the Department of Electrical and Computer Engineering at The University of Texas at Austin, Dr. Shuochao Yao at the Department of Computer Science at University of Illinois Urbana-Champaign, Dr. Chen Yang, Dr. Ala Altaweel, Yukun Zeng, Liuyi Jin of Department of Computer Science and Engineering at Texas A&M University, Roger Blalock and Maxwell Maurice of the Communications Technology Laboratory at the National Institute of Standards and Technology (NIST).

Funding Sources

Graduate study was supported by scholarships from Texas A&M University. This work was made possible by NIST under Grant Number #70NANB17H190. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of NIST.

NOMENCLATURE

CNN	Convolutional Neural Networks
DMSP	Distributed Mobile Stream Processing
DRN	Disaster Response Network
DTN	Delay Tolerant Network
EOC	Emergency Operation Center
ECC	Edge Computing and Communication
HPC	High Performance Computing
IoI	Information of Interest
MSP	Mobile Stream Processing
PDD	Packet Delivery Delay
PDE	Packet Delivery Efficiency
PDR	Packet Delivery Rate
PDS	Standard Deviation of Packet Delivery Delay
QoS	Quality of Service
RSSI	Received Signal Strength Indicator
TTE	Total Transmission Energy

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

1.1 Motivation

Natural disasters are becoming more frequent, growing more destructive, and affecting more people than ever before [1]. For example, Hurricane Maria in 2017 devastated entire Puerto Rico by killing 2982 lives and causing US$90 billion damage [2]. In order to minimize the personnel and property loss caused by disasters, developing techniques to effectively support disaster response is critical. Recently, with the development of AI technology, more and more AI-based applications, such as victim tracking [3], medical assistant [4], damage assessment [5], etc., are installed on the first responders' mobile phones to help them perform rescue tasks in the disaster area. Most of these AI-based applications have a thin client architecture, which gathers sensing data at each disaster site and offloads the computation-intensive analysis to the remote cloud. An effective disaster response depends on efficient data gathering and processing at each disaster site of the affected area [6], so that the rescue dispatchers at the Emergency Operation Center (EOC) can have an overview of the whole disaster to properly coordinate different disaster response organizations. However, large scale disasters may totally destroy the infrastructure of the affected area, which makes the Internet and cloud inaccessible. For example, 89.3% of the cellular towers in Puerto Rico are still out of service even after 9 days of Hurricane Maria [7]. In this case, the normal "processing after gathering" approach, which collects sensing data locally at each disaster site and forwards the data to process at the cloud does not work. To deal with this issue, Edge Computing and Communication (ECC) [8–10], which performs both data processing and gathering at the edge of network, is applied to provide temporary computing and communication services to the first responders. By using ECC, the first responders can process sensing data at each disaster site and only send the processing results back to EOC via the Disaster Response Networks (DRNs) [6].

Although ECC is promising for achieving effective disaster response, a lot of challenges need to be overcome before it becomes a solid solution. First, mobile devices carried by the first responders usually have very limited computing resources. However, the stream processing applications utilized for analyzing the sensing data at each disaster site requires a lot of computation. In this case, *how to leverage the resource-constrained mobile devices to support the computation intensive stream processing applications* becomes the first challenge. Many existing solutions [11–17] offload the computation-intensive tasks to the cloud or nearby High Performance Computing (HPC) servers to reduce the computation workload at mobile devices. However, as we mentioned earlier, in an infrastructure-less scenario like disaster response, a stable access to the cloud or HPC servers is not always available, especially for the first 24 hours after a disaster, when the technical teams have not set up the temporary cellular tower yet. Even after the temporary cellular tower is set up, the overall bandwidth from it to the cloud is still limited. First responders should not send all the sensing data at each disaster site to the cloud through the temporary cellular tower, because that will easily use up all the bandwidth. Some other existing solutions [8, 18] propose to offload some computation workload to the nearby mobile devices to perform Mobile Stream Processing (MSP). However, as far as we know, these solutions underestimate two critical aspects of distributed mobile stream processing: One is the *dynamic computing resources* at mobile devices and the other is the *dynamic wireless networks* connecting them. With dynamic computing resources, how much workload can each mobile device undertake and how the performance of offloaded tasks will be are unpredictable. If an "offloadee"mobile device which is assigned with lots of stream processing tasks starts some of its own applications, the actual available computing resources for stream processing will decrease. If the "offloader" mobile device does not realize this situation and continues to send the original amount of stream there, congestion and increasing queuing delay will occur at that "offloadee", which further affects the overall application performance. With dynamic wireless networks, the bandwidth between mobile devices changes from time to time. If the bandwidth between an "offloader" and an "offloadee" decreases below a certain threshold, communication instead of computation will become the new bottleneck of stream processing. Besides, the wireless connections among mobile devices are sometimes intermittent. To improve the resilience of mobile stream processing, when an "offloader" offloads stream processing tasks to nearby mobile devices, it should carefully choose "offloadees" that are more available to itself. All the above factors make conducting distributed mobile stream processing at the edge challenging.

Second, video cameras now are very important sensors for disaster response [19–22]. With on-body cameras, first responders can record the whole rescue process without any interference. This is pretty important for first responders because they are already overloaded by other activities. However, just recording video stream cannot give the first responders any instant summary about the Information of Interests (IoIs) they concern. For example, how many victims have they rescued? Are those victims male or female? Are they old or young? Do they look happy or sad? Ideally, all these basic information about victims should be extracted automatically from the video stream during the rescue process. When a first responder wants to check this information and sends it to EOC, he just needs to take out his mobile phone and clicks one button, instead of looking through the whole video again and recording the above information by hand. To enable automatic victim information extraction from video streams, a multitask video processing application based on Convolutional Neural Networks (CNNs) can be applied to extract different IoIs. However, *executing multiple computation intensive CNNs on the resource-constrained mobile device* is not trivial. Existing solutions can be roughly divided into three categories: offloading, compression, and sharing. First, most of the offloading strategies [14, 23–26] assume that there is a connection between mobile devices and the remote cloud (or a nearby HPC server) so that parts of CNNs can be offloaded there to achieve the desired performance. However, as we mentioned earlier, a stable connection to the cloud or a nearby HPC server is not available during the disaster response. Other offloading solutions [27–31] try to distribute a whole CNN to several wireless connected mobile/IoT devices. However, they only consider the single CNN case, which is much simpler than the multi-CNNs scenario we consider. Second, the model compression strategies construct efficient CNN models for mobile/IoT devices through different compression techniques, such as low-rank expansions [32], parameter quantization [33], pruning and Huffman coding [34], fully factorized convolution [35], depth-wise separable convolution [36], channel-wise sparse connection [37], etc. Unfortunately, these solutions provide a one-for-all model, i.e., a fixed model compression technique is used for different performance goals, which easily leads to the sub-optimal solution. Third, the sharing strategies try to reduce the total computation costs [23] and memory footprint [38] by sharing common layers among different CNNs. However, these solutions only consider running multiple CNNs on a single device. Therefore, the performance improvement is restricted by the single device's resource and their system scalability is poor.

Third, at the early stage of disaster response, the cellular towers of the disaster area have not been recovered and the temporary cellular tower has not been set up yet. DRNs are used to provide a temporary communication infrastructure for disaster response. In DRNs, battery-powered wireless routers are deployed at each disaster site [39]. First responders send the processing results at each disaster site back to EOC via the wireless routers. Since an end-to-end connection is not available in DRNs, packets are stored in the wireless routers temporarily. Vehicles (e.g., ambulances, supply vehicles, patrol cars) with on-board wireless routers move around the disaster area, collect packets stored in the routers at different disaster sites, carry and forward them to the intermediate or destination nodes (EOC) [40]. Many existing Delay Tolerant Networking (DTN) routing protocols can be used in DRNs [41–44], which improve the routing metrics Packet Delivery Delay (PDD) and Packet Delivery Rate (PDR) by using an unlimited level of packet replications [45]. These approaches, however, are not energy-efficient, because the unlimited level of packet replication depletes the batteries of the wireless routers soon. To prolong the lifetime of DRNs, *how to restrict the level of packet replication to reduce Total Transmission Energy (TTE), while maintaining other metrics like PDD and PDR at an adequate level* becomes the first challenge we need to overcome. Moreover, most existing DTN routing protocols only address routing metrics such as PDD, PDR, and TTE. However, in DRNs, the Standard Deviation of Packet Delivery Delay (PDS) is also important. A path with the minimum PDD might have a large PDS, which makes the actual delivery delay of some packets much longer than expected. If the first responders prefer some information to be sent back to EOC with a stable delay, the path with the minimum PDS is actually a better choice. To satisfy the preference of different users, the second challenge arises: *how to allow first responders to express their preferences on PDD or PDS when choosing routing paths in DRNs*?

Furthermore, first responders sometimes have the requirements to deliver some urgent information to EOC before a deadline. Otherwise, the value of this information will be discounted. To this end, the third challenge arises: In DRNs, *how to maximize the probability of delivering packets to the destination before a given deadline*?

To address all the aforementioned challenges, we propose an adaptive edge computing and communication framework for disaster response. This framework consists of a Distributed Mobile Stream Processing (DMSP) platform, a CNN-based multitask video processing system, and a user-customizable delay-tolerant routing protocol. Specially, the mobile stream processing platform allows first responders to perform computation-intensive stream processing on a cluster of wireless-connected mobile devices. It adopts feedback-based executor and task configuration, resilient task assignment, and adaptive stream grouping to deal with dynamic computing resources and network connectivity. The CNN-based multitask video processing system allows first responders to extract different IoIs from the on-body camera video stream by using different CNNs derived from the same base CNN. These CNNs can adaptively share different amount of common layers to trade off between the total computation cost and inference accuracy. Each of them can be adaptively divided into two separate parts to run on different mobile devices to meet the specific performance goals. The user-customizable delay-tolerant routing protocol enables first responders to send back different information obtained at each disaster site to EOC based on the specific Quality of Service (QoS) requirements. We evaluate different components of the proposed framework through extensive simulations and real-world experiments, which demonstrate its effectiveness in enabling high efficient ECC for disaster response.

1.2 Dissertation Statement

As effective disaster response depends on timely gathering and processing of data from different disaster sites of the infrastructure-less disaster area, the design of an adaptive edge computing and communication framework, which consists of a distributed mobile stream processing platform, a CNN-based multitask video processing system and a user-customizable delay-tolerant routing protocol, is critical to enable first responders to process the huge sensing data at each disaster site and send back the important processing results to EOC through disaster response networks.

1.3 Main Contributions

The main contributions of this dissertation are outlined as follows:

- We propose **feedback-based mobile stream processing (F-MStorm)**, which adopts feedback-based approaches in the configuration, scheduling, and execution levels of system design to deal with the dynamic computing resources of mobile devices. We implement F-MStorm on Android phones and evaluate its performance through benchmark applications with different computing resource conditions. We show that it achieves up to 75% lower response time, 10% higher throughput, and consumes 23% less communication energy than the state-of-the-art systems.

- We propose **resilient mobile stream processing (R-MStorm)**, which improves the MSP survivability by 1) assigning tasks to mobile devices with higher availability to improve the availability of physical stream paths; 2) assigning tasks of the same application components to different devices to increase the diversity of physical stream paths; 3) adopting adaptive stream grouping to efficiently direct the output stream of upstream tasks to downstream tasks; 4) adopting adaptive stream selection to skip stream data to alleviate congestion caused by network disconnection and stream redirection. We implement R-MStorm on Android phones and evaluate its performance through a video face recognition application under different network conditions. We show that, compared with baseline approaches, R-MStorm achieves up to 1.5x higher throughput, 75% lower response time, at a cost of 3.3% accuracy loss.

- We propose **an adaptive execution framework for CNN-based multitask video processing (AMVP)**, which combines three types of orthogonal strategies (offloading, compression and sharing) to support the execution of multiple computational intensive CNNs on resource-constrained mobile devices. AMVP selects the most appropriate CNN implementation and executing device for each vision processing task based on the user performance goals and actual available system resources. We implement AMVP on Android phones and evaluate its performance through a CNN-based multitask facial processing application. We show that AMVP achieves up to 60% lower latency and 10% higher throughput than two other status quo approaches with comparative accuracy.

- We propose **an energy-aware risk-averse routing protocol (EAR)**, which 1) applies the risk-aversion concept to address PDS, an important routing metric in DRNs that is overlooked by previous DRN routing protocols; 2) leverages a parameter L to restrict the level of packet replication; 3) applies a differentiated service model to deliver packets with less TTE while maintaining other metrics at an adequate level; 4) introduces a "λ-optimal" algorithm to search for the routing path with the lowest risk or highest probability to deliver the packets before a deadline; 5) extends the "λ-optimal" algorithm to a multipath version and proposes an EAR routing protocol based on it. We evaluate EAR through extensive simulations on the Opportunistic Network Environment (ONE) simulator [46]. We show that EAR provides flexible control of the routing risks and delivers packets to the destinations in a more energy-efficient way (up to 8x higher Packet Delivery Efficiency (PDE) with 4% lower PDR) than the well-known DRN routing protocols Prophet, MaxProp, RAPID, and Spray&Wait.

1.4 Organization

This dissertation is organized as follows. The current section motivates our work and states the contributions of the research. In Chapter 2, the state-of-art solutions are presented. In Chapter 3, we present the system architecture of the proposed edge computing and communication framework. In Chapter 4, the design of feedback-based mobile stream processing is presented. In Chapter 5, we present the design of resilient mobile stream processing. In Chapter 6, we present the design of an adaptive execution framework for CNN-based multitask video processing. In Chapter 7, the energy-aware risk-averse DRN routing protocol is presented. Finally, in Chapter 8, we conclude this dissertation and present a few future directions.

2. STATE OF THE ART

In this section we present the state-of-the-art of this dissertation. We first present the related work for mobile stream processing, which includes computation offloading, edge computing and distributed stream processing systems. Next, we discuss the state-of-the-art solutions for CNN-based video processing, which are roughly divided into three categories. Finally, we present the related work about routing protocols in DRNs.

2.1 Related Work For Mobile Stream Processing

Computation Offloading and Edge Computing. Computation offloading [11, 14, 18, 47–53] has been a popular research area. Based on the offloading destinations, the existing work can be categorized into cloudlet offloading, cloud offloading and hybrid offloading. Cloudlet offloading is named as edge computing [15, 54–56] as well, which aims at reducing the application response time and saving the Internet communication bandwidth by taking the control of computing applications, data, and services away from some central nodes (“core") to the logical extreme (“edge") of the Internet [57]. CloneCloud [48], JustInTime [49] and Odessa [11] are cloudlet offloading systems that leverage virtual machine migration mechanisms to reduce the application response time. However, all these systems rely on powerful nearby cloudlets, which are not always available in critical scenarios like disaster response, because first responders have to move from place to place and they are only allowed to carry on some lightweight mobile devices. *Different from the above systems, our F-MStorm and R-MStorm only utilize nearby mobile devices as offloading destinations*, which are more practical for the disaster response scenario, because first responders always work together as a group and their mobile devices can be connected together as a cluster. MAUI [47] is a cloud offloading system which enables energy-aware offloading by using remote function calls to the cloud. MCDNN [14] is a cloud offloading framework that employs a run-time scheduler to trade off the application accuracy for resource usage and latency. Orbit [51] and LEO [52] are similar systems that utilize profile-based partitioning of applications to offload computation tasks to hybrid computing resources. All these systems rely on the Internet access, which however are not always available in some critical scenarios, because first responders always work in some extremely difficult environments that do not provide any access to the Internet. *Different from these systems, F-MStorm and R-MStorm do not rely on the Internet.* Instead, they only need a mobile device to be set up as a hot spot, such that all the mobile devices in a group can be connected together. Some existing work like Hyrax [53] and Serendipity [18] also offload computing tasks to the nearby mobile devices. However, their work focuses on processing bounded batch jobs that have relatively low requirements on the latency. *Instead, F-MStorm and R-MStorm focus on processing unbounded stream data, which has a higher requirement on the latency.*

Distributed Stream Processing Systems. Apache Storm [58] is a distributed stream processing system deployed on the cloud servers. Many improvements based on Storm have been proposed [59–64]. Among them, AdaptiveStorm [59] continuously monitors the system performance and reschedules tasks at run-time to reduce the overall response time. T-Storm [60] accelerates stream processing by using traffic-aware scheduling, which minimizes the inter-device and inter-process traffics. R-Storm [61] improves the throughput and minimizes the network latency by maximizing the resource utilization. *All these works are closely related to F-MStorm and R-MStorm, but they do not consider the detailed differences among inter-device links, which however are essential in stream processing at the edge.* The authors in [62] propose a scalable centralized scheme for job reconfiguration, which minimizes the communication cost while keeping the nodes below a computational load threshold. The authors in [63] propose a dynamic resource scheduler for cloud-based distributed stream processing systems, which measures the system workload with the minimal overhead and provisions the minimum resources to meet the response time constraints. The authors in [64] provide a general formulation for the optimal data stream processing placement and takes explicitly into account the heterogeneity of computing and networking resources. Similar to these works, in F-MStorm and R-MStorm, we propose a general framework for stream task assignment that takes delay, energy and load balance all into account. *The difference is that, our problem is more challenging, because in stream processing at the edge, the inter-device delay* *is dynamic and the users' own application may cause disturbance to F-MStorm and R-MStorm.* Except for Storm and its subsequent improvements, there are other popular stream processing systems like Apache Spark Stream [65], Flink [66], Samza [67], etc. However, *all these systems are designed and implemented to run on cloud servers instead of mobile devices.* To the best of our knowledge, MStorm [8] is the first work that performs online distributed mobile stream processing at the edge. However, it is inefficient in the system configuration, task scheduling and execution aspects. *F-MStorm improves its efficiency by using the feedback information.* EdgeWise [68] is a stream processing engine for edge which incorporates a congestion-aware scheduler to improve throughput. However, all the above systems assume the wireless connections among devices are always on, which is not always true in real life. *R-MStorm improves the resilience of MSP by considering the dynamic network connection among mobile devices.* MobiStreams [69] also provides an MSP runtime in dynamic networks. However, it mainly focuses on the issue of fault tolerance. Swing [70] is an MSP framework which considers both dynamism and heterogeneity of devices to achieve performance objectives and energy efficiency. Frontier [71] is also an MSP system that deals with network dynamics through back pressure stream grouping and failure recovery. However, both of the above two systems overlook the effects of task assignment on the MSP resilience. They simply assign tasks to mobile devices in a simple round-robin manner, regardless of that fact that some devices may frequently go out of range and have low availability.

2.2 Related Work For CNN-based Video Processing

CNN offloading. The key idea of CNN offloading is moving some CNN layers or the whole CNN model from the resource constrained mobile devices to some resource sufficient servers, no matter the server is in the cloud or at the edge. MCDNN [14] is an earlier representative work which deploys different CNN variants on both cloud and mobile devices. The variants at the cloud have higher accuracy but higher computation cost and extra communication overhead. The variants at mobile devices have lower computation cost and no communication overhead but lower accuracy. MCDNN selects a proper CNN variant at runtime to adapt to the dynamic operating conditions. Another representative work of CNN offloading is Neurosurgeon [72], which performs CNN computation partition between mobile devices and cloud at the granularity of neural network layers. It demonstrates that, compared to those cloud-only solutions, this collaborative solution achieves lower latency, lower energy consumption and higher datacenter throughput. A recent work called Couper [26] brings edge server into CNN offloading by quickly slicing CNNs into components executing on both the cloud and edge. It proves via extensive experiments that a powerful edge server is essential for CNN offloading when the required latency is low while the network condition to the cloud is bad. Although the above solutions achieve significant performance improvement, they either rely on the cloud or a powerful edge server. Under some extreme conditions without Internet or powerful edge server, such solutions do not work. Recent works also start to study distributing CNN execution on several IoT or mobile devices. Modnn [27] and Mednn [28] utilize specific partition schemes to partition CNN models onto several mobile devices to alleviate device-level computing cost and memory footprint. DeepThings [30] proposes distributed inference on IoT devices by employing a fused tile partitioning of convolution layers to expose parallelism, a distributed work stealing approach to balance dynamic workload and a novel scheduling procedure to reduce the overall execution latency. Musical Chair [31] supports efficient recognition at local by harvesting aggregated computational power from IoT devices in the same network. It explores both data parallelism and model parallelism of CNN to deal with the inherit dynamic. *Different from above works which focus on the execution of single CNN, AMVP studies how to run multiple CNNs on the mobile devices, which is obviously more challenging.*

CNN compression. CNN compression uses different compression techniques to construct efficient CNN models for mobile devices. XNOR-Net [73] approximates both input and weights into binary values to reduce the computation workload for real-time inference at the cost of some accuracy loss. ThiNet [74] applies filter level pruning to compresses CNNs by greedily pruning the filter that has the minimum effect on the activation values of the next layer. Factorized Networks [35] factorizes a high-cost 3D convolution operation as a low-cost single intra-channel convolution and a linear channel projection to reduce the computation while maintain the accuracy. All these works, however, have an identical limitation: they use a fixed compression technique to compress a CNN, which results in a one-for-all model that cannot adapt to different performance goals and resource constraints. To deal with this issue, a recent work [75] enables on-demand model compression by applying proper compression techniques to different CNN layers, which achieves an optimal balance between performance goals and resource constraints. In current AMVP, there is no direct model compression technique to support CNN model compression. However, in the future, we plan to integrate some. Except for the above works which compress CNN layers to reduce computation cost and memory footprint, there are also some other works which compress the intermediate features to reduce the feature transmission size. For example, the authors in [76] claim that intermediate deep feature compression will become the next battlefield of collaborative intelligent sensing. In [77] and [78], the authors study the impact of lossy and near-losses feature compression on object detection accuracy. In [79], the authors present a lossy compression framework for intermediate deep feature compression based on quantization and codec, which has been adopted by the Audio Video Coding Standard Workgroup as the visual feature coding standard. *In AMVP, in order to reduce the feature traffic size between separated CNN components, we use similar methods to compress the feature before sending.*

CNN sharing. The key idea of CNN sharing is to share common layers or parameters among multiple related CNNs to reduce the total computation workload or memory footprint. For example, in NestDNN [38], the authors propose a nesting method to allow different variants of a deep learning model to share common parameters, so that it can dynamically select the optimal resource-accuracy trade-off at runtime to fit each model's resource demand to the system available resources. In Mainstream [23], the authors propose to share some common layers among multiple CNNs at an edge server to reduce the computation workload. At deployment time, based on the available resources and mix of applications on the edge server, it automatically determines the right trade-off between per-frame accuracy and more frames per second by choosing different number of shared layers. AMVP adopts similar layer sharing strategy as Mainstream. *The difference is, instead of choosing proper shared layers based on the available resource at edge server, AMVP needs to consider the available resources of a mobile device cluster, as well as the condition of* *wireless network connecting them.*

2.3 Related Work For Disaster Response Networks

Many routing protocols have been proposed for general DTNs [80]. *Epidemic routing* [41] is a flood-based routing protocol that assumes purely random encounters among nodes and requires each node to replicate and transmit packets to the new contacts that do not have a copy. It achieves high PDR and low PDD via unlimited level of packet replication, which however, wastes a lot of resources, because it never eliminates replications that are unlikely to improve the performance. To reduce unnecessary replications, *Prophet* [42] maintains a set of probabilities for successful delivery to different destinations based on the history information. A node replicates packets to the encountered nodes only when they do not have the packets and have a better chance to deliver these packets. Similar to *Prophet*, *MaxProp* [43] is a history-based routing protocol that maintains a queue ordered by the estimated likelihood of a future transitive path to the destination of each packet. When nodes encounter, the packets at the head of the queue are transmitted first and those at the tail are dropped first if the buffer space is full. It should be noted that all the above protocols incidentally improve PDR and PDD by increasing the packet replicas. *RAPID* [44] is a utility-based protocol that intentionally affects the routing metrics by estimating the marginal utility gained from replicating a packet. Different from aforementioned protocols, *EAR* both incidentally and intentionally affects routing metrics by using some specific parameters. Moreover, to the best of our knowledge, none of the above protocols pays enough attention to PDS, which nevertheless, is very important in DRNs. The work *(p,q)-Epidemic* [81] studies the distribution of delay in a general way. It assumes that nodes have no knowledge about the network or mobility and the historic information is useless for predicting the future. This assumption, however, is not true in *EAR*, where the nodes follow some certain patterns to move. A recent work *ICR* [82] also regards PDS as an important routing metric. However, instead of providing a parameter to adjust the importance of PDS based on the preference, *ICR* uses a fixed number 1.65 to combine PDS with PDD as a metric. Recently, many researchers have introduced the understanding of social structures into DTN design [83–87]. By combing the knowledge on community structure and centrality, *BubbleRap* [88] maintains both the global and local rankings for each packet. A packet is replicated and transmitted to the community of the destination using the global ranking and then to the destination using the local ranking. SAPC [89] presents a similar two-stage routing algorithm which performs multi-copy spreading based on social activity and single-copy forwarding based on physical contact factor. *SMART* [90] assigns a weight to the link between two nodes by combining their encountering frequency and social closeness. Based on these links, packets are forwarded to the node whose shortest path to the destination has the lowest weight. Similar to *SMART*, *EAR* assigns a weight to the link between two nodes and routes along the shortest path. The difference is that *EAR* uses recurrent movement pattern information instead of social structure information. In TCCB [84], the authors try to use the past temporal correlations to infer the temporal social contact patterns in remaining valid time of data and propose an efficient temporal closeness and centrality-based data forwarding strategy. In [85], the authors construct a routing-delivery scheme for the opportunistic networks based on the node profile. They claim that, the node profile can effectively characterizes nodes by analyzing and comparing their attributes. In DRNs, different vehicles also have their own movement patterns (or profile). Similarly, we can exploit these movement patterns to efficiently forward packets to the destination. A recent work *CTR* [91] resembles *EAR* by using some movement pattern information to improve the efficacy of packet forwarding. However, it only exploits the gathering pattern of survivors in the shelters, whereas *EAR* exploits the movement patterns of all mobile agents. A key objective of DRNs routing is to reduce the energy consumption per delivered packet to prolong the lifetime [91]. To achieve this goal, *Spray&Wait* [92] sets a strict upper bound L on the number of replicas for each packet. It performs routing by first "spraying" L packet copies into the network, and then "wait" until one of the relays meets the destination. In [93], the authors extend *epidemic routing* by mathematically characterizing the trade-off between packet forwarding efficacy and energy conservation as a heterogeneous dynamic optimal control problem. They prove that the optimal dynamic forwarding decisions follow a simple threshold based structure, where the threshold for each node depends on its current remaining energy. To reduce the resource overhead per packet, *ICR* [82] exploits the recurrent mobility and contact patterns in DRNs and proposes the differentiated service that allows DTN nodes to manage the energy consumption based on the relative urgency of messages. Inspired by these significant works, *EAR* proposes a mechanism that combines the strict upper bound, threshold and differentiated service, which enables packets to be delivered in an energy-efficient way while maintaining other metrics like PDD, PDS and PDR at an adequate level.

an

de:

dif

3.1

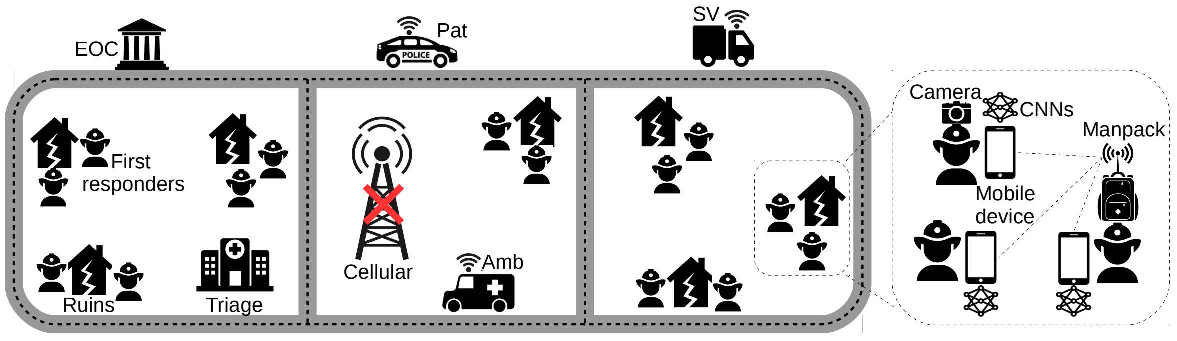


Figure 3.1: Hardware architecture of adaptive edge computing and communication framework for disaster response.

An overview of the hardware architecture of our edge computing and communication framework for disaster response is shown in Figure 3.1, which mainly consists of two parts: The first part is set up at each disaster site where first responders are dispatched to perform rescue tasks and the second part is built up by leveraging different vehicles (including patrol cars, supply vehicles, ambulances, etc.) equipped with wireless routers moving around the disaster area.

We name the first part as "**edge bubble**", where a battery-powered wireless (WiFi/LTE) router carried in a first responder's manpack is set up as the temporary communication infrastructure at the edge. By connecting to the wireless router, mobile devices carried by the first responders can connect with each other to form a mobile computing cluster, which is used as the temporary com-

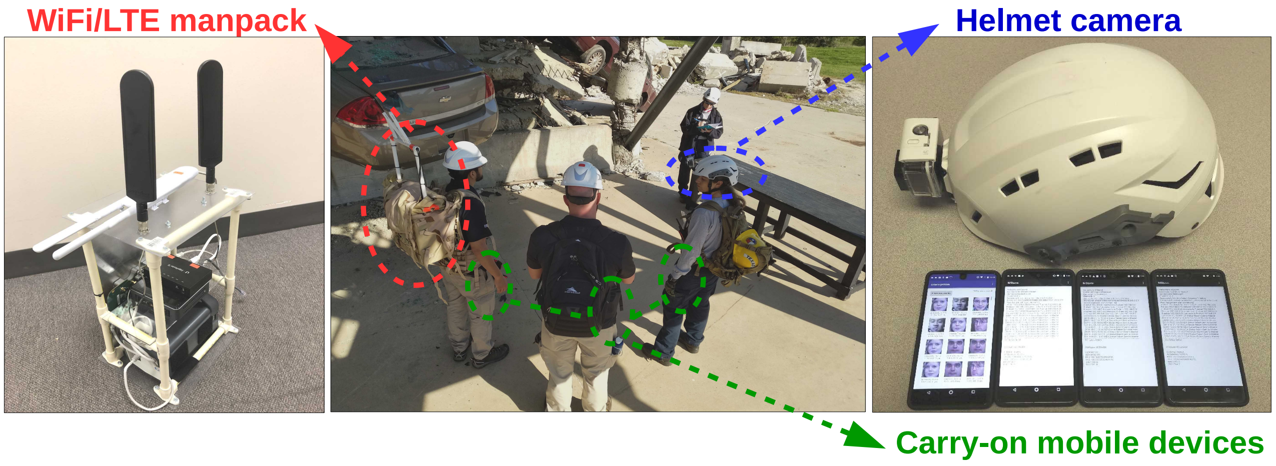


Figure 3.2: Hardware implementation of an edge bubble.

puting infrastructure at the edge. When a first responder needs to execute a computation intensive stream processing application, he can offload some stream processing tasks to other mobile devices in the same cluster to perform distributed mobile stream processing. There is also a first responder in each team equipped with an on-body/helmet camera to record the video during the rescue process. By pairing the camera with a mobile device through a WiFi hot-spot link, video stream can be pulled from the camera to the mobile device to perform CNN-based video processing. The processing results will be temporarily stored in a hard disk co-located with the wireless router at the manpack. In Figure 3.2, we show the real hardware implementation of our "edge bubble". The wireless manpack is implemented by a Ubiqutous® Wifi router, a Baicells® eNodeB and an Intel NUC® running an opensource EPC. The helmet of a first responder is equipped with a Yi® 4K camera, which supports both WiFi AP and hotspot mode. Each first responder is equipped with an Essential® Android phone, which supports Wifi AP, hotspot and LTE mode at the same time.

We name the second part as "**disaster response networks**", where vehicles equipped with on-board wireless routers move around the disaster area, collect packets stored in the wireless routers at different disaster sites, carry and forward them to the intermediate (other disaster sites or vehicles) or destination node (EOC). In this style of "store and forward" delay-tolerant network, an end-to-end connection does not exist. The delay of each hop is mainly determined by the moving pattern an

to the oth

3.2 Soft

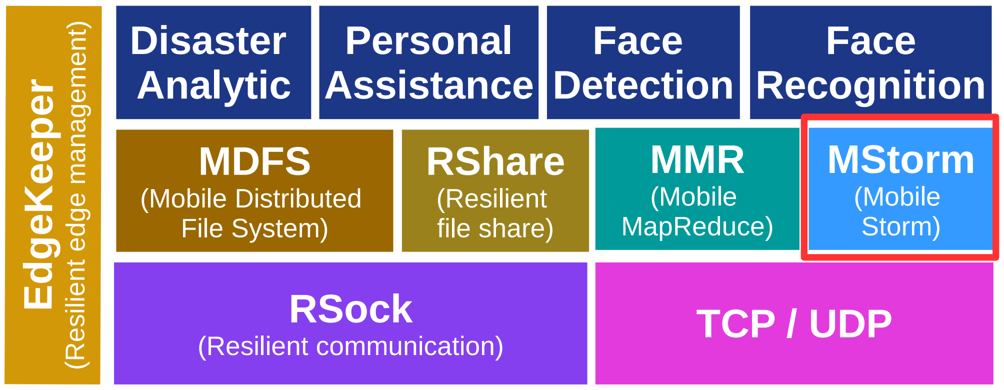


Figure 3.3: Distressnet-NG ecosystem and the role of MStorm.

Before we describe the software architecture of our adaptive edge computing and communication framework for disaster response, we first spend some time to introduce our Distressnet-NG ecosystem. As shown in Figure 3.3, our Distressnet-NG ecosystem contains many different components. The EdgeKeeper component is used for resilient edge management, its role in Distressnet-NG is as Zookeeper's role in Hadoop. The MDFS component is a mobile distributed file system, and the RShare component is a resilient file sharing application. Both MDFS and RShare depend on a resilient communication component called RSock. RSock is designed and implemented as a transport layer protocol. It can adapt to network connectivity ranging from totally disconnected to well connected. MMR is a batch processing platform deployed on mobile devices, its role in Distressnet-NG is very similar to the role of MapReduce in Hadoop. Finally, our MStorm platform is a stream processing platform running on mobile devices. Its role to Distressnet-NG is very similar to the role of Storm, Spark or Flink in the cloud.

Next, we introduce the software architecture of our edge computing and communication framework for disaster response in Figure 3.4, which mainly consists of MStorm for mobile stream pro-

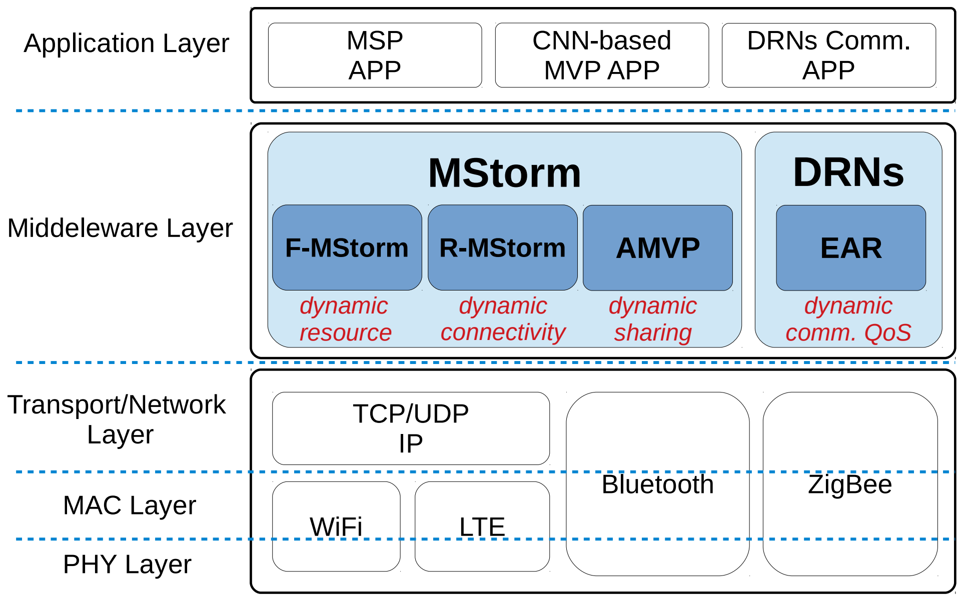


Figure 3.4: Software architecture of adaptive edge computing and communication framework for disaster response.

cessing and DRNs for communication. F-MStorm and R-MStorm are two modules inside MStorm which deal with dynamic computing resources and dynamic network connectivity, respectively. AMVP is a module in MStorm supporting dynamic layer sharing among multiple CNN models for multitask video processing. EAR is a routing protocol inside DRNs that helps choosing routing paths to send the processing results of MStorm back to EOC. When disaster responders start to rescue at the disaster site, the video stream taken by the on-body/helmet camera will be pulled to the mobile device by a CNN-based multitask video processing application running on top of MStorm. The AMVP module of MStorm chooses the most appropriate CNN implementation for each vision analysis task inside the application and offloads parts of CNNs to execute on other mobile devices. The processing results will be stored in a folder of local file system. The DRNs module will fetch the stored results and forward it to the wireless router. The DRNs module at the wireless router will forward the processing result to a proper vehicle passing by based on the routing results calculated by the EAR routing protocol.

4. F-MSTORM: FEEDBACK-BASED MOBILE STREAM PROCESSING*

Emerging computation-intensive mobile applications [94–99] that process stream data collected by various mobile sensors often require more computation resources than a single device can provide. Therefore, many existing systems [11, 14, 52] offload the computation-intensive tasks to the cloud or nearby high performance computing (HPC) servers to achieve low latency. For example, MCDNN [14] offloads deep neural network based video processing tasks to the cloud, Odessa [11] chooses nearby HPCs as additional computing resources to recognize objects from real-time videos and LEO [52] utilizes on-chip DSP co-processors, GPUs together with the cloud to run inference algorithms. Although such systems achieve low latency and high throughput, resources in the cloud or nearby HPC servers are not always accessible in some infrastructure-less scenarios. For example, imagine the following scenario:

"A group of first responders, equipped with mobile devices, is assigned to a post-earthquake area to discover dangerous zones (e.g., leaking chemical pipes or unstable buildings) to avoid. The teams collect a large amount of data via different sensors (e.g., on body video cameras) and analyze the data in real time via analytical software. Usually, such data analysis requires significant computational resources and, thus, it is pushed to the cloud for analysis. However, the communication infrastructure was destroyed during the earthquake, which makes offloading to the cloud impossible. In such case, offloading computation to the nearby mobile devices at the edge becomes a promising option."

In this chapter, we focus on **a distributed stream processing system deployed on a cluster of mobile devices without an Internet access (stream processing at the edge)**. Different from most stream processing systems that run on a cluster of wire-connected servers in the cloud (such as Storm [58], Spark [65] and Flink [66], etc.), stream processing on a cluster of mobile devices is much more complicated, because mobile devices have very limited computation resources and

*Reprinted with permission from "F-mstorm: Feedback-based online distributed mobile stream processing" by Mengyuan Chao, Chen Yang, Yukun Zeng and Radu Stoleru in Proceedings of 2018 IEEE/ACM Symposium on Edge Computing (SEC), pages 273–285, Seattle, USA, 2018, Copyright 2018 by IEEE.

batteries, and the wireless links between different devices are unstable. Some existing works [18, 53] are designed for the same environment as ours. However, they focus on processing bounded batch jobs instead of unbounded stream data.

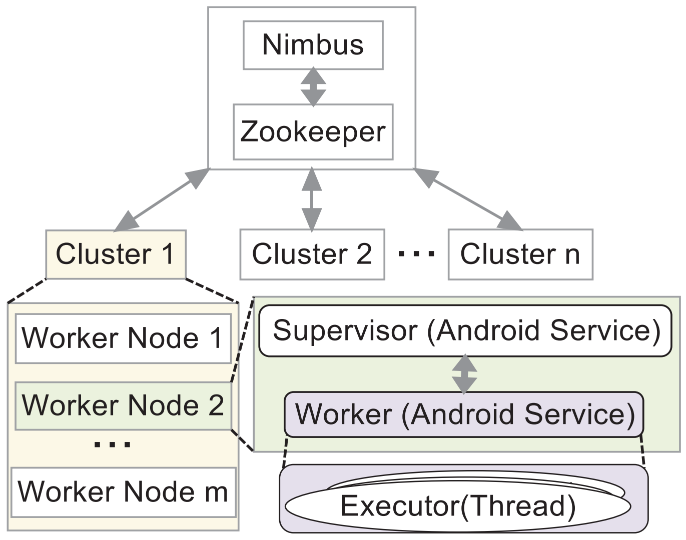
MStorm [8] makes an important first step towards mobile stream processing at the edge by implementing a lightweight system on mobile phones. It provides some basic functionality, such as *parallelism configuration*, *task scheduling* and *stream grouping*. However, since its current implementation ignores some specific characteristics of stream processing at the edge, it is inefficient as we demonstrate through some simple experiments. First, MStorm configures the number of executors (threads that execute tasks) at each device simply based on CPU cores while not taking into account the current CPU utilization. As the computation resources of a mobile device is also shared by other applications, this static configuration can easily lead to a bottleneck that negatively impacts the system performance (response time and throughput). Second, the task assignment, when MStorm assigns computation tasks to devices, is based on a naive round robin strategy. This may incur unnecessary inter-device traffic and consequently higher delay and energy consumption. Third, MStorm adopts a shuffle stream grouping mechanism, where upstream tasks distribute the output to downstream tasks uniformly at random. However, as downstream tasks may run on highly occupied devices, the shuffle grouping mechanism might cause congestion there and lead to high response time and low throughput.

To solve these problems, one potential solution is to carve out some static resources dedicated for stream processing [100]. However, unlike servers in the cloud, the resources of mobile devices at the edge are very limited and need to be shared with some other resource-intensive applications. It is unreasonable to allow MStorm to take up some resources even when there is no stream processing tasks. Another approach is to apply a pull model [101, 102] like most modern cloud computing systems, where the machines ask for tasks when they have free slots. However, this model is not enough for stream processing at the edge as it does not consider other factors like device-to-device delays and remaining batteries of devices. Our insight is that, *instead of adopting an open-loop task scheduling which assumes a static environment, a feedback-based approach that* *makes decisions based on the changing system state should be utilized to deal with the dynamic environment at the edge*.

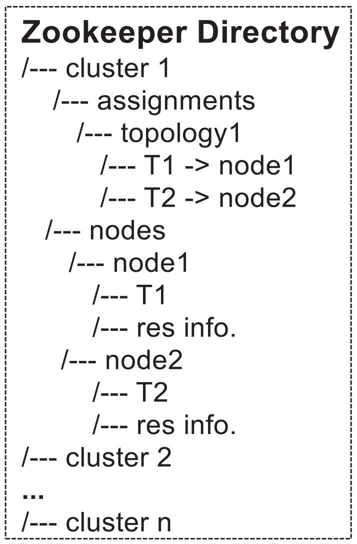
To accomplish this goal, we propose F-MStorm, a feedback-based online distributed mobile stream processing system. F-MStorm adopts a feedback-based approach at many different levels of system design so that the system can adapt quickly to the changing environment to achieve high performance. At the configuration level, F-MStorm configures the number of executors on each mobile device based on the free CPU resources of mobile devices and CPU usage of tasks. At the scheduling level, F-MStorm assigns tasks to mobile devices based on the task-to-task traffic and device-to-device communication delay and energy consumption. At the execution level, the upstream tasks distribute the output data to the downstream tasks based on the latter's stream arrival/processing rate and waiting queue length. We implement a prototype of F-MStorm on Android phones and evaluate its performance through a customizable benchmark application. We also compare F-MStorm with two scheduling algorithms proposed for Storm (i.e., T-Storm [60] and R-Storm [61]). The experimental results show that, by using the feedback information, F-MStorm achieves up to 3x lower response time, 10% higher throughput and 23% less communication energy than the state-of-the-art systems.

Our main contributions are summarized as follows:

- Through some real world experiments, we demonstrate that, without an accurate estimation of the current system state and appropriate adjustment of the initial configuration, task scheduling and stream grouping, MStorm suffers up to three order of magnitude increase in response time and 60% reduction in throughput (Section 4.1), which calls for the feedback-based system design.
- We propose F-MStorm (Section 4.2), which consists of a feedback-based configuration (FBC) method, a feedback-based task assignment (FBA) algorithm and a feedback-based stream grouping (FBG) strategy.
- We implement F-MStorm on Android phones and conduct real world experiments to evaluate



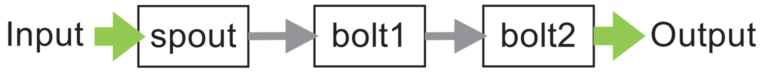
(a) MStorm Architecture



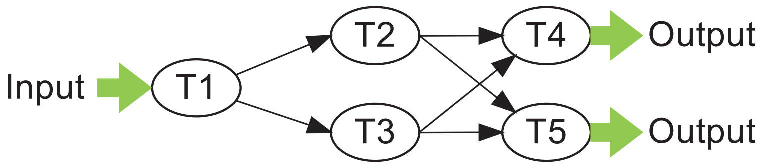
```
Zookeeper Directory
/--- cluster 1
    /--- assignments
       /--- topology1
          /--- T1 -> node1
          /--- T2 -> node2
   /--- nodes
      /--- node1
         /--- T1
         /--- res info.
     /--- node2
         /--- T2
         /--- res info.
/--- cluster 2
...
/--- cluster n
```

(b) MStorm system architecture.

Figure 4.1: MStorm system architecture



(a) Topology example



(b) Extended topology example

Figure 4.2: Example of an MStorm application.

its performance (Section 4.3), which demonstrate the superiority of F-MStorm over MStorm and two other state-of-the-art systems.

4.1 Background and Motivation

In this section, we at first briefly introduce the background of MStorm and its architecture. Then, we show the inefficiency of MStorm via three experiments. Finally, we motivate our F-MStorm by explaining why existing solutions do not work.

4.1.1 MStorm

MStorm [8] is the first online distributed stream processing system running on mobile devices with Android OS. It is designed for critical scenarios such as military operations and disaster response, where no Internet access is available, whereas the mobile devices of the team members are connected as a cluster through a manpack Wi-Fi access point. Instead of porting popular stream processing systems (such as Storm [58], Spark [65] or Flink [66]) running on the cloud, MStorm is designed and implemented from scratch with a lightweight infrastructure. This is paramount for mobile stream processing at the edge, which only has limited resources.

MStorm adopts some technical designs from Apache Storm. Its architecture is shown in Figure 4.1(a). An MStorm master node contains one **Nimbus** and one **Zookeeper** service. Nimbus schedules task execution while Zookeeper coordinates between Nimbus and mobile devices and maintains the cluster metadata in a directory-like structure shown in Figure 4.1(b). Every mobile device in the MStorm cluster runs a **supervisor** process and a **worker** process, both as Android services. The supervisor receives tasks from Nimbus and assigns tasks to the worker, while the worker manages multiple **executors** (threads) which are used to execute tasks. MStorm guarantees an at-most-once processing semantics.

An application in MStorm is modeled as a directed graph called **topology**. A topology contains two types of nodes, i.e., **spout** and **bolt**. A spout partitions the input stream into tuples and sends these tuples to downstream bolts. A bolt processes tuples from spout or upstream bolts, and sends the processed tuples to downstream bolts for further processing. We refer to a spout or a bolt as **application component** (or simply component) in the rest of the chapter. A directed edge between two nodes in a topology indicates that traffic flows from one to the other. Each component can spawn multiple parallel tasks which are executed by devices' executors. If we expand the component in a topology with multiple nodes, each of which represents an individual task, we get another directed graph, i.e., the **extended topology**. The extended topology graph shows the actual data flow between individual tasks. Figure 4.2 shows the topology and the extended topology of a sample MStorm application that contains one spout and two bolts, i.e., bolt1, bolt2. Each bolt further contains two parallel tasks, i.e., T2, T3 for bolt1, and T4, T5 for bolt2, respectively.

The MStorm application developer needs to provide the application topology as well as configure the *parallel tasks* for each application component. MStorm then decides *the number of executors* for each device and assigns tasks to devices for execution. The output tuples of each task may need to be sent to the downstream tasks. The mechanism for distributing tuples is called **stream grouping**. MStorm currently adopts a *shuffle grouping* strategy, i.e., tuples are randomly distributed to downstream tasks such that all tasks have identical expected workload in the long run.

4.1.2 Limitation of MStorm

Although MStorm makes an important first step towards a successful design of a mobile distributed stream processing system, *its unawareness of system resource utilization and mobile network's characteristics* lead to the suboptimal behaviors that prevent it from achieving a good performance. In the following sections, we present three experiments that show the inefficiency of the current MStorm system.

4.1.2.1 Resource-unaware configuration

We refer to the configuration as *the users' preference of parallelism for each component* and *the system's initial configuration of the number of executors at each device*. In MStorm, the system configures the number of executors at each device based on its CPU cores. However, since users may run other applications on the mobile device, the available CPU resources depend not only on the CPU cores but also on the current utilization. Configuring the number of executors without an accurate estimation of current resource utilization may lead to performance bottleneck, where the heavily-used nodes may be assigned identical number of executors compared to the idle ones.

We demonstrate this inefficiency through a sample application shown in Figure 4.3, which consists of 4 tasks (T1 - T4). Four Google Nexus 5 mobile devices M1 - M4 form a cluster and MStorm configures identical number of executors per node. Due to *round robin* task assignment, which we discuss later, as it is also inefficient, each mobile device needs to execute one task.

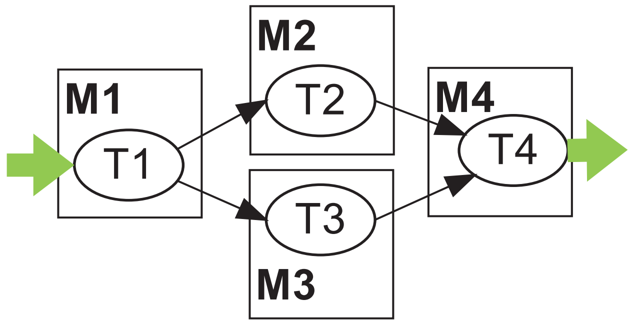
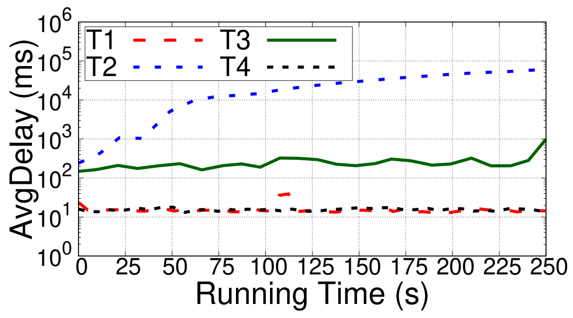
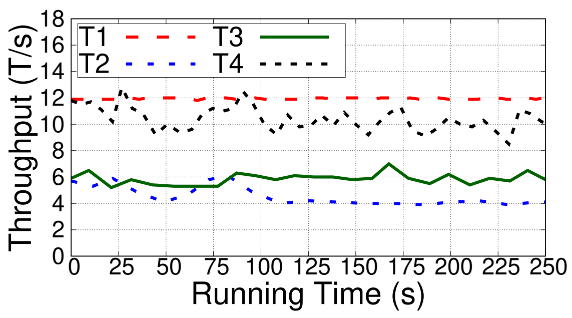


Figure 4.3: Sample application that demonstrates the inefficiency of resource-unaware configuration and stream grouping.



(a) Delay



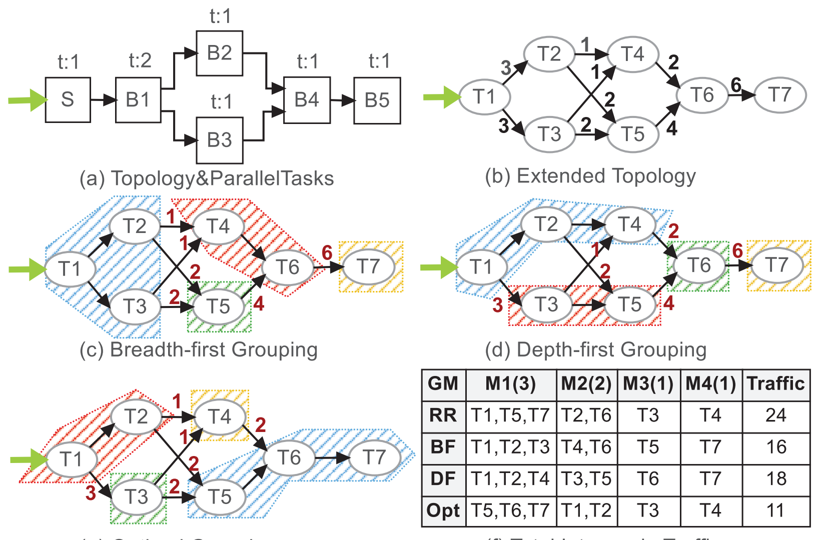
(b) Throughput

Figure 4.4: The results of sample application that demonstrate poor performance of resource-unaware configuration.

Consider the case when M2 is actively used by other user's application, MStorm fails to adapt to this situation and results in poor performance as shown in Figure 4.4. We generate a stream of data at 12 tuples per second (T/s) and measure the delay and throughput of each task. As we can see, the delay for T2 increases from 100ms to 100s because of queuing after the system runs for 250s. This is unacceptable for any real-time mobile application. Besides, the overall throughput (10T/s) is smaller than the input rate (12T/s) due to the bottleneck at T2. This leads to increasing queues in the system.

4.1.2.2 Traffic-unaware task assignment

As mentioned earlier, MStorm uses a *round robin* task assignment strategy, i.e., it sequentially assigns tasks to each mobile device until all tasks are assigned. Although it is easy to implement, it



(a) Topology&ParallelTasks (b) Extended Topology

(c) Breadth-first Grouping (d) Depth-first Grouping

(e) Optimal Grouping

GM	M1(3)	M2(2)	M3(1)	M4(1)	Traffic
RR	T1,T5,T7	T2,T6	T3	T4	24
BF	T1,T2,T3	T4,T6	T5	T7	16
DF	T1,T2,T4	T3,T5	T6	T7	18
Opt	T5,T6,T7	T1,T2	T3	T4	11

(f) Total Inter-node Traffics

Figure 4.5: The inter-device traffic incurred by different task assignment methods.

does not minimize the inter-device traffic, which leads to a higher delay and energy consumption. To reduce the inter-device traffic, an intuitive idea is to put as many tasks as possible on the same node. This leads to our two preliminary attempts for a more efficient task scheduling algorithm, namely breadth and depth-first scheduling, respectively. The breadth (depth) first scheduling works as follows: at first, sort the mobile device based on the number of configured executors; then assign tasks to the same mobile device in a breadth (depth) first order when traversing the extended topology, until the assigned tasks reach its capacity or there is no more task to assign. However, we reveal by the following example that, even if the breadth (depth) first scheduling reduces some inter-device traffic, their performance are still much worse than the optimal schedule.

Figure 4.5 (a) and (b) represent the topology and extended topology of an application. The edge weights in the extended topology represent the total traffic from task to task during a period ΔT. All tasks are assigned to mobile devices M1, M2, M3, M4 with 3, 2, 1, 1 executors by different scheduling algorithms. Based on the breadth/depth-first scheduling, tasks are assigned to mobile devices as shown in Figure 4.5 (c) and (d). Figure 4.5 (e) represents the optimal scheduling that minimizes the inter-device traffic. Figure 4.5 (f) summarizes the scheduling result and corresponding inter-device traffic for each scheduling algorithm, which shows that the round robin scheduling generates 118% more inter-device traffic than the optimal scheduling, whereas the breadth-first and depth-first scheduling also generates 45%, 64% more inter-device traffic than the optimal. This is because they fail to further distinguish different inter-task traffic and don't assign tasks with large inter-task traffic to the same node.

Moreover, even if we distinguish different inter-task traffic, it is still coarse grained, considering the diversity of wireless links. For example, given two tasks with fixed inter-task traffic, if they are assigned to two nodes with a lower inter-device delay, the total delay will be lower. If they are assigned to two nodes with lower communication power, the total energy consumption for traffic transmission will be less. With round robin or breadth (depth) first scheduling, all these potential chances of improving system performance will be missed.

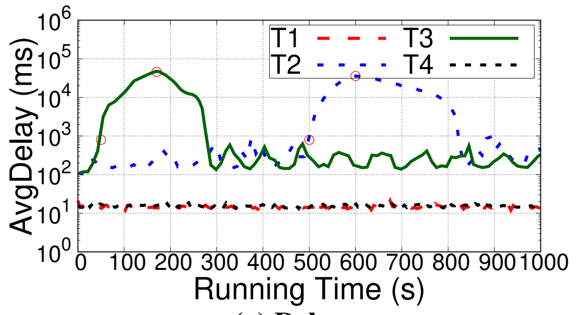
Furthermore, except for minimizing delay and energy consumption, sometimes soldiers or first responders require the system to last longer. To achieve this goal, the tasks need to be assigned based on the remaining battery of each device. With round robin or breadth (depth) first scheduling, the batteries of some devices might be depleted very soon.

4.1.2.3 Resource-unaware stream grouping

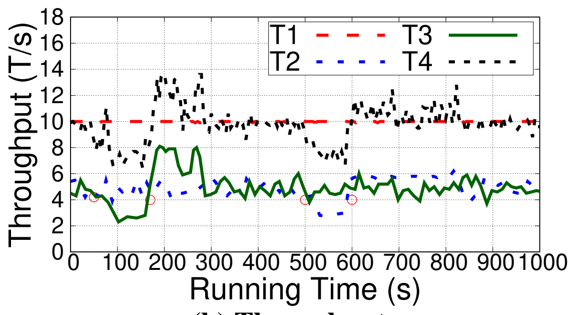
Recall that MStorm adopts a *shuffle grouping* strategy to distribute output tuples to downstream tasks. Although shuffle grouping achieves fairness among tasks in terms of overall workload, we demonstrate that it cannot adapt to the resource fluctuation caused users' own application usage.

We use the same application as shown in Figure 4.3, where all nodes are idle in the beginning. We set the input rate at 10T/s. We run resource-intensive applications on M2 and M3 at 500s and 50s, and close them at 600s and 170s, respectively. Since shuffle grouping assigns output tuples from T1 to T2 and T3 uniformly at random, the arrival rate at T2 and T3 are 5T/s on average, even if M2 and M3 are busy with other applications. As a result, we can see a significant increase in response time in Figure 4.6(a) (from roughly 10^2ms to 10^5ms) and a decrease in throughput in Figure 4.6(b) (from roughly 5T/s to 2T/s), when the resource-intensive application is running.

A key observation from Figure 4.3 is that the throughput of T2 and T3 increases to 6T/s and



(a) Delay



(b) Throughput

Figure 4.6: The results of sample application that demonstrate poor performance of resource-unaware shuffle grouping.

8T/s after the resource-intensive application terminates, which indicates a higher processing capability than the average input rate of 5T/s. Therefore, it is possible to improve the overall throughput by assigning more stream to other more capable nodes, given that we have an accurate online estimation of resource utilization at each node.

4.1.3 Motivation of F-MStorm

One simple approach to solve the problems above is to carve out some static resources on each mobile device dedicated for stream processing [100]. However, unlike servers in the cloud that are uniformly managed by a cluster manger to undertake specific jobs, the limited resources of mobile devices at the edge are shared by both MStorm and other user applications. Those applications might also be resource-intensive and even have higher priorities. It is unreasonable to allow MStorm to take up the valuable resources even when there is no stream processing tasks. Another approach is to apply a pull model [101,102] like most modern cloud computing systems, where the machines ask for tasks when they have free slots. However, this model is not enough for stream processing at the edge as it does not consider other factors like device-to-device delays, energy consumption of inter-device communication and remaining batteries of devices. We argue that, instead of adopting an open-loop task scheduling algorithm which assumes a rather static environment, a feedback-based approach which makes decisions based on the current system state (such as CPU utilization, delays and energy consumption of inter-device communication, remaining batteries, etc.) should be utilized to deal with the dynamic environment at the edge. Two ameliorations proposed for Apache Storm, namely T-Storm [60] and R-Storm [61], have similar insights with F-MStorm. They also adopt a feedback-based approach to improve the system performance. However, neither of them concerns the energy consumption and balance of energy usage, as they are proposed for systems running in the cloud. Nevertheless, for a mobile stream processing system running at the edge, the above two factors directly decide how long a system can last for. It is paramount for F-MStorm to take all these factors into account.

4.2 Design and Implementation of F-MStorm

In this section, we present the design and implementation of F-MStorm. To overcome the inefficiencies presented in the previous section, F-MStorm sets configuration, task scheduling and stream grouping all based on the feedback information. To better present our idea, we use a scenario with m devices and an application with N components. The mathematical model of the problem involves many notations. For readers' convenience, we summarize them in Table 4.1.

4.2.1 Overview

Similar with MStorm, F-MStorm configures the parallelism of each application component based on the user's experience and assigns tasks to the mobile devices through round robin at the beginning. Then, after a "warm-up" period, each mobile device periodically reports the feedback information, including task execution, device resources and network condition, etc., to Nimbus. Based on this feedback, F-MStorm reconfigures tasks for each component, resets available executors for each device and recalculates the best schedule for the whole application. Because of the dynamic processing workload and changing environment, the best task schedule might change from time to time. However, the new best schedule sometimes only achieve a small performance improvement than the previous one. In such case, it is not beneficial to switch to the new schedule, considering the rescheduling overhead and system stability. To deal with this issue, we propose several reschedule conditions. If none of these conditions are met, the system just keeps the origin schedule; otherwise, the reschedule takes place. Except for reporting to Nimbus, each downstream

Table 4.1: Main notations used in F-MStorm.

Notation	Description
i	Index of component, $i = 1, ..., N$
j	Index of task, $j = 1, ..., n$. n varies with configurations
k	Index of mobile device, $k = 1, ..., m$
$A(i)$	Task set of component i
$c(j)$	Component that task j belongs to
$v(j)$	Device that task j is assigned to
W_i	Expected CPU usage of component i
I_i	Expected input rate of component i
O_i	Expected output rate of component i
w_j	CPU usage (MHz) of task j
l_j	Waiting queue length at task j
λ_j	Input rate (T/s) of task j
μ_j	Processing rate (T/s) at task j
$t_{jj'}$	Output rate (T/s) from task j to j'
$s_{jj'}$	Average tuple size (bit) from task j to j'
f_k	Single core frequency (MHz) of device k
c_k	The number of CPU cores of device k
u_k	Total CPU usage (MHz) of device k
r_k	Available CPU resource (MHz) at device k
$d_{kk'}$	Communication delay (ms) from device k to k'
b_k	Remaining battery at device k
e_k^t	Energy consumption (nJ) per bit for Tx at device k
e_k^r	Energy consumption (nJ) per bit for Rx at device k
$\boldsymbol{P}$	Vector: parallel task number for each component
$\boldsymbol{E}$	Vector: available executors for each mobile device
$\boldsymbol{B}$	Vector: remaining battery for each mobile device
$\boldsymbol{T}$	Matrix: average output rate from task to task
$\boldsymbol{S}$	Matrix: average tuple size from task to task
$\boldsymbol{D}$	Matrix: communication delay from device to device
$\boldsymbol{Q}$	Matrix: energy/bit for transmitting between devices
$\boldsymbol{X}$	Matrix: task assignment to mobile devices
ΔT	Period that mobile devices report to Nimbus
Δt	Period that tasks report to upstream tasks

task needs to report the execution information to the upstream tasks periodically. The upstream tasks then distribute the output to the downstream tasks based on the feedback information.

4.2.2 Status Report

In F-MStorm, each mobile device reports the following information to Nimbus periodically (every ΔT):

- w_j: the CPU usage (in MHz) of task j obtained from */proc/stat* and */proc/stat/pid/task/tid/stat* [60].

- l_j: the queue length at task j, i.e., the number of stream tuples that are waiting to be processed.
- λ_j and μ_j: the average input and processing rate (in T/s) of task j during ΔT.
- $t_{jj'}$: the output rate (in T/s) from task j to j' during ΔT.
- $s_{jj'}$: the average tuple size (bit) from task j to j' during ΔT. It is defined as the ratio between the total tuple data size and the total number of tuples from task j to j'.
- r_k: the available CPU resource (in MHz) at device k, defined as $r_k = f_k \cdot c_k - (u_k - \sum_{v(j)=k} w_j)$, where f_k is the single core frequency, c_k is the number of cores, u_k is the current CPU usage at device k, and $\sum_{v(j)=k} w_j$ is the total CPU usage of current F-MStorm tasks at device k.
- $d_{kk'}$: device-to-device communication delay (in ms) from device k to k'.
- b_k: remaining battery (in mAh) at device k.
- e_k^t and e_k^r: energy consumption per bit (nJ/bit) for tuple transmission (Tx) and reception (Rx). They can be estimated based on throughput [103], which we obtained from *DeviceBandwidth-Sampler* [104].

Nimbus maintains a moving average for each status, that is, $V = \delta * V_{old} + (1 - \delta) * V_{new}$, where V_{old} is the old value stored at Nimbus, V_{new} is the new feedback value, and $0 \leq \delta \leq 1$ is a factor used to indicate how the status depends on the history.

It deserves to be mentioned that, periodically reporting and updating these system statuses might cause some extra communication and computing overhead. However, compared with the communication traffic size and processing workload of stream data, the overhead is negligible.

4.2.3 Feedback Based Configuration (FBC)

Based on the feedback, Nimbus calculates the following vectors to reconfigure the system: $\boldsymbol{P} = [P_i]_{i=1}^N$, $\boldsymbol{E} = [E_k]_{k=1}^m$. P_i represents the number of parallel tasks of component i and E_k represents the available executors of each mobile device k. They are calculated by equation $P_i = \lceil \frac{W_i}{\mathcal{R}} \rceil$ and $E_k = \lfloor \frac{r_k}{\mathcal{R}} \rfloor$, where W_i is the expected CPU usage of component i, r_k is the available

CPU resource at device k and $\mathcal{R}$ represents the computing resource of each executor. The ceiling and floor functions are used to leave some margins for device resource fluctuation. $\mathcal{R}$ is calculated by the following equations:

$$\begin{cases} \mathcal{R} &= \min\{\max\{\mathcal{R}_l, \mathcal{R}_e\}, \mathcal{R}_u\} \\ \mathcal{R}_l &= \eta_l * \min_k\{f_k\} \\ \mathcal{R}_e &= \min_{i,k}\{W_i, \frac{r_k}{c_k}\} \\ \mathcal{R}_u &= \eta_u * \min_k\{f_k\} \end{cases} \tag{4.1}$$

where η_l and η_u ($0 \leq \eta_l \leq \eta_u \leq 1$) are parameters to control the lower and upper bound of an executor's resource. The intuitions are as follows. First, the resource of an executor is mostly determined by the CPU usage of components and the available resource of mobile devices. Based on this, we can calculate a basic version of executor resource, namely $\mathcal{R}_e$. Then, to make full use of the CPU resource, a single executor should not occupy more resource than a CPU core. Therefore, we need to set an upper bound $\mathcal{R}_u$ for the executor resource. On the other hand, the resource of an executor should not be too little, otherwise a mobile device will be configured with too many executors, which incurs a lot of OS scheduling overheads. Therefore, we also need to add a lower bound $\mathcal{R}_l$ for the executor resource.

4.2.4 Feedback Based Assignment (FBA)

We formulate the task assignment in F-MStorm as a mixed-integer quadratic programming (MIQP) and solve it by a genetic algorithm. Moreover, to ensure the system stability, we propose 4 reschedule conditions to avoid frequent reschedules.

4.2.4.1 Problem Formulation

Let matrix $\boldsymbol{T} = [T_{jj'}]_{n\times n}$ represent the expected tuple output rate from task to task, with $T_{jj'} = \frac{I_i/P_i}{\sum_{j'} t_{jj'}} * t_{jj'}$, where $i = c(j)$ is the component that task j belongs to, I_i is the expected input rate of component i. I_i/P_i represents the expected tuple input rate of each task. Let matrix $\boldsymbol{S} = [s_{jj'}]_{n\times n}$ represent the measured average tuple size from task to task and let matrix $\boldsymbol{D} = [d_{kk'}]_{m\times m}$ represent the communication delay between mobile devices. Let matrix $\boldsymbol{Q} = [Q_{kk'}]_{m\times m}$ represent the energy per bit for communication from device k to k', where $Q_{kk'} = e^t_k + e^r_{k'}$. We denote the decision variables for the task assignment as a n-by-m 0-1 matrix $\boldsymbol{X}$, with $X_{jk} = 1$ representing that task j is assigned to device k.

Our objective is to minimize the average *end-to-end delay and energy consumption* for each tuple while ensuring the load balance in energy consumption. The end-to-end delay consists of processing delay, queuing delay and communication delay. The energy consumption consists of processing energy and communication energy. Since the system we care is homogeneous in executor's processing speed and energy consumption model, the way we assign tasks will not affect the end-to-end processing delay, queuing delay and processing energy. Therefore, the objective is reduced to minimize the average *end-to-end communication delay and communication energy consumption* while ensuring the load balance in energy consumption. We formulate the problem as:

$$\underset{\boldsymbol{X}}{\text{minimize}} \quad F = \alpha * \frac{g_d}{g_d^{max}} + \beta * \frac{g_q}{g_q^{max}} + \gamma * \frac{g_b}{g_b^{max}} \tag{4.2}$$

$$\begin{aligned} \text{s.t.} \quad & \forall j, \sum_{k=1}^{m} X_{jk} = 1 \\ & \forall k, \sum_{j=1}^{n} X_{jk} \le E_k \\ & \forall j,k,\ X_{jk} \in \{0,1\} \end{aligned} \tag{4.3}$$

where g_d is the average communication delay, g_q is the average communication energy consumption and g_b is the load balance index. g_d^{max}, g_q^{max}, g_b^{max} represent the maximum g_d, g_q and g_b that are used to unify the units. $\alpha, \beta, \gamma \in [0,1]$ are customized according to the user's preference.

g_d is calculated as follows. Given the task-to-task output rate matrix $\boldsymbol{T}$ and task assignment matrix $\boldsymbol{X}$, we can obtain matrix $\boldsymbol{T'} = \Delta T \cdot \boldsymbol{TX}$, where element T'_{jk} represents the total number of tuples output by j from device $v(j)$ to k during ΔT. Similarly, we can obtain matrix $\boldsymbol{D'} = \boldsymbol{XD}$, where element D'_{jk} represents the communication delay from $v(j)$ to k. With $\boldsymbol{T'}$ and $\boldsymbol{D'}$, we can obtain matrix $\boldsymbol{M^d} = \boldsymbol{T'} \odot \boldsymbol{D'}$, where element M^d_{jk} represents the total communication delay of tuples output by task j from $v(j)$ to k during ΔT, and $\odot$ represents Hadamard product.

$\lambda = \sum_{j \in A(1)} \mu_j$ represents the total output rate of the spout. Then, the average communication delay of each tuple is calculated as:

$$
\begin{aligned}
g_d = \frac{\sum_{j,k} \boldsymbol{M^d_{jk}}}{\lambda \Delta T} &= \frac{\sum_{j,k} \Delta T \cdot (\boldsymbol{TX} \odot \boldsymbol{XD})_{jk}}{\lambda \Delta T} \\
&= \frac{\sum_{j,k} (\boldsymbol{TX} \odot \boldsymbol{XD})_{jk}}{\lambda}
\end{aligned}
\tag{4.4}
$$

Next, we consider the calculation of g_q. Similar to the communication delay, we utilize matrix $\boldsymbol{T''} = \Delta T \cdot (\boldsymbol{T} \odot \boldsymbol{S})\boldsymbol{X}$ to represent the total traffic data size output by task j from device $v(j)$ to k in ΔT, matrix $\boldsymbol{Q'} = \boldsymbol{XQ}$ to represent the energy consumption per bit for tuple transmission by Wi-Fi from device $v(j)$ to k. Thereby, the total energy consumption for tuple transmission from device $v(j)$ to k in ΔT can be represented as matrix $\boldsymbol{M^q} = \boldsymbol{T''} \odot \boldsymbol{Q'}$. Then, the power of tuple transmission is calculated as:

$$
\begin{aligned}
g_q = \frac{\sum_{j,k} \boldsymbol{M^q_{jk}}}{\Delta T} &= \frac{\sum_{j,k} \Delta T \cdot ((\boldsymbol{T} \odot \boldsymbol{S})\boldsymbol{X} \odot \boldsymbol{XQ})_{jk}}{\Delta T} \\
&= \sum_{j,k} ((\boldsymbol{T} \odot \boldsymbol{S})\boldsymbol{X} \odot \boldsymbol{XQ})_{jk}
\end{aligned}
\tag{4.5}
$$

Finally, g_b is calculated as follows:

$$
g_b = \sum_{k=1}^{m} (\sum_{j=1}^{n} X_{jk} - \frac{b_k}{\sum_{k=1}^{m} b_k} * n)^2 \tag{4.6}
$$

where b_k is the remaining battery of device k. The effect of this item is intuitive: when two devices posses the same available CPU resources, the computation tasks should be assigned to the one with more remaining battery to prolong the lifetime of the whole system.

4.2.4.2 Genetic Algorithm-based Solution

The aforementioned problem is typically solved by a CPLEX solver [105]. However, based on our experimental results, it takes 77 seconds on average to solve a moderately sized (e.g., 15 nodes and 15 tasks) problem on a desktop, which is impractical for real time applications on mobile platforms. To deal with this issue, we implement an approximation algorithm (Algorithm 1), which returns a near-optimal solution (within 5% of the optimal) in less than $1\mathrm{s}$ for the same problem. Algorithm 1 is based on a "GeneTaskAlloc" procedure that implements the Genetic Algorithm (GA) to solve the optimization problems.

Algorithm 1 works as follows. Notice that, F, g_d^{max}, g_q^{max} and g_b^{max} share the same constraints as shown in Equation 4.3. Therefore, they can be solved by GeneTaskAlloc with different objective functions and goals, i.e. $\max$ or $\min$. We first solve the optimization problem to get g_d^{max}, g_q^{max} and g_b^{max} (line 1 - 9). Then, we use g_d^{max}, g_q^{max} and g_b^{max} to construct the final objective function F (line 10), and call GeneTaskAlloc again to get the final solution (line 11).

The GeneTaskAlloc procedure, which takes a fitness function and an optimization type as input, maintains an iterative process containing the following operations [106]: *SelectParents*, which selects parents from all candidate schedules with the probability proportional to the fitness function value; *GenerateOffspring*, which generates children schedules with parent schedules by uniform crossover; *Mutate*, which chooses a certain number of rows randomly from the schedule matrix according to the mutate rate and changes the position of 1 randomly; *Recombination*, which goes through a schedule matrix row by row and replaces the original schedule with a better one by exchanging adjacent two rows; *FilterOffSpring*, which filters the offspring schedules that do not satisfy the constraints; *SelectPopulations*, which selects a fixed number of populations from the current available schedules according to the fitness function value; *SelectBestSchedules*, which selects the best schedule in terms of the fitness function value. When the iteration times reach the previously set threshold, the procedure will exit with the current best schedule.

4.2.4.3 Reschedule Condition

Sometimes, the new schedules achieve small performance improvement, and switching to them will actually hurt the performance, considering the rescheduling overhead and system stability. To avoid such unnecessary reschedules, we propose the following reschedule conditions:

- It is the first time that the system gets feedback and do reschedule.

Algorithm 1: ApproxTaskSchedulingAlg()

```
Input  : $\boldsymbol{T}$, $\boldsymbol{D}$, $\boldsymbol{S}$, $\boldsymbol{Q}$, $\boldsymbol{B}$, $\alpha$ ,$\beta$, $\gamma$, $\lambda$
Output: Task schedule matrix $\boldsymbol{X}$
1  if $\alpha \neq 0$ then
2      $g_d \leftarrow$ Equation 4.4
3      $g_d^{max} \leftarrow$ GeneTaskAlloc$(g_d, max).value$
4  if $\beta \neq 0$ then
5      $g_q \leftarrow$ Equation 4.5
6      $g_q^{max} \leftarrow$ GeneTaskAlloc$(g_q, max).value$
7  if $\gamma \neq 0$ then
8      $g_b \leftarrow$ Equation 4.6
9      $g_b^{max} \leftarrow$ GeneTaskAlloc$(g_b, max).value$
10 $F \leftarrow$ Equation 4.2
11 $\boldsymbol{X} \leftarrow$ GeneTaskAlloc$(F, min).solution$
12 return $\boldsymbol{X}$
```

- The average end-to-end delay exceeds a threshold τ.
- The input rate of any component i exceeds a threshold times the output, i.e., $\exists i, \sum_{j:c(j)=i} \lambda_j > \sigma * \sum_{j:c(j)=i} \mu_j$.
- The metric of old schedule exceeds a threshold times the metric of new schedule, i.e., $F(\boldsymbol{X}) > \xi * F(\boldsymbol{X}_{new})$, where F is the objective function in Equation 4.2.

When any of the above conditions is met, the task reschedule will take place. To guarantee consistent processing, before switching to a new schedule, the spout of the old schedule will stop pulling stream from the data source and the old schedule will continue running for a while until all the remaining tuples in the system are processed.

4.2.5 Feedback Based Grouping (FBG)

Except for reporting to Nimbus, each mobile device also reports the execution information (such as task input and output rates, queue lengths, etc.) to the upstream tasks periodically (every Δt). The upstream tasks then direct the output stream tuples to the downstream task with the "Least Expected Waiting Time (LEWT)", which is calculated as follows.

Without loss of generality, we assume task j which belongs to the component i receives the task execution report from task j', which belongs to the downstream component i', at time t_0. Then, at time t which satisfies $t - t_0 < \Delta t$, if task j chooses to send an output stream tuple to j', the "Expected Waiting Time (EWT)" for this tuple at task j' can be calculated by

$$EWT_{j'} = \frac{[(\lambda_{j'} - t_{jj'} - \mu_{j'})(t - t_0) + l_{j'} + \Delta l]^+ + 1}{\mu_{j'}} \tag{4.7}$$

where $(\lambda_{j'} - t_{jj'})$ is the input rate from other tasks to task j', $\mu_{j'}$ is the processing rate and $l_{j'}$ is the waiting queue length at task j'. Δl is the number of tuples sent to j' from task j in the past $(t - t_0)$ time. Function $[x]^+ = \max(0, x)$. Since we consider *applications in which tuples have no temporal and spatial relations with each other*, according to LEWT stream grouping, an output tuple of j should be sent to task $j' \in A(i')$ that achieves the minimum $EWT_{j'}$.

4.3 Evaluation

In this section, we at first introduce a benchmark application for evaluating the system performance. Then, we present the experimental setup and analysis for the evaluation results.

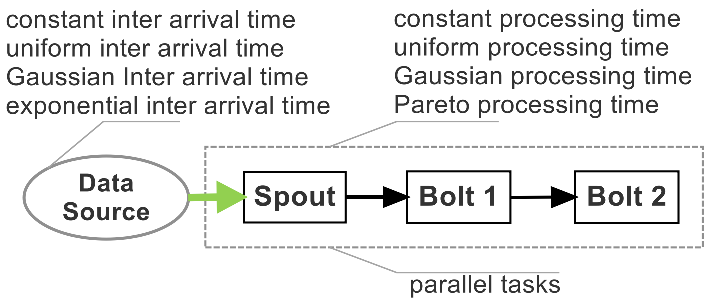


Figure 4.7: The benchmark application for evaluating F-MStorm performance.

4.3.1 Benchmark Application

In order to thoroughly test the performance of F-MStorm, we developed a benchmark application shown in Figure 4.7. The application consists of a data source and three components, i.e., spout, bolt1 and bolt2. To precisely control the input, we let the data source directly generate tuples with the same size and different inter-arrival time (IAT). The IAT can be configured with different distributions, including constant, uniform (UR), Gaussian (GA) and exponential (EP) distributions. The processing time (PT) for each tuple can be configured with different distributions as well, which includes constant, uniform, Gaussian and Pareto (PA) distributions. For the ease of presentation, we denote spout, bolt1, bolt2 by C1, C2, C3 in the rest of the chapter.

4.3.2 Experimental Setup

We conduct experiments on three Google Nexus 5 phones running Android 6.0 and a laptop configured as WiFi hotspot. Each Nexus 5 phone has a 4-core CPU and each core is set to run at 1574MHz. All phones are connected to the WiFi hotspot. We run F-MStorm or MStorm on these phones and set system parameters $\Delta T = 15s$, $\Delta t = 5s$, $\delta = 0.4$, $\tau = 2s$, $\sigma = 1.1$ and $\xi = 1.5$. To simulate the resource fluctuation when users invoke other applications during the execution of F-MStorm or MStorm, we developed a resource-intensive disturbance application. We define the light, medium, and heavy disturbance as the scenarios where we run this disturbance application with 10%, 80% and 270% CPU utilization respectively (the total is 400%).

We thoroughly evaluate the system through different types of experiments. First, we run the benchmark application with constant IAT (10T/s) and different constant workloads (defined below) to demonstrate the efficiency of FBC and FBA. We define the light, medium, and heavy constant workload (LCW, MCW, and HCW respectively) as the scenarios where the processing time ratios between C1, C2, and C3 are 1:1:1, 1:15:1, and 1:25:15, respectively. Then, we show the efficiency of FBG by running experiments with constant IAT and MCW, with light, medium, and heavy disturbances respectively. We further evaluate the system's overall performance and compare it with two state-of-the-art solutions (T-Storm [60] and R-Storm [61] on MStorm) under different constant input speeds, IAT distributions and processing time distributions. Finally, we investigate the overall performance when the CPU frequency of phones decreases due to overheating.

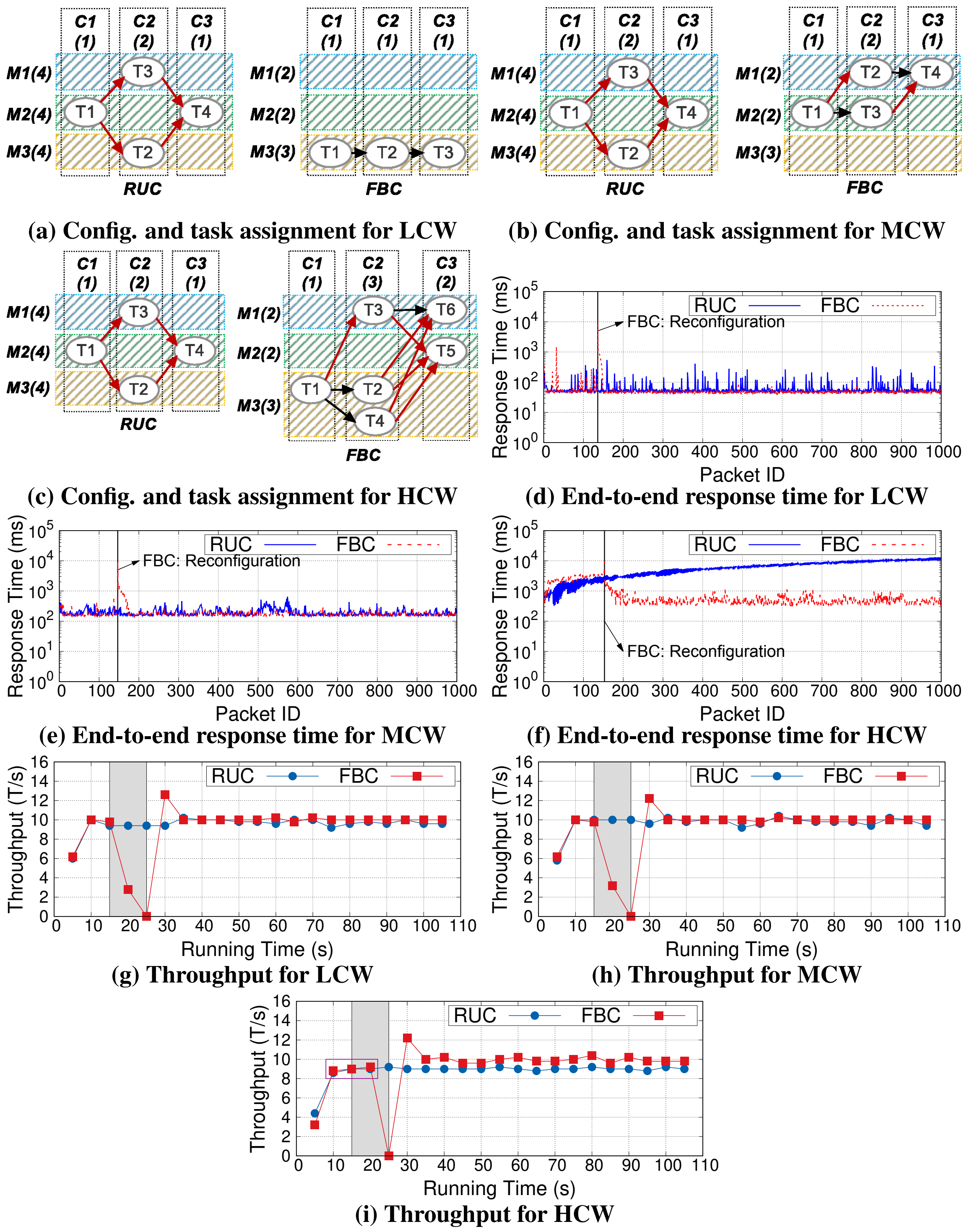
For most experiments, we are interested in the response time (RT, in ms) and throughput (T/s). For FBA experiments specifically, we care about the communication delay (ms) and communication power (mW). In most experiments, we set $\alpha = 0.5$, $\beta = 0.5$, $\gamma = 0$ because delay and energy consumption are more important for us. However, to show that our system also provides configuration for load balance, we run extra FBA experiments with $\alpha = 0.1$, $\beta = 0.1$, $\gamma = 0.8$ and compare its performance with the $\alpha = 0.5$, $\beta = 0.5$, $\gamma = 0$ case.

4.3.3 Evaluation Results

The Effects of FBC: Figure 4.8 (a)-(c) show the results of both configuration and task assignment when we have the same constant input rate and different workloads. The left side of each figure shows the result of resource-unaware configuration (RUC) and round robin task assignment, while the right side shows the result of FBC and FBA. The number of executors are shown beside each mobile device. With RUC, the number of parallel tasks for C1, C2, C3 are always configured as 1:2:1, regardless of the workload. The number of executors is configured as 4 for all devices, which equals the number of CPU cores. On the other hand, FBC reconfigures the number of parallel tasks for the light workload as 1:1:1 and for the heavy workload as 1:3:2. The number of executors for M1, M2, M3 are reconfigured as 2, 2, 3 based on the feedback.

Figure 4.8 (d)-(f) show the results of response time. Since FBC reconfigures the number of tasks of LCW as 3, it allows FBA to assign all tasks to a single node M3. This significantly reduces the communication delay and hence the total response time (Figure 4.8 (d)). In Figure 4.8 (f), FBC increases the number of tasks for C2 and C3 of HCW to 3 and 2 respectively, which eliminates the congestion and reduces the response time.

Figure 4.8 (g)-(i) show the results of throughput. For LCW and MCW, both RUC and FBC have throughput equal to the input rate; whereas for HCW, RUC has throughput lower than the input rate because there are not enough parallel tasks for C2 and C3. On the other hand, the throughput of FBC in HCW equals the input speed as the parallel tasks for C2 and C3 are reconfigured. An interesting observation is about MCW, where FBC doesn't change the original parallel tasks, but the FBA algorithm reduces the inter-device traffic by half, just as shown in Figure 4.8 (b). However, since the communication delay is much less than the computing delay in MCW, the end-to-end response time only decreases a little bit in Figure 4.8 (e).



(a) Config. and task assignment for LCW (b) Config. and task assignment for MCW

(c) Config. and task assignment for HCW (d) End-to-end response time for LCW

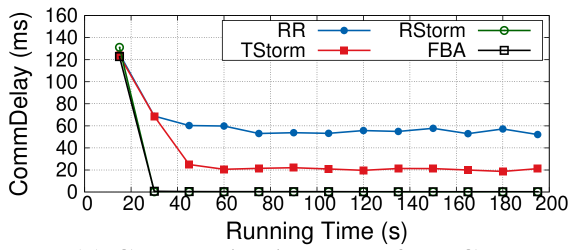
(e) End-to-end response time for MCW (f) End-to-end response time for HCW

(g) Throughput for LCW (h) Throughput for MCW

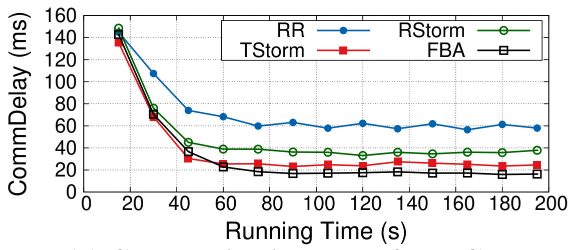
(i) Throughput for HCW

Figure 4.8: Evaluation results of F-MStorm for constant inter arrival and processing time.

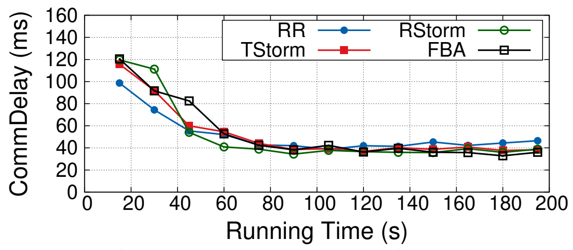
The Effects of FBA: To isolate the effects of task assignment algorithms, when we compare FBA with round robin (RR), TStorm and RStorm, we let RR use the correct parallelism configuration



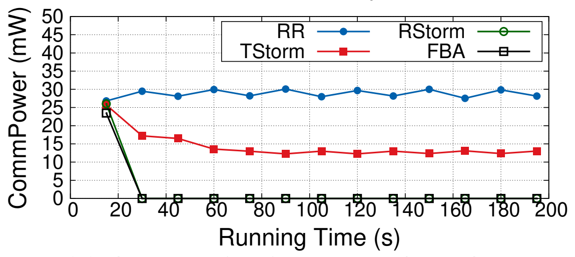
(a) Communication delay for LCW



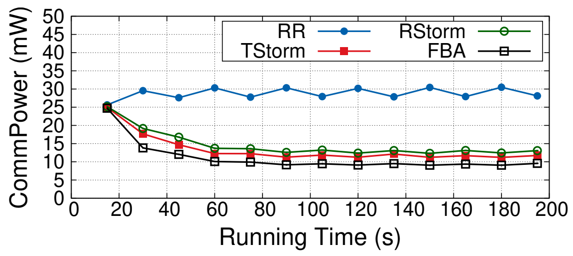
(b) Communication delay for MCW



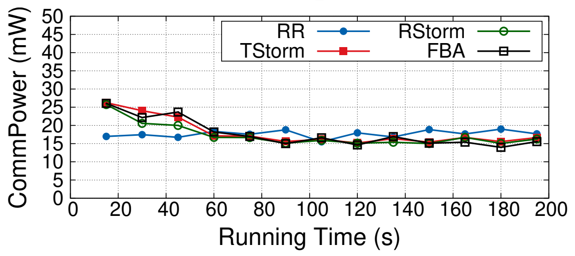
(c) Communication delay for HCW



(d) Communication power for LCW



(e) Communication power for MCW



(f) Communication power for HCW

Figure 4.9: Evaluation results for round robin and feedback based task assignment under different constant workloads.

from the beginning, i.e., 1:1:1 for LCW, 1:2:1 for MCW and 1:3:2 for HCW (see Figure 4.8 (a) - (c)). In contrast, in FBA, TStorm and RStorm, the parallelism configuration is reconfigured based on the feedback.

Figure 4.9 shows the experimental results, where CommDelay is the average end-to-end communication delay and CommPower is the power consumed for transmitting tuples. For LCW, FBA and RStorm achieve much lower communication delay (Figure 4.9 (a)) and communication power (Figure 4.9 (d)) than RR by assigning all tasks to a single device. Meanwhile, TStorm achieves better performance than RR but worse performance than FBA and RStorm, because its task scheduling algorithm not only tries to reduce the inter-device traffic but also aims at achieving load balance between different nodes. This prevents assigning all tasks to a single node and therefore incurs extra communication delay and energy consumption. For MCW, FBA reduces the communication

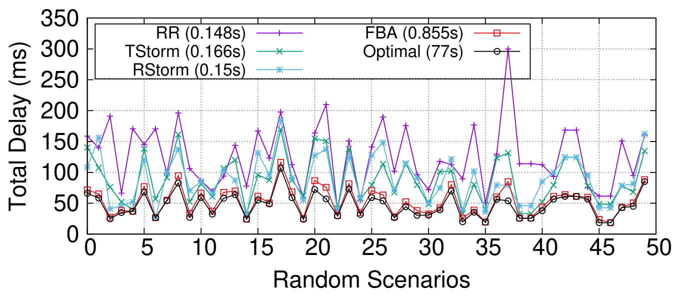


Figure 4.10: Comparison of executing time for different task assignment algorithms.

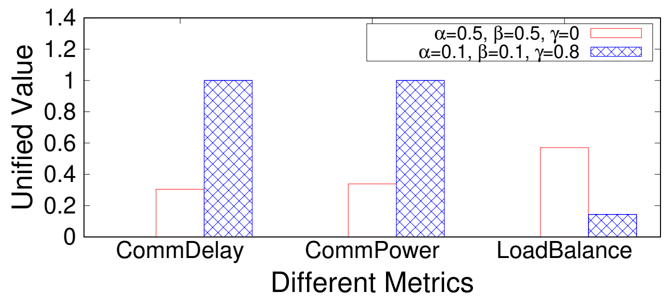
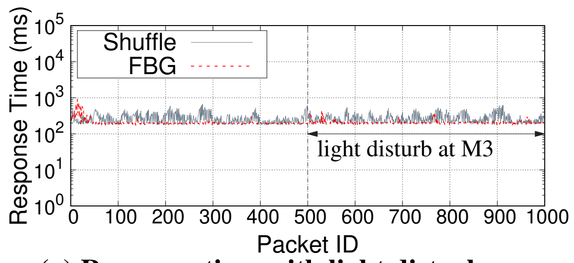


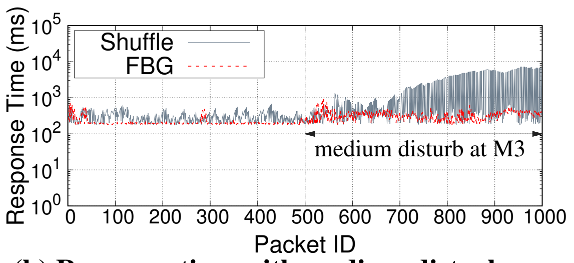
Figure 4.11: Comparison of metrics with different α, β, γ settings.

delay (Figure 4.9 (b)) of RR, RStorm and TStorm by 68%, 50% and 26% respectively and the communication power (Figure 4.9 (e)) of RR, RStorm and TStorm by 64%, 23% and 17% respectively. This demonstrates the importance of distinguishing different inter-device communication, in terms of traffic size, communication delay and power. As for HCW, FBA achieves similar communication delay and power as TStorm and RStorm while a little bit lower communication delay and power than RR (Figure 4.9 (c) and (f)). This is because, for this specific heavy workload, the FBA scheduling algorithm happens to achieve the same schedule as TStorm and RStorm, and a little bit better schedule as RR. It is interesting that, the performance of FBA, TStorm and RStorm is worse than RR until they do rescheduling. This is because, RR utilizes the optimal parallelism at the beginning, while FBA, TStorm and RStorm reschedule to the optimal parallelism based on the feedback by themselves.

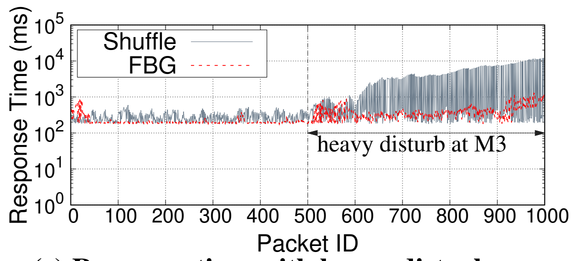
In order to throughly compare the performance of FBA, RR, TStorm and RStorm in more general cases, we did 50 extra simulations with different mobile device capacities, application



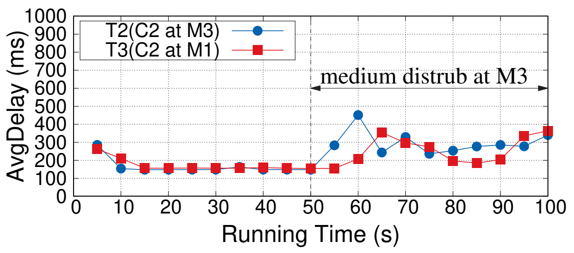
(a) Response time with light disturbance



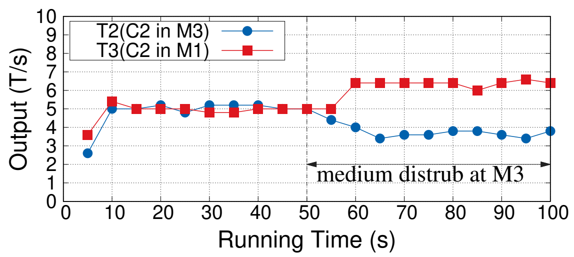
(b) Response time with medium disturbance



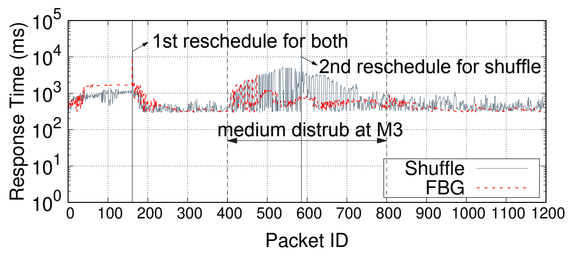
(c) Response time with heavy disturbance



(d) Dealy of C2 with medium disturbance



(e) Output of C2 with medium disturbance

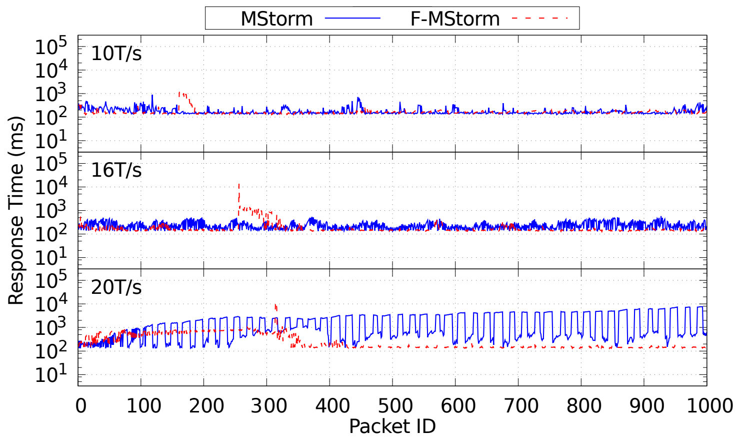


(f) FBG reduces rescheduling frequency

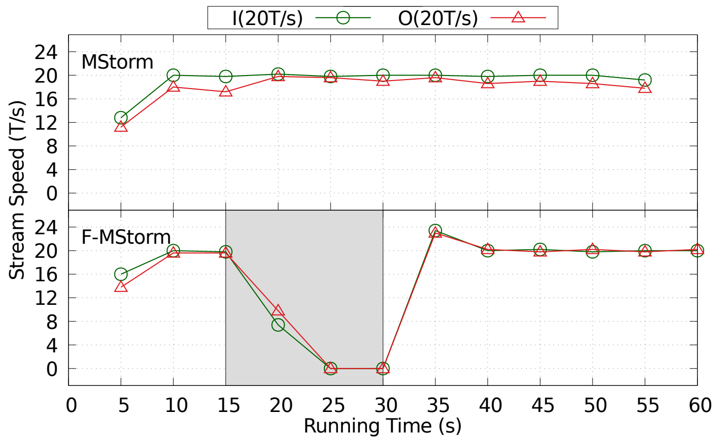
Figure 4.12: Shuffle and feedback based stream grouping under different disturbance.

topologies, number of tasks, task-to-task delays and traffic sizes. The scale the experiment is about 15x15, which means 15 tasks are assigned to 15 mobile devices. Figure 4.10 shows the experimental results. As we can observe, the performance of FBA is always close to (within 5% of) the optimal schedule, while the performance of TStorm and RStorm is unstable, ranging from near the optimal to more than 3x times worse. In terms of running time, TStorm and RStorm can run as fast as RR (150ms), and our FBA takes about 850ms, while the optimal scheduling takes 77s. Taking both performance and running time into account, our FBA is more practical for a real system deployment.

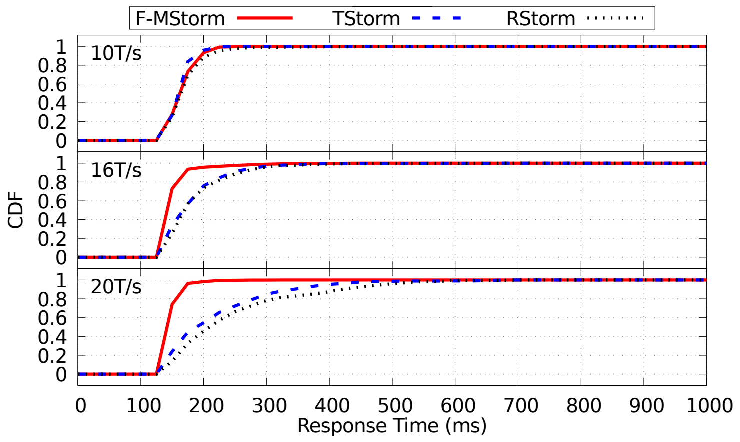
In order to show that our system provides flexible configuration for load balance, we run extra FBA experiments with MCW and set $\alpha = 0.1$, $\beta = 0.1$, $\gamma = 0.8$. As shown in Figure 4.11, a large γ chooses a schedule with better load balance but higher communication delay and power. This is



(a) RT for Diff. Input Speeds

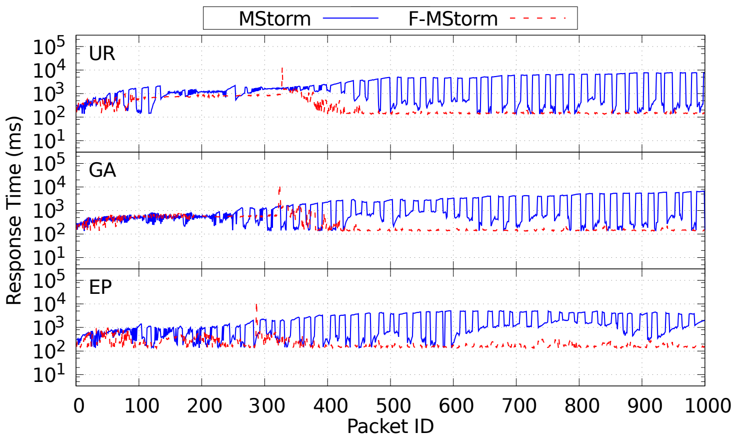


(b) Input/output for Diff. Input Speeds

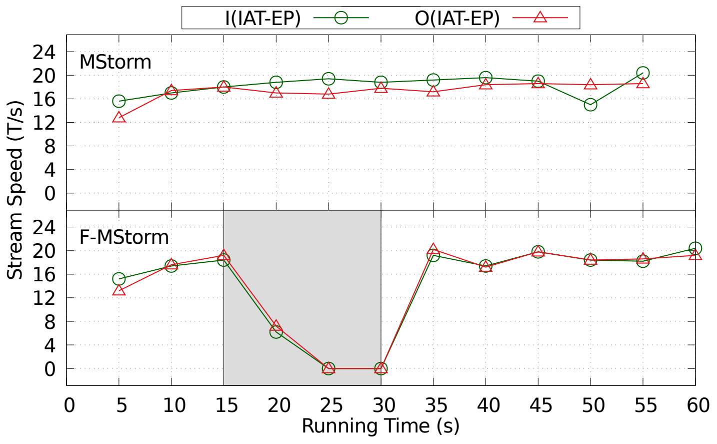


(c) RT CDF comparison for Diff. Input Speeds

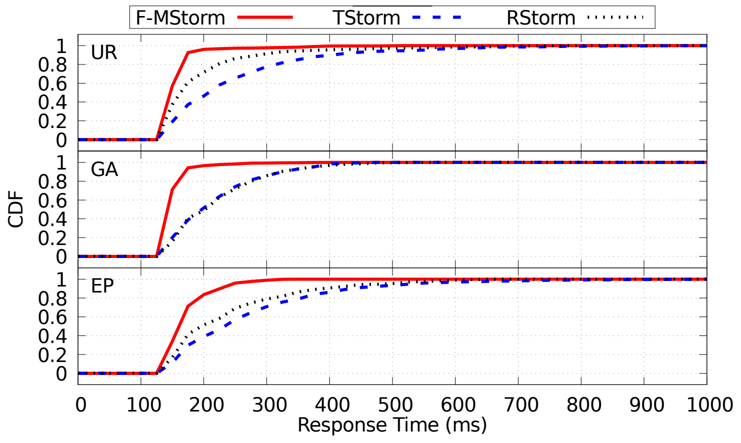
Figure 4.13: Evaluation results of F-MStorm when we vary the constant input speeds.



(a) RT for Input with Diff. IAT

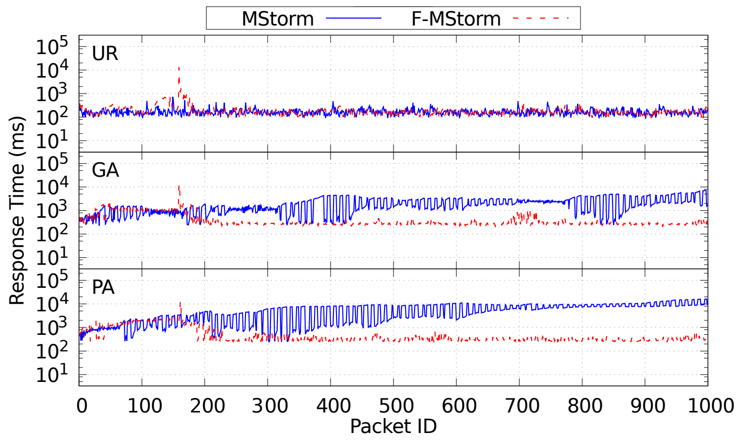


(b) Input/output for Input with Exp. IAT

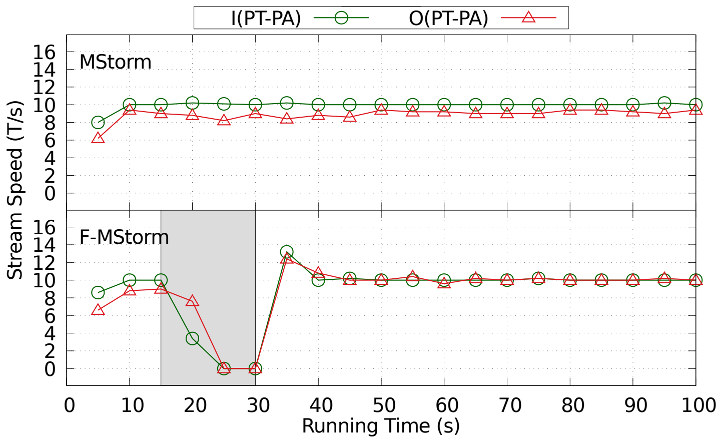


(c) RT CDF comparison for input with Diff. IAT

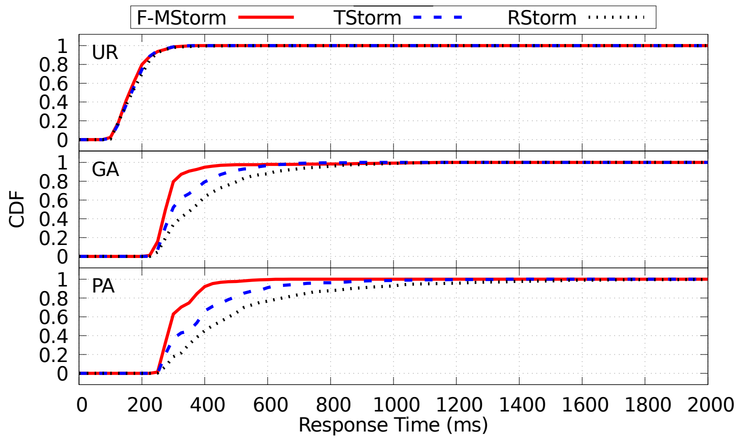
Figure 4.14: Evaluation results of F-MStorm when we vary the inter arrival time of input.



(a) RT for Input with Diff. PT



(b) Input/output for Input with Pareto PT



(c) RT CDF comparison for Input with Diff. PT

Figure 4.15: Evaluation results of F-MStorm when we vary the processing time of input.

because, if a schedule achieves a good load balance, those tasks are very likely to be assigned to different nodes, which will definitely increase the communication delay and energy consumption.

The Effects of FBG: In order to show the effectiveness of FBG, we turn off FBC and FBA. Tasks T1, T2, T3, T4 are assigned to M2, M3, M1, M2 respectively (see Figure 4.8 (a) left). We start the disturbance application at 50s on M3. As we can observe from Figure 4.12 (a-c): The light disturbance does not impact the system performance. However, the medium and heavy disturbance lead to an increasing response time for Shuffle. This is because T1 completely ignores the disturbance at M3 and still sends the output tuples to T2 and T3 randomly. This causes congestion at T2 and leads to increasing delay. On the other hand, FBG relieves the negative impact of disturbance on the performance. As shown in Figure 4.12 (d) and (e), when the disturbance begins, the throughput of T2 begins to decrease and the throughput of T3 begins to increase, while the delay at T2 remains low. This is because T1 sends more output tuples to T3 at M1 after it receives feedback from T2 and T3. We also run experiments with constant IAT and HCW with FBC and FBA turned on. As shown in Figure 4.12 (f), when the medium disturbance occurs, the system with Shuffle grouping has to perform rescheduling to achieve low latency while the system with FBG can avoid rescheduling by directing more stream to the tasks running on the under-loaded devices.

Varying Input Speed: Figure 4.13 (a)-(c) show the results when we adopt the MCW and use different constant input speeds. Figure 4.13 (a) shows the response time. When the input speeds are 10T/s and 16T/s, both MStorm and F-MStorm can achieve a stable low response time. However, when the input speed increases to 20T/s, the response time becomes unstable and keeps increasing. This is because, the input speed exceeds the maximum processing speed and causes congestions at the computation intensive component. In that case, F-MStorm will do reschedule and increase the parallelism for that component, which finally eliminates the congestion and brings the latency back to normal. Figure 4.13 (b) shows the results for throughput with constant input at 20T/s. As we can see, MStorm always has a lower output rate than the input rate, while F-MStorm has an output rate equal to the input rate after rescheduling. Figure 4.13 (c) shows the CDF of stable response time in F-MStorm, TStorm and RStorm with input speeds equal to 10T/s, 16T/s and 20T/s respectively.

When the input speed is 10T/s, the response time of F-MStorm, TStorm and RStorm are similar with each other. This is because they adopt similar task assignment, and have similar inter-devices communication delay. However, when the input speed increases to 16T/s, the latency of TStorm and RStorm can be up to 1.5x latency of F-MStorm. And when the input speed increases to 20T/s, the advantage of F-MStorm becomes more obvious, which can be up to 3x faster than TStorm and RStorm. This is because, as the input speed increases, there are more tasks after rescheduling. The inter-device communication among different tasks will increase and the advantages of F-MStorm, which takes inter-device communication diversity into account, will become more obvious.

Varying Inter Arrival Time: Figure 4.14 (a)-(c) show the results when we adopt the MCW and vary the IAT distribution. Figure 4.14 (a) shows the response time. In MStorm, regardless of the IAT distribution, the response time increases with the running time. However, in F-MStorm, although the response time is increasing at the beginning, it goes back to be low after rescheduling. This is because F-MStorm increases the parallelism of the computing-intensive component to eliminate the congestion. The results in Figure 4.14 (b) prove this claim, where the IAT distribution is exponential. MStorm always has a 1T/s lower output rate than the input rate, while F-MStorm has an output rate equal to the input rate after rescheduling. It should be noted that, although the throughputs of MStorm and F-MStorm seem similar, there is a fundamental difference: a congestion happens in MStorm, while no congestion happens after rescheduling in F-MStorm. The delays of MStorm and F-MStorm in Figure 4.14 (a) clearly reflects this phenomenon. Due to space limitation, we omit the throughput results for other distributions because they are similar. Figure 4.14 (c) shows the CDF of the stable response time in F-MStorm, TStorm and RStorm with uniform, Gaussian and exponential IAT distribution, respectively. With uniform IAT distribution, the response time of RStorm can be up to 1/3 shorter than that of TStorm but still up to 2x that of F-MStorm. With Gaussian IAT distribution, the response time in TStorm and RStorm are similar with each other, but they are both up to 2x that of F-MStorm. With exponential IAT distribution, the response time of F-MStorm becomes longer. However, it is up to 1/2 shorter than that of TStorm and RStorm. The mechanism behind is that: more tasks and inter-device communication exist in the Gaussian and exponential IAT distribution cases, so the advantages of F-MStorm, which takes inter-device communication diversity into account becomes more obvious.

Varying Processing Time: Figure 4.15 (a)-(c) show the results when we adopt the constant input rate (10T/s) and vary the processing time. Figure 4.15 (a) shows the response time. As we can see, with uniform distribution of processing time, both MStorm and F-MStorm achieve stable and low latency. However, with Gaussian and Pareto distribution of processing time, F-MStorm achieves much lower and stable response time than MStorm, because F-MStorm increases the parallelism of the computing-intensive component, which eliminates the congestion. Figure 4.15 (b) shows the results for throughput with Pareto distribution. MStorm always achieves 10% lower output rate than the input rate, while F-MStorm achieves an output rate that is the same as the input rate after rescheduling. Due to space limitation, we omit the throughput results for other distributions because they are similar. Figure 4.15 (c) shows the CDF of the stable response time in F-MStorm, TStorm and RStorm with uniform, Gaussian and Pareto distribution of processing time respectively. With uniform distribution of processing time, the response time of F-MStorm, TStorm and RStorm are similar with each other. However, with Gaussian distribution of processing time, TStorm achieves up to 1/4 shorter response time than RStorm but 1/5 longer response time than F-MStorm. With Pareto distribution of processing time, the advantage of F-MStorm becomes more obvious. The mechanism behind is that: more tasks and inter-device communication exist in the Gaussian and Pareto distribution of processing time cases, so the advantages of F-MStorm, which takes inter-device communication diversity into account becomes more obvious.

CPU Frequency Decrease: The CPU overheating protection mechanism on Android Phones will reduce the CPU frequency when its temperature goes too high [107]. The CPU frequency of smart phones we use might drop from 1574MHz to 1190MHz when it is overheating. To compare MStorm and F-MStorm in this situation, we first run the resource-intensive application for a while such that the CPU temperature gets high. Then, we start MStorm/F-MStorm and the benchmark application with constant IAT and workload with processing time ratio 1:25:1. The tasks are assigned to mobile devices as shown in Figure 4.16. In F-MStorm, an initial rescheduling happens

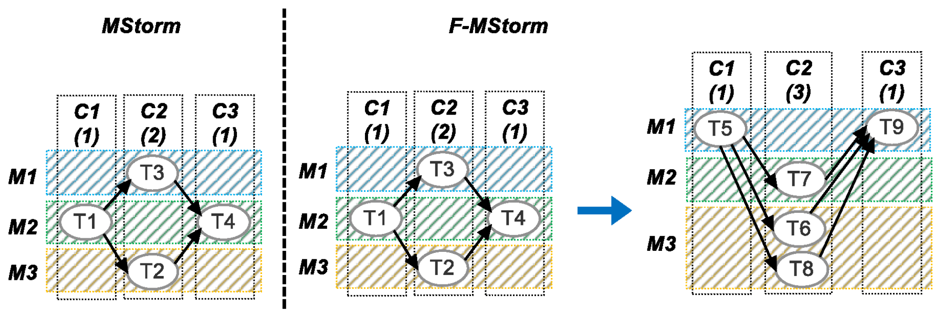
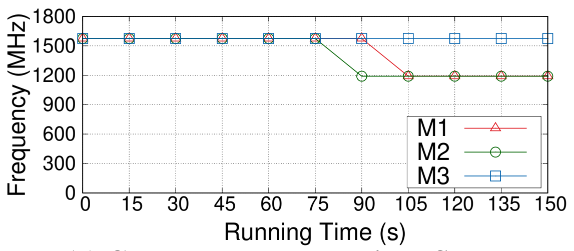


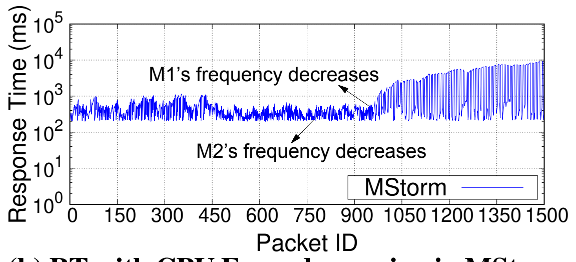
Figure 4.16: Sample example for CPU frequency decrease.

before the CPU frequency drops. The tasks of C1 change from T1 to T5; the tasks of C2 change from T2, T3 to T6, T7, T8; and the tasks of C3 change from C4 to C9.

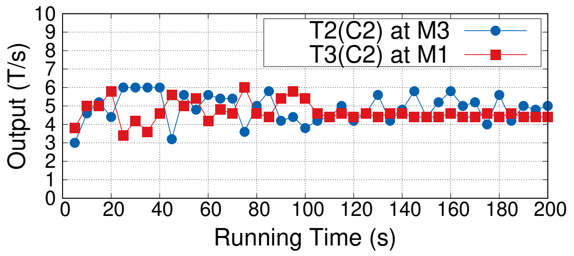
Figure 4.17 (a) and (d) show the time when the CPU frequency of mobile phones drops in MStorm and F-MStorm respectively. We are interested in how the systems may react to it. Figure 4.17 (b) and (c) show the results for MStorm. When the CPU frequency of M2, which runs T1 and T4 (light tasks), drops, the end-to-end response time is not impacted. However, when the CPU frequency of M1, which runs T3 (heavy task), drops, the end-to-end response time increases instantly and the total output rate of C2 (9.5T/s) becomes lower than its input rate (10T/s). On the other hand, Figure 4.17 (e) and (f) shows the results for F-MStorm. In F-MStorm, when the CPU frequency of M1, which runs T5 and T9 (light tasks), drops, the end-to-end response time is not impacted. When the CPU frequency of M2 and M3, which run T6, T7 and T8 (heavy tasks), drops, the end-to-end response time only increases a little bit at the beginning, but soon returns to normal. This is due to the fact that, when the CPU frequency of M2 drops, T5 directs more stream tuples to T8; and when the CPU frequency of M3 drops, T5 directs more stream tuples to T7. The total output rate of C2 keeps at 10T/s.



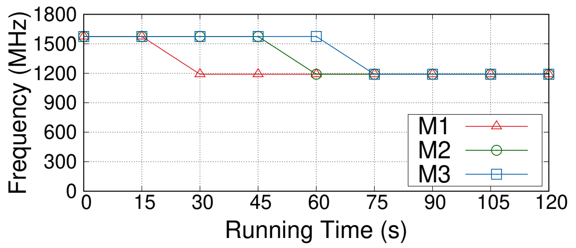
(a) CPU Freq. decreases in MStorm



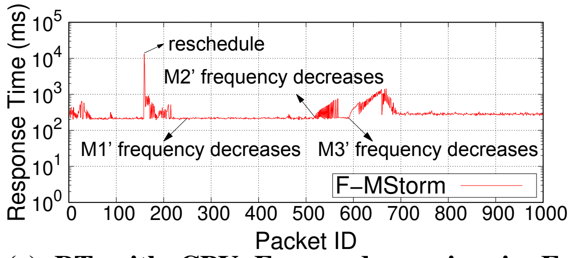
(b) RT with CPU Freq. decreasing in MStorm



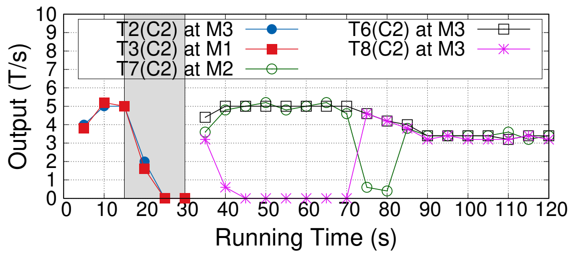
(c) C2's Output with CPU Freq. decreasing in MStorm



(d) CPU Freq. decreases in F-MStorm



(e) RT with CPU Freq. decreasing in F-MStorm



(f) C2's Output with CPU Freq. decreasing in F-MStorm

Figure 4.17: How MStorm and F-MStorm deal with CPU frequency decrease.

5. R-MSTORM: RESILIENT MOBILE STREAM PROCESSING*

In an MSP system, mobile devices are interconnected with wireless networks such as WiFi, Bluetooth or LTE. Different from wired networks, the communication quality of wireless networks are severely affected by the environment condition, signal attenuation, channel contention, etc. Therefore, the connections among devices are with dynamic latency, fluctuating bandwidth and intermittent connectivity. Such properties make MSP at the edge very inefficient: user applications may have a long end-to-end response time, achieve a low throughput and suffer uncontrollable data loss during the processing. Previous research [8, 108] designs the basic framework and resolves the computing resource fluctuation in MSP, respectively. However, they both assume the wireless connections among devices are always on, which is not a fact. A recent work [71] deals with the network fluctuation in MSP by increasing the parallelism of application components. However, it overlooks the effects of task assignment on the MSP resilience. Different from the existing works, in this chapter, we present R-MStorm, a MSP system which aims to improve the resilience of MSP at the edge from the following perspectives:

1) Task Assignment. In R-MStorm, an application is modeled as a Directed Acyclic Graph (DAG), where a path from the source to the destination is defined as a *stream path*. Since each application component may spawn several parallel tasks, there can be multiple stream paths at the same time. To achieve resilient MSP, on one hand, all stream paths should be less affected by the network fluctuation; on the other hand, even when the network is completely down, there should be at least one stream path to continue the stream processing. To this end, R-MStorm uses *Resilient Task Assignment (RTA)* to assign tasks of an application to mobile devices. First, RTA chooses mobile devices with higher availability to run tasks, so that the availability of all stream paths improves. Then, RTA assigns tasks of the same component to different devices to increase the diversity of physical stream paths. Finally, RTA ensures that there is at least one stream path on the submitter

*Reprinted with permission from "R-mstorm: A Resilient Mobile Stream Processing System for Dynamic Edge Networks" by Mengyuan Chao and Radu Stoleru in Proceedings of 2020 IEEE International Conference on Fog Computing (ICFC 2020), pages 1–9, Sydney, Australia, 2020, Copyright 2020 by IEEE.

device, so that even if all the nearby mobile devices become disconnected, the stream processing can still continue.

2) Stream Grouping. To efficiently divide stream among multiple stream paths, R-MStorm uses *Adaptive Stream Grouping (ASG)*. The core idea is: instead of sending the output of a task to downstream tasks in a random way, the upstream task sends the output to a downstream task with the minimum weight. The weight is calculated based on the link quality [109], queue lengths and input, output and processing speed at each task. By applying ASG, R-MStorm schedules the transmission and processing workload among multiple stream paths in an adaptive manner to avoid congestion and improve throughput.

3) Stream Selection. Based on ASG, if the counterpart tasks (defined in Section 5.1.2) of a task become temporarily unreachable, its upstream task(s) will send more stream data to this task. However, since each task has a limited processing capability, this might cause congestion and degrade the overall performance. To alleviate the impact of congestion, R-MStorm uses an approximate computing mechanism called *Adaptive Stream Selection (ASS)*. ASS enables a congested task to skip some stream units to reduce the processing workload while slightly sacrificing the processing accuracy temporarily.

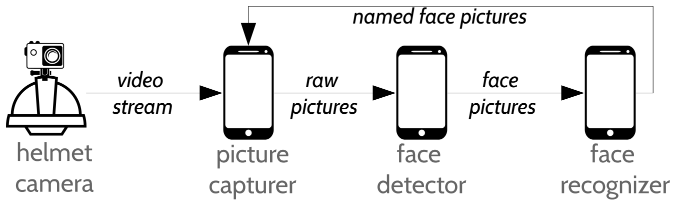
We implement R-MStorm on a cluster of mobile phones and evaluate it through a video face recognition App under different network conditions. The experimental results show that, compared with the baseline approaches, R-MStorm achieves up to 1.5x higher throughput, 75% lower response time, at a cost of 3.3% temporary accuracy loss.

5.1 Mobile Stream Processing

This section describes the application scenarios, application models, and challenges of MSP.

5.1.1 Application Scenarios

With increasing processing capabilities of mobile devices, MSP, which combines mobile devices via wireless networks, provides an innovative way to perform computation-intensive stream processing autonomously at the edge. This paradigm is paramount for some specific application



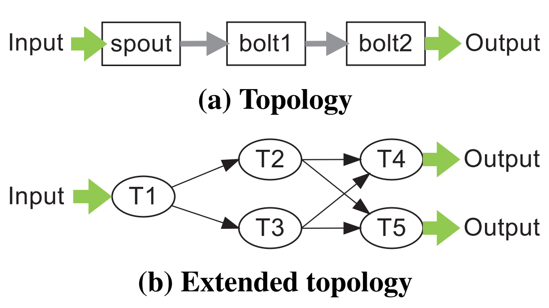
(c) Face recognition App for disaster response

Figure 5.1: A disaster response team enters a city after disaster to search and rescue victims stuck in ruins. They are equipped with a wireless manpack, a helmet camera and multiple mobile phones. A face recognition App assists them to automatically recognize victims during the rescue process.

scenarios, where the Internet connectivity is unstable, unavailable, unsafe or costly.

As an example, consider a victim searching and recognition application for disaster response in Figure 5.1. A disaster response team enters a city after disaster (Figure 5.1(a)) to perform a “wide area search”. Since there is no Internet connectivity, they are equipped with a helmet camera for video recording, a wireless manpack for communication and several mobile phones for data processing (Figure 5.1(b)). Their task is to search victims stuck in ruins and recognize them with a face recognition App.

As shown in Figure 5.1(c), this face recognition App for disaster response consists of three components: 1) a *picture capturer* which continuously pulls stream from the helmet camera and divides the stream into groups of pictures; 2) a *face detector* which detects faces from the input pictures and outputs them to 3) a *face recognizer*. The face recognizer matches the input faces with a database and outputs the recognized faces with names. Since these components are computation intensive, it is impossible to execute them on a single mobile device to achieve high through-



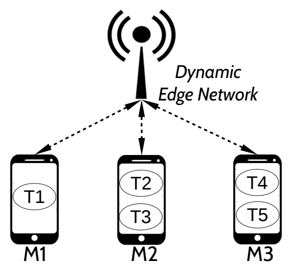


Figure 5.2: A sample MSP application.

put. Traditional cloud-based approaches, which offload computation-intensive components to the cloud, fail to work when there is no Internet connectivity. In this case, these disaster responders can perform MSP by connecting their mobile devices via wireless networks at the edge, such that the face recognition App can still achieve a good throughput.

5.1.2 Application Models

In R-MStorm, an MSP application is modeled as a directed acyclic graph (DAG) called *topology*. Each topology contains two types of nodes, i.e., *spout* and *bolt*. A spout continuously partitions the input stream into small *tuples* and sends them to downstream bolts. A bolt processes tuples from spout or upstream bolts, and outputs the processed tuples to downstream bolts for further processing. We refer to a spout or a bolt as an *application component* (or simply component). A directed edge between two components indicates the dependency: for a tuple, a component cannot start execution until its precedent component completes. Each component can spawn multiple parallel tasks to perform the same operations. These parallel tasks are *counterpart tasks* of each other. If we expand all the components with parallel tasks, we obtain another DAG called *extended topology*, which shows the actual data flow among tasks. To improve the MSP throughput, tasks of extended topology are assigned to different mobile devices. By choosing one task from each component, we obtain a combination of tasks that we name a *stream path*. There are multiple stream paths in an extended topology. Given a specific stream path, we define the combination of mobile devices that run tasks of the stream path as a *physical stream path*.

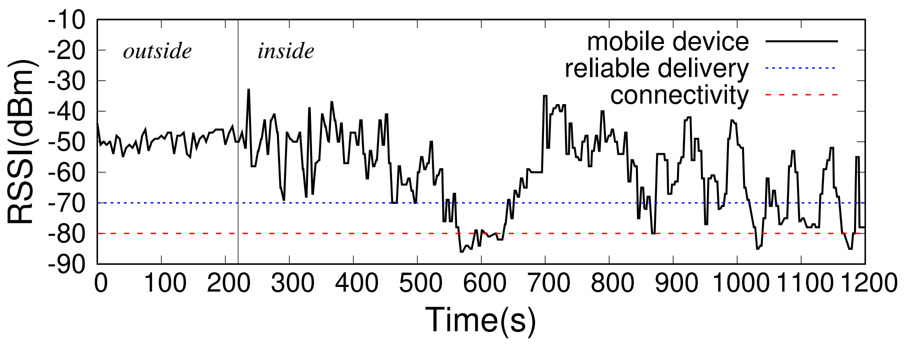


Figure 5.3: RSSI of a mobile device during disaster response.

Figure 5.2 shows the (a) topology, (b) extended topology and (c) distributed execution of an MSP application. This application contains one spout and two bolts, i.e., bolt1, bolt2. Each bolt further contains two parallel tasks, i.e., T_2, T_3 for $bolt_1$ and T_4, T_5 for $bolt_2$. All tasks are assigned to mobile devices M_1, M_2, M_3 connected by dynamic wireless networks. T_1-T_2-T_4 is a stream path and M_1-M_2-M_3 is its physical stream path.

5.1.3 Challenges

In MSP systems, mobile devices are interconnected with wireless networks such as WiFi, Bluetooth or LTE. Different from wired networks, the communication quality of wireless networks is severely affected by signal shielding, attenuation and interference, hidden and exposed terminal problems, channel contention, etc. These factors may bring unstable latency, dynamic bandwidth, asymmetric and intermittent connectivity to the connections among mobile devices, which makes MSP at the edge challenging: user applications may experience a long end-to-end response time, achieve a low throughput and suffer uncontrollable data loss during the processing.

As an example, Figure 5.3 shows the Received Signal Strength Indicator (RSSI) value of a mobile phone carried by a disaster responder during a wide area search exercise. Due to node mobility, signal shielding and attenuation, RSSI value of this phone changes and sometimes goes below the minimum RSSI value required by a stream application. This makes MSP tasks running on this device suffer congestion and unpredictable delay. To solve this issue, we need a resilient MSP system that can adapt to dynamic edge networks.

5.2 System Overview

We present R-MStorm, a resilient MSP system running on a group of mobile devices at the edge. First, we introduce the system architecture of R-MStorm, which shows how mobile devices collaborate to perform MSP. Then, we introduce the R-MStorm client in detail, which consists of several important modules that enable resilient MSP in dynamic edge networks.

5.2.1 System architecture

As shown in Figure 5.4, an R-MStorm system consists of a server and multiple clients. The server is responsible for managing the clients and the clients are in charge of running the concrete stream processing tasks. First, the clients which have spare resources send a request to the server to join. Then, the server receiving the "joining" requests organizes the clients into one cluster. For security reason, only clients belonging to the same cluster can share resource with each other. In a real deployment, one server can manage multiple clusters.

When a user needs to perform stream processing, he/she at first sends an extended topology of the application to the R-MStorm server via a supervisor at the client. When the server receives the extended topology, it assigns corresponding tasks to multiple clients through a resilient task assignment manager. This manager is very important for achieving resilient MSP, because its RTA algorithm directly determines the availability and diversity of stream paths, which makes stream processing resilient to the network change and failure. When a client gets assigned tasks from the server, it runs each task in an executor of the worker and establishes connections to the upstream and downstream tasks. The task execution and system states are monitored by a system state manager. It maintains states of downstream tasks based on downstream reports. As we describe later, multiple modules of R-MStorm client depend on these states to achieve resilient MSP.

During the stream processing, an R-MStorm client might get temporarily disconnected from the R-MStorm server because of losses in communication. To solve this issue, the client will keep reconnecting until the connection is restored or the maximum number of attempts has been reached. It should be noted that, a client-to-server disconnection does not affect the stream pro-

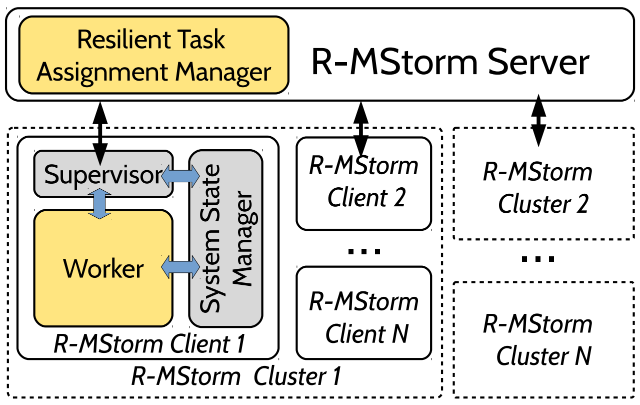


Figure 5.4: System architecture of R-MStorm.

cessing performance, because all stream transmissions are through connections between clients. Nevertheless, if two clients get disconnected, the situation is different: one client not only needs to keep reconnecting to the other, it also needs to process the output in a resilient manner. In the following, we describe the detailed design of R-MStorm client to show how it deals with the client-to-client disconnection.

5.2.2 Client architecture

As mentioned earlier, an R-MStorm client mainly consists of a supervisor for communication with the server, a system state manager for recording and reporting the system states, and a worker for executing stream processing tasks. In this section, we mainly focus on the detailed design of worker, as it directly affects the resilience of MSP. As shown in Figure 5.5, the worker of R-MStorm client maintains an executor pool for running stream processing tasks, a group of input/output queues for holding the input/output tuples of tasks, an adaptive stream dispatcher which dispatches the output of each task to downstream tasks and an adaptive stream selector which skips some tuples to avoid congestion and get short response time.

When tuples from the upstream tasks arrive at an R-MStorm client, they are pushed into the input queue of a task. Then, based on the system states (e.g., input and output speeds of that task), the adaptive stream selector determines whether an input tuple should be processed or not. The

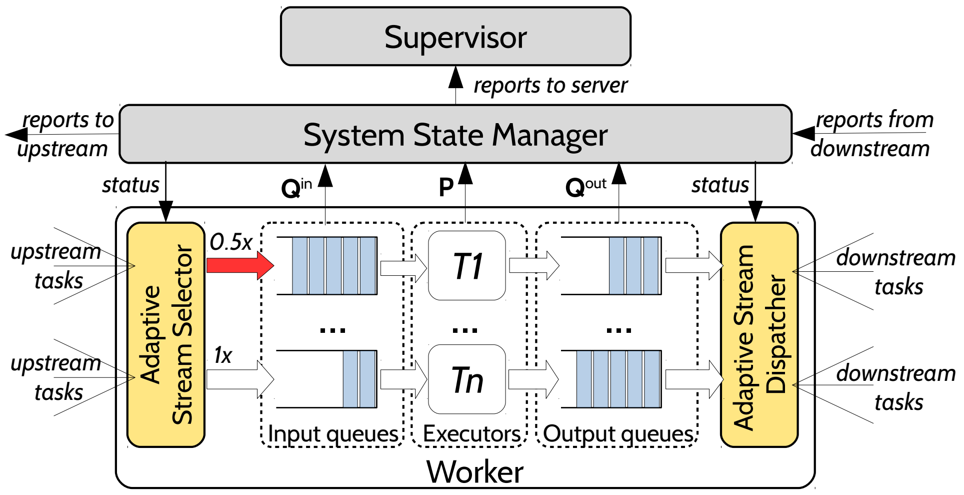


Figure 5.5: Client architecture of R-MStorm.

selected tuples are sent to the task executor for processing and the output tuples are sent to the output queue. The adaptive stream dispatcher pulls tuples from the output queue and dispatches them to the downstream tasks based on an adaptive stream grouping method. When the connection to a downstream task is down, the adaptive stream dispatcher will immediately change corresponding states of that task and resend stream tuples to other counterpart tasks.

5.3 Resilient Mobile Stream Processing

The resilience of R-MStorm is implemented by three mechanisms, i.e., resilient task assignment, adaptive stream grouping and adaptive stream selection. This section describes these mechanisms in detail. For convenience, we first list the main notations adopted in this section in Table 5.1.

5.3.1 Resilient Task Assignment

Given a topology consisting of components $C_1, C_2, ..., C_N$, tasks $T_1, T_2, ..., T_n$, and a mobile device cluster that contains devices $M_1, M_2, ..., M_m$, we model the MSP task assignment with matrix $\boldsymbol{X}$, where binary variable $x_{jk} = 1$ if task j is assigned to device k and $x_{jk} = 0$ otherwise. Correspondingly, we model the MSP component assignment with matrix $\boldsymbol{Y}$, where binary variable $y_{ik} = 1$ if any task of component i is assigned to device k and $y_{ik} = 0$ otherwise.

Table 5.1: Main notations adopted in R-MStorm.

Notation	Description
C_i	Component i, where $i = 1, ..., N$
$\Theta(i)$	Task set of component i
T_j	Task j, where $j = 1, ..., n$ and $n = \sum_{i=1}^{N} \|\Theta(i)\|$
$comp(j)$	Component that task j belongs to
λ_j	Input rate of task j
μ_j	Output rate of task j
p_j	Processing rate of task j
$r_{jj'}$	Transmission rate from task j to task j'
q_j^{in}	Input queue length of task j
q_j^{out}	Output queue length of task j
M_k	Mobile device k, where $k = 1, ..., m$
a_k	Availability of device k
e_k	Number of executors at device k
$\boldsymbol{X}$	x_{jk} denotes if task j is assigned to device k
$\boldsymbol{Y}$	y_{ik} denotes if component i has tasks on device k

5.3.1.1 Availability-oriented task assignment

With task assignment $\boldsymbol{X}$, the *availability* of MSP application $A(\boldsymbol{X})$ is defined as the availability of all the devices involved in the processing. In our scenario, the device availability is determined by the wireless connection to the manpack and the connection is affected by the node mobility and other environmental factors. For simplicity, we assume that the availability of each device is independent. With this assumption, we have:

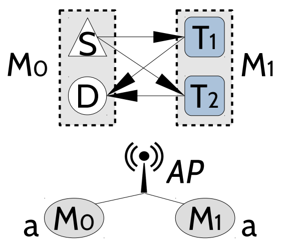
$$A(\boldsymbol{X}) = \prod_{k=1}^{m} a_k^{U_k(\boldsymbol{X})} \tag{5.1}$$

where

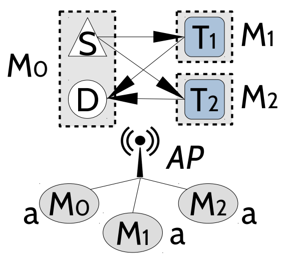
$$U_k(\boldsymbol{X}) = \bigvee_{j=1}^{n} x_{jk} = x_{1k} \vee x_{2k} \vee ... \vee x_{nk} \tag{5.2}$$

represents whether device k is involved in the MSP processing. If $U_k(\boldsymbol{X}) = 1$, the actual availability of device k equals to a_k; otherwise, the actual availability of device k is 1. To avoid dealing with exponentiation, we use the logarithm of the MSP availability as follows:

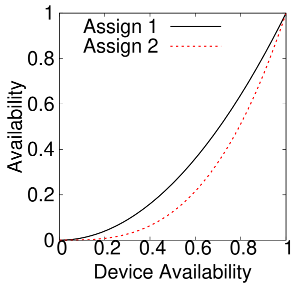
$$\log A(\boldsymbol{X}) = \sum_{k=1}^{m} \log a_k^{U_k(\boldsymbol{X})} = \sum_{k=1}^{m} \log a_k \cdot U_k(\boldsymbol{X}) \tag{5.3}$$



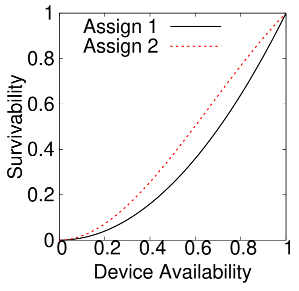
(a) Assign 1



(b) Assign 2



(c) Availability



(d) Survivability

Figure 5.6: A simple example which shows that higher availability does not always mean higher survivability.

From Equation 5.3, we observe that: in order to maximize the availability of an MSP application, we need to: 1) assign tasks to as few devices as possible; 2) assign tasks to devices with higher availability first.

5.3.1.2 Availability vs. Survivability

As we described earlier, a typical MSP application usually has multiple stream paths. Ideally, all the stream paths can work at the same time to improve the processing throughput. However, in dynamic edge networks, this ideal situation does not always happen. Some stream paths may break down when some devices become temporarily unavailable. Nevertheless, for an MSP application that dynamically divides stream onto different physical stream paths, temporary partial availability is acceptable: the remaining physical stream paths can handle the entire stream for a short period of time until the impaired physical stream paths recover. Therefore, in MSP, instead of availability, we care more about *survivability*, which is defined as the probability that at least one physical stream path still exists when some of the involved devices become temporarily unavailable.

With the definition of survivability above, a research question comes up: *Does a task assignment with higher availability always have higher survivability?* Unfortunately, the answer is no. To demonstrate this, we show an example in Figure 5.6, where an application requires to offload one component (two tasks) to other devices to perform MSP. For simplicity, we assume that each device has the same availability a. In Assignment 1, both tasks T_1 and T_2 are assigned to device M_1. The availability of MSP is a^2. Because both M_0 and M_1 are indispensable, the survivability of MSP is a^2 as well. In Assignment 2, task T_1 is assigned to device M_1 and task T_2 is assigned to device M_2. The availability of MSP is a^3, which is lower than a^2 of Assignment 1 as shown in Figure 5.6(c). However, since T_1 and T_2 belong to the same component, one of M_1 and M_2 can be temporarily unavailable. Therefore, the survivability of MSP is $a^3 + 2a^2(1-a) = 2a^2 - a^3$, which is always higher than a^2 of Assignment 1 as shown in Figure 5.6(d).

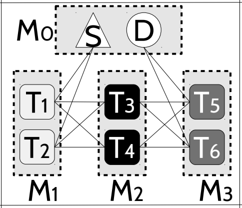
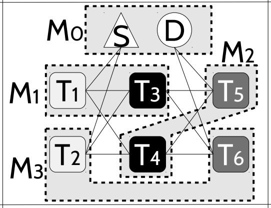
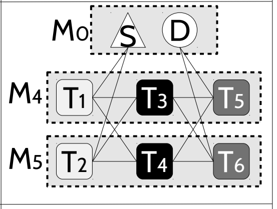
Now that higher availability in MSP does not always mean higher survivability, the second research question becomes: *How to improve the survivability of a task assignment?* Unlike MSP availability that is only determined by the availability of devices involved in the processing, an accurate calculation of MSP survivability is very complicated: it requires to consider all the cases when some involved devices become temporarily unavailable. For a general case which involves many devices, a simple expression for the MSP survivability is impossible. To solve this issue, we leverages simpler metrics to depict the MSP survivability. First, we adopt a metric called component-to-device (CTD) diversity. It measures the number of distinct devices that tasks of a component are assigned to. We observe that, with the same devices, if we increase CTD diversity of each component, survivability of the whole MSP assignment improves as well. The idea behind is intuitive: since tasks of the same component are assigned to different mobile devices, when some devices become unavailable, other devices running counterpart tasks can take over the stream to perform the same processing. With CTD diversity D_i for each component i, we define CTD diversity of assignment $\boldsymbol{X}$ (and its corresponding component assignment $\boldsymbol{Y}$) as:

$$D(\boldsymbol{X}) = D(\boldsymbol{Y}) = \prod_{i=1}^{N} D_i = \prod_{i=1}^{N} \sum_{k=1}^{m} y_{ik} \tag{5.4}$$

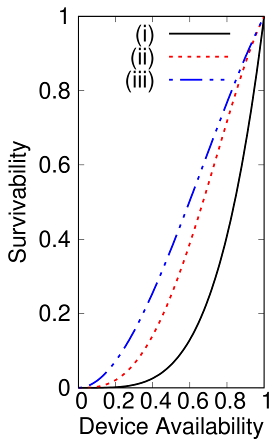
where

$$y_{ik} = \bigvee_{j \in \Theta(i)} x_{jk} \tag{5.5}$$

denotes whether tasks of component i are assigned to device k. We also observe that, with the same CTD diversity, improving MSP availability can increase MSP survivability significantly. Therefore, availability is also an important part of survivability.

#	(i)	(ii)	(iii)
Assignment X			
A(X)	a^4	a^4	a^3
S(X)	a^4	$3a^3$-$2a^4$	$2a^2$-a^3
Dev	4	4	3
D(X)	1x1x1=1	2x2x2=8	2x2x2=8

(a) **Metric comparison of different assignments**

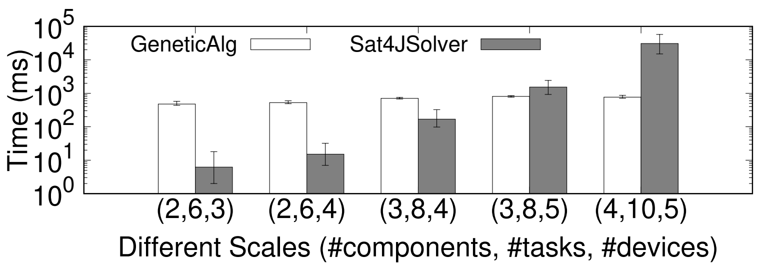


(b) **Survivability**

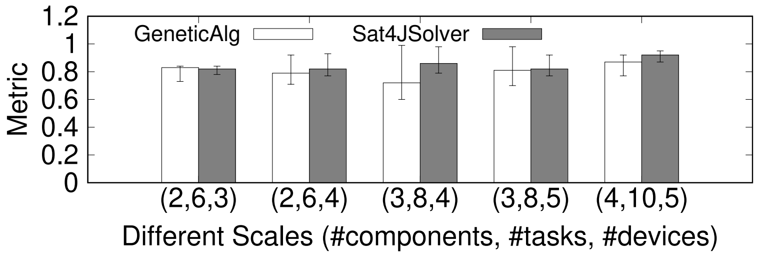
Figure 5.7: Example of how to improve MSP survivability.

Figure 5.7 provides an example to showcase how to improve the MSP survivability of an assignment by increasing its MSP availability and CTD diversity. For simplicity, all devices are assumed to have an identical availability a. Except the source and destination, all other components have two parallel tasks. However, as shown in Figure 5.7(a)(i), increasing parallel tasks for each component does not necessarily improves its CTD diversity: if two tasks of one component are assigned to the same device, its CTD diversity still equals to 1. In Figure 5.7(a)(ii), we run tasks on the same devices as Figure 5.7(a)(i) to keep MSP availability but assign tasks of the same component to different devices. Then, its CTD diversity increases from 1 to 8 and the MSP survivability changes from a^4 to $3a^3 - 2a^4$. In Figure 5.7(a)(iii), we keep CTD diversity as 8 but assign tasks to fewer devices to improve MSP availability from a^4 to a^3. Accordingly, its MSP survivability changes from $3a^3 - 2a^4$ to $2a^2 - a^3$. As shown in Figure 5.7(b), with the higher MSP availability and CTD diversity, Assignment (iii) achieves higher survivability than Assignment (i) and (ii).

Sometimes, the above two metrics might conflict with each other as in Figure 5.6. This leads to a multi-objective optimization (MOO) problem. This problem can be transformed into a single objective problem by using the Simple Additive Weighting (SAW) technique [110]. According to SAW, we can define MSP survivability as a weighted sum of MSP availability and CTD diversity



(a) Computing time comparison



(b) Metric comparison

Figure 5.8: Performance comparison of genetic algorithm and Sat4J.

as follows:

$$S(\boldsymbol{X}) = w_a \frac{\log A(\boldsymbol{X}) - \log A_{min}}{\log A_{max} - \log A_{min}} + w_d \frac{D(\boldsymbol{X}) - D_{min}}{D_{max} - D_{min}} \tag{5.6}$$

where $w_a, w_d \geq 0$, $w_a + w_d = 1$ are weights for availability and CTD diversity, $\log A_{max}$ and $\log A_{min}$ denote the maximum and minimum value of the availability term, D_{max} and D_{min} denote the maximum and minimum value of the CTD diversity term, respectively. With the definition of survivability, we formulate the resilient task assignment with a Nonlinear Pseudo-Boolean Optimization (NPBO) model [111] as:

$$\underset{\boldsymbol{X}}{\text{maximize}} \quad S(\boldsymbol{X}) \tag{5.7}$$

$$\text{subject to:} \quad \sum_{k=1}^{m} x_{jk} = 1 \quad \forall j \tag{5.8}$$

$$\sum_{j=1}^{n} x_{jk} \leq e_k \quad \forall k \tag{5.9}$$

$$x_{jk} \in \{0, 1\} \quad \forall j, k \tag{5.10}$$

where equation (5.8) guarantees that each task is assigned to one device and constraint (5.9) limits the number of tasks assigned to each device based on its available executors. We solve this problem by both a genetic algorithm-based solver (Alg. 2) and a Sat4J [112] solver. We compare the performance of these two solvers at different problem scales in Figure 5.8. The result shows that, the genetic algorithm-based solver achieves a metric value within 5% worse than the Sat4J solver, but its computation time is much shorter as the problem scale goes up. Therefore, in R-MStorm, we adopt the genetic algorithm-based solver to solve the resilient task assignment problem.

Algorithm 2: GeneAlgBasedSolver()

```
Input  : Availability $a_k$, executors $e_k$, constraints $Q$
Output: Task Allocation $\boldsymbol{X}$
1  if $w_a \neq 0$ then
2      $\log A \leftarrow$ Equation 5.3
3      $\log A_{max} \leftarrow$ GeneTaskAlloc($\log A, max, Q$).$value$
4      $\log A_{min} \leftarrow$ GeneTaskAlloc($\log A, min, Q$).$value$
5  if $w_d \neq 0$ then
6      $D \leftarrow$ Equation 5.4
7      $D_{max} \leftarrow$ GeneTaskAlloc($D, max, Q$).$value$
8      $D_{min} \leftarrow$ GeneTaskAlloc($D, min, Q$).$value$
9  $S \leftarrow$ Equation 5.6
10 $\boldsymbol{X} \leftarrow$ GeneTaskAlloc($S, max, Q$).$solution$
11 return $\boldsymbol{X}$
```

Algorithm 2 summarizes the genetic algorithm-based solver, which is based on a "GeneTaskAlloc" procedure described in Algorithm 3. It works as follows: First, it gets the maximum and the minimum values of the availability and diversity by calling "GeneTaskAlloc" with different objective functions and optimization types. Then, based on these parameters, it calls "GeneTaskAlloc" again to find the optimal task assignment which maximizes the survivability.

The "GeneTaskAlloc" procedure, which takes a specific objective function, an optimization type and some constraints as inputs, performs an iterative process containing the following operations: 1) *SelectParents*, which selects a certain number of parents from all candidates based on the objective function; 2) *GenerateChildren*, which generates children assignments by uniform

```
Algorithm 3: GeneTaskAlloc()
   Input  : ObjFunc $F$, OptType $T$, constraints $Q$
   Output: Task allocation $\mathcal{X} = (solution, value)$
 1 $P_s \leftarrow InitPopulation(s)$ // population size $s$
 2 $\mathcal{X} \leftarrow SelectBest(F, T, P_s)$
   // evolve over $n$ generations
 3 for $i \leftarrow 1$ to $n$ do
 4     $P_p \leftarrow SelectParents(F, T, P_s, p)$ // parent size $p$
 5     $P_c \leftarrow GenerateChildren(P_p)$
 6     $P_m \leftarrow P_c$
 7     for $m \in P_m$ do
 8         $m \leftarrow Mutate(m, r)$ // mutation rate $r$
           // recombine every $t$ generations
 9         if $i\%t = 0$ then
10             $m \leftarrow Recombination(m)$
11     $P_o \leftarrow FilterOffspring(Q, P_c \cup P_m)$
12     $P_s \leftarrow SelectPopulations(F, T, P_p \cup P_o, s)$
13     $\mathcal{X} \leftarrow SelectBest(F, T, \{\mathcal{X}\} \cup P_s)$
14 return $\mathcal{X}$
```

crossover; 3) *Mutate*, which chooses some rows of an assignment based on a certain rate and changes values at some positions randomly; 4) *Recombination*, which exchanges two rows of a task assignment to achieve a better objective function value; 5) *FilterOffSpring*, which deletes the task assignments that do not meet the constraints; 6) *SelectPopulations*, which selects new populations from the available candidates based on the objection function values; 7) *SelectBest*, which selects the assignment that achieves the optimal objective function value. After iterating above operations for a certain number of times, the procedure finally returns an assignment which achieves an optimal objective function value with best efforts.

5.3.2 Adaptive Stream Processing

In MSP, each component has multiple tasks that are assigned to different devices to improve the throughput and resilience. These tasks jointly handle the output tuples from the upstream. To ensure good performance in a highly dynamic environment, when an upstream task chooses the downstream task to send its outputs, it needs to take into account both the transmitting rate to and processing rate at different downstream tasks, rather than evenly or randomly distributing the output to downstream tasks as in the cloud [58]. Additionally, when some tasks of a component become unreachable, the upstream tasks will send all the outputs to the remaining counterpart tasks. This may cause congestion and performance degradation at those tasks. To solve this issue, a stream selection mechanism needs to be applied to selectively drop some stream tuples under control to ensure smooth stream processing while keeping the processing accuracy at a certain level. In the following, we describe the above two mechanisms in detail.

5.3.2.1 Adaptive Stream Grouping (ASG)

The core idea of ASG is to send the output to the downstream task with the minimum weight, where the weight of each downstream task j' for task j is calculated as follows:

$$weight_{j'} = \frac{\lambda_{j'} \times q_{j'}^{in} \times q_{j'}^{out}}{r_{jj'} \times p_{j'} \times \mu_{j'}} \tag{5.11}$$

where $q_{j'}^{in}$ and $q_{j'}^{out}$ represent the input and output queues of task j', $\lambda_{j'}$, $p_{j'}$ and $\mu_{j'}$ represents the input, processing and output rates at task j', and $r_{jj'}$ represents the transmission rate from task j to j', respectively. The principle behind this equation is intuitive: an upstream task should first send outputs to a downstream task with the lowest possibility of congestion.

As mentioned in Section 5.2, the above system parameters are maintained by the system state manager and updated based on the reports from downstream. To achieve a high accuracy, the reporting and updating periods should not be set too long. Moreover, to quickly adapt to network disconnections, the $r_{jj'}$ value should be set to 0 by the dispatcher immediately when task j' becomes unreachable.

5.3.2.2 Adaptive Stream Selection (ASS)

As described before, when a task temporarily gets disconnected from some of its downstream tasks, it will send all the output to other remaining counterpart tasks to continue the processing. However, the increased input speed at those tasks may exceed their processing speed, which causes congestion and performance degradation. To deal with this issue, we apply an adaptive stream

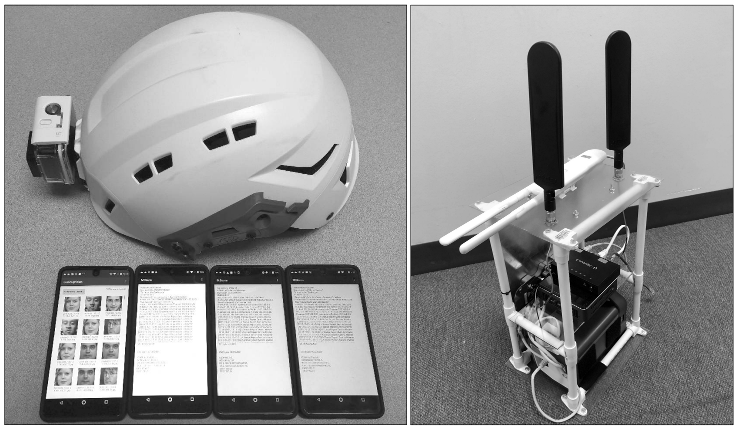


Figure 5.9: A test platform for R-MStorm that consists of a helmet camera, four Android phones and a wireless manpack.

selection mechanism to decide whether to process a tuple in real time or store it to the local file system for later batch processing. The selection mechanism works as follows. First, the selector calculates an IO-based skipping probability for task j as:

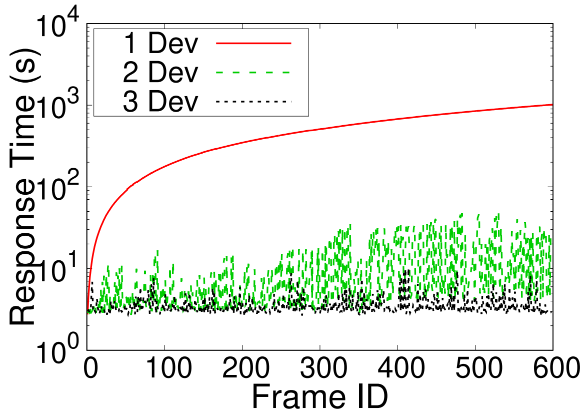
$$prob_j = \max\{\frac{\lambda_j - \mu_j}{\lambda_j}, 0\} \tag{5.12}$$

where λ_j and μ_j represents the input and output rates at task j, respectively. Then, for each input tuple, the selector generates a random number to compare with $prob_j$. If the number is smaller than $prob_j$, the tuple is skipped and stored to the local file system; otherwise, it is processed in real time. The key idea of ASS is to temporarily sacrifice the processing accuracy to avoid congestion and get shorter response time, which is very important for real time stream processing. By processing tuples stored in the file system later, the processing accuracy will be the same in the end.

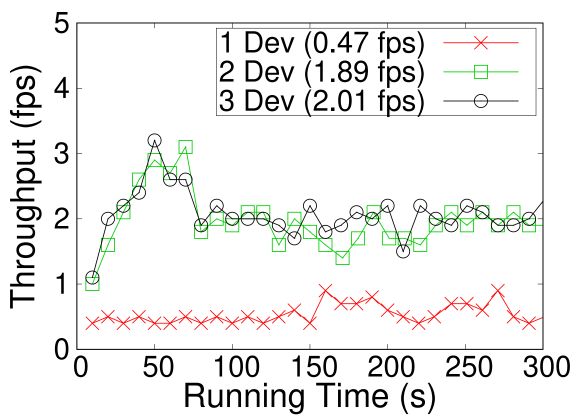
5.4 Evaluation

5.4.1 Experimental setup

The prototype implementation of R-MStorm includes a Java server and an Android client. As shown in Figure 5.9, we run the R-MStorm server on a manpack and four R-MStorm clients on four Essential phones. All these devices are interconnected with wireless networks provided by the



(a) RT with diff. devices



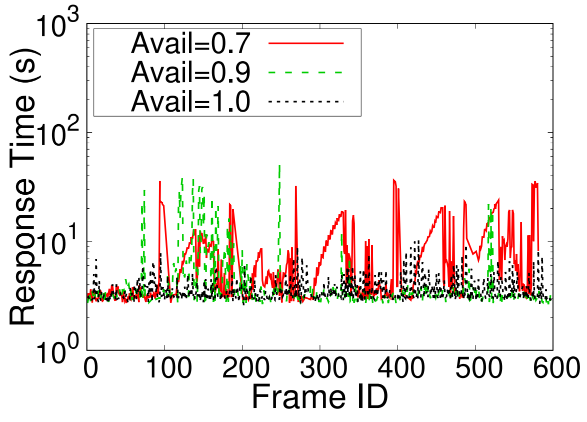
(b) Thr with diff. devices

Figure 5.10: Increasing parallelism improves MSP performance.

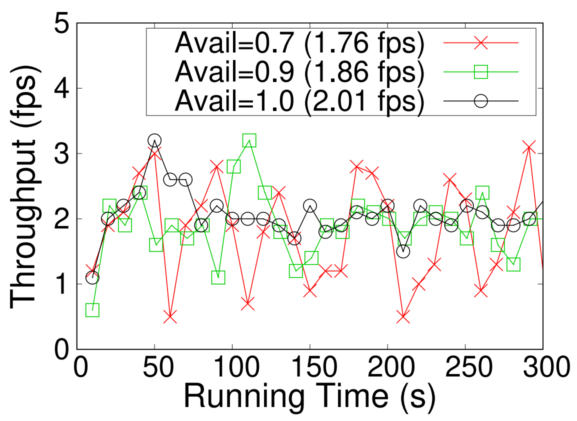
WiFi access point on the manpack. One phone runs a video face recognition App, pulls the video stream from the helmet camera and offloads the stream processing workload to other phones. For all the experiments, we set the video resolution to 1080p and input speed to 2 fps. To mimic the intermittent connectivity of wireless networks during disaster response, we developed an Android application that turns on/off the WiFi of each phone periodically based on a presetting probability. The lower probability means the lower availability and the higher network dynamics. We vary its value from 1 to 0.5 to create different network scenarios. We assume availability and diversity contribute equally to survivability and set w_a and w_d as 0.5 in all the experiments. The metrics we use to evaluate the MSP performance are response time and throughput.

5.4.2 Experimental Results

First, we run experiments with different mobile devices to show how parallelism of tasks influences MSP performance. Starting with one device, we configure the task parallelism of the face recognition App to be 1:1:1:1, which means there is only one task for the picture capture, face detection, face recognition and face sink component, respectively. As shown in Figure 5.10, since the throughput 0.47fps is far below the input rate 2fps, heavy congestion exists in the system, which makes the response time to increase. Then, we increase the number of devices to 2 and reconfigure the task parallelism to 1:2:6:1. This improves the average throughput to 1.89fps and decreases the



(a) RT with diff. networks



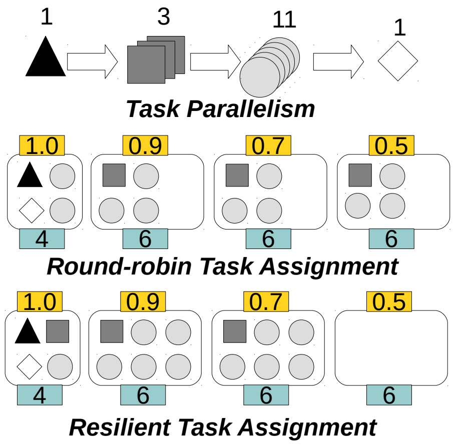
(b) Thr with diff. networks

Figure 5.11: Network condition affects MSP performance.

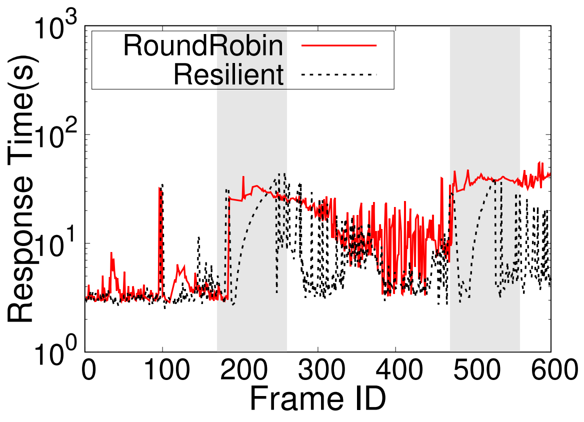
response time to 10.2s. Finally, we increase the number of devices to 3 and configure the task parallelism to 1:3:11:1. Then, the throughput of MSP increases to 2.01fps and the end-to-end response time decreases to 3.6s on average. This experiment shows that, by using nearby mobile devices to increase the task parallelism, the system throughput can be improved and the response time can be reduced significantly.

Then, with 3 devices and task parallelism 1:3:11:1, we run experiments under different network conditions. We change the availability of each phone from 1 to 0.7 to mimic more and more dynamic networks. As we see in Figure 5.11, when the network becomes more dynamic, the response time and throughput fluctuate more as well. Meanwhile, the average response time increases from 3.6s to 7.7s and average throughput decreases from 2.01fps to 1.76fps. This experiment proves that the dynamic edge networks will degrade the MSP performance in terms of both response time and throughput.

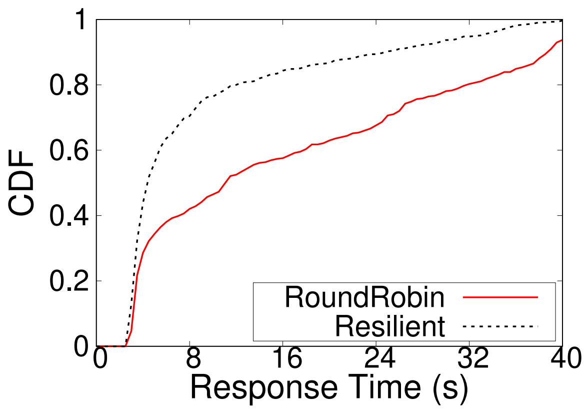
To deal with dynamic edge networks, we use resilient task assignment to assign tasks of an MSP application to mobile devices. In the following, we show how it improves the MSP performance compared with the round-robin method. As shown in Figure 5.12(a), except for the device which submits the topology (with availability 1.0 and executors 4), there are three other devices with availability 0.9, 0.7, 0.5 and executors 6, 6, 6, respectively. Now, we need to assign tasks of the



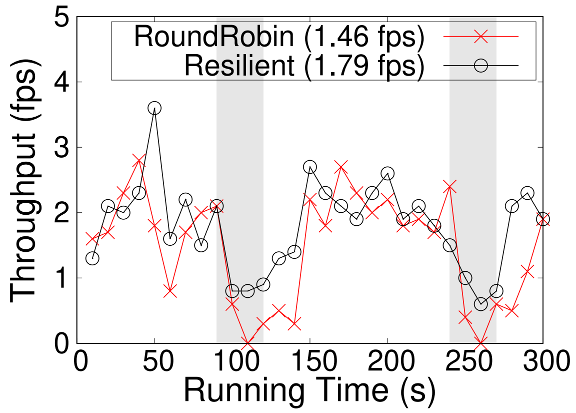
(a) Task assignment methods



(b) Response Time



(c) CDF of Response Time

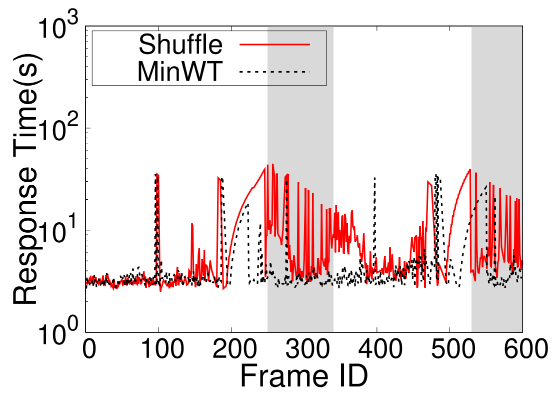


(d) Throughput

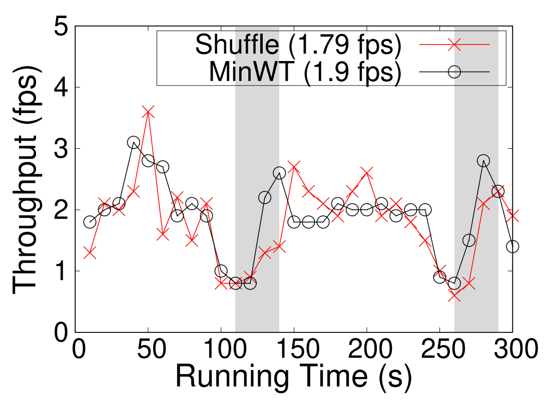
Figure 5.12: Task assignment methods affect MSP performance.

face recognition App (with parallelism 1:3:11:1) to these devices to perform MSP. According to the round-robin method, all devices will be assigned some tasks. However, with RTA, it only chooses three devices with higher availability to run tasks. Besides, RTA ensures that there is at least one complete stream path on the device which submits the topology.

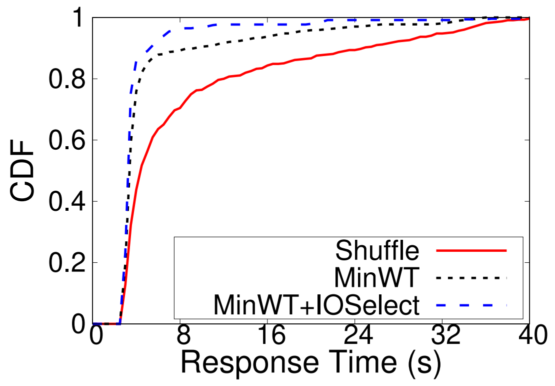
To fairly compare the above two tasks assignment methods, we configure the WiFi on/off time points in two experiments to be the same. As shown in Figure 5.12(b), Figure 5.12(c), and Figure 5.12(d), the response time with RTA is much lower than that with round-robin and the average throughput increases from 1.46fps in round-obin to 1.79fps in RTA. When all other devices become unreachable, with RTA, since there is still a stream path left on the submitter device, the MSP can continue. However, with round-robin, the MSP has to pause until the network recovers. This is



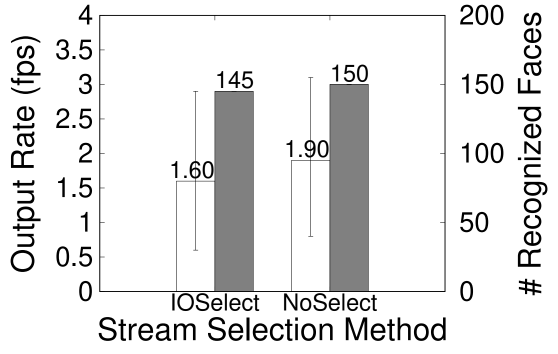
(a) RT of diff. SG methods



(b) Thr of diff. SG methods



(c) CDF of RT for diff. SG & SS



(d) Output & recog. faces with SS

Figure 5.13: Stream grouping and stream selection methods affect MSP performance.

clearly shown in the gray areas of Figure 5.12(b), where the response time gradually increases in RTA (because of the congestion) while the response time in round-robin directly jumps to a high value. The gray areas in Figure 5.12(d) also show this phenomenon, where the throughput in RTA decreases to a low level whereas the throughput in round-robin directly decreases to 0.

Although RTA significantly improves the MSP performance, the response time is still very high in dynamic edge networks. To further improve the performance, we combine RTA with the minimum weight based adaptive stream grouping (MinWT). As shown in Figure 5.13(a) and Figure 5.13(b), with MinWT, the response time decreases from 8.8s to 5.2s and the throughput increases from 1.79fps to 1.9fps on average. The effect of MinWT is particularly obvious when the network recovered from disconnection. As shown in the gray areas of Figure 5.13(a), when the network just recovers from disconnection, the response time in MinWT immediately decreases to the normal level. However, with Shuffle stream grouping, the response time is fluctuating. This is because, in MinWT, since upstream tasks only send the output to the downstream task with the minimum weight, those tasks on the submitter device will not receive any stream for a period time. However, in Shuffle, since upstream tasks send output to the down stream tasks randomly, those congested tasks at the submitter device will continue to get more stream. Tuples being processed by those tasks will have a large response time. The gray areas in Figure 5.13(b) also demonstrate this phenomenon, where the throughput in MinWT recovers much faster than that in Shuffle after the network recovers.

Although RTA with MinWT improves the performance of MSP significantly, the average response time (5.2s) is still far above the average response time (3.6s) in the ideal network. To further reduce the response time in dynamic edge networks, we use the IO-based stream selection mechanism (IOSelect). As shown in Figure 5.13(c), with IOSelect, the average response time is further reduced from 5.2s to 4.0s, which is very close the ideal value. Because of skipping data, the system with IOSelect achieves a lower output rate (1.6fps) than that without selection (1.9fps). However, since a lot of the skipped data is duplicated with the processed ones, the actual accuracy loss is very small: there are only 5 out of 150 (3.3%) faces temporarily missed because of IOSelect. By processing tuples stored in the file system later, the processing accuracy will be the same as without IOSelect in the end.

6. AMVP: ADAPTIVE MULTITASK VIDEO PROCESSING

Nowadays, a core technology to enable complicated vision applications is Convolutional Neural Networks (CNNs), which have replaced traditional methods to become the mainstream technology for vision processing due to their distinguished accuracy and performance [113–116]. In this case, *CNN-based multitask video processing*, which utilizes multiple sophisticated CNNs to run vision analysis tasks on a given raw video stream becomes popular [117–119]. However, running multiple computational intensive CNNs on a resource constrained IoT device (e.g., a surveillance camera) to achieve the user desired application performance (e.g., high accuracy and low latency) is not easy. Additionally, the fact that different users may have different performance preference on different tasks makes this problem more complicated and challenging.

Existing solutions for addressing the above challenge can be roughly divided into three categories: offloading, compression and sharing. First, many offloading strategies [14, 23–26] assume there is a connection between mobile devices and the remote cloud (or nearby edge server) so that parts of CNNs can be offloaded there to achieve the desired performance. However, a stable connection to the cloud is not always available and deploying a powerful edge server near each video camera is costly. Other offloading solutions [27–31] distributes a whole CNN to run on several wireless connected IoT devices. However, they only consider the single CNN case, which is simpler than the multi-CNNs case we consider. Second, the model compression strategies construct efficient CNN models for IoT devices through different compression techniques, such as low-rank expansions [32], parameter quantization [33], pruning and Huffman coding [34], fully factorized convolution [35], depth-wise separable convolution [36], channel-wise sparse connection [37], etc. Unfortunately, these solutions provide a one-for-all model, i.e., a fixed model compression technique is used for different performance goals. A recent work [75] uses on-demand compression, which applies proper compression techniques to different CNN layers to achieve an optimal balance between performance goals and resource constraints. However, it still only considers the single CNN case. Third, the sharing strategies reduce the computation costs [23] and memory footprint [38] by sharing some common layers among multiple CNNs. However, the existing solutions run multiple CNNs on a single device. The performance improvement is restricted by the device resource and its scalability is poor.

Different from existing works which use the above strategies separately, in this chapter, we combine these strategies together as AMVP, an Adaptive execution framework for CNN-based Multitask Video Processing on wireless connected IoT/mobile devices. First, AMVP reduces the computation cost by sharing some common components among different analysis pipelines. For the distinct components which run different vision tasks, AMVP provides multiple CNN implementation candidates. Those candidates are pre-trained from some well-known base CNNs (e.g., MobileNet [120], ResNet [121]) via transfer learning. If two distinct components choose CNNs derived from the same base CNN, they can share some common frozen layers to further reduce the computation cost. There is a trade-off between the number of common frozen layers different CNNs can share and the inference accuracy each CNN can achieve. AMVP aims to keep an optimal balance between them based on the user setting performance goals and available computational resources. Second, AMVP supports distributed execution of CNNs by splitting a big CNN into two smaller components running on different mobile devices. In order to reduce the corresponding communication costs, AMVP leverages an efficient feature compression method based on quantization to compress the features transmitted between separated CNN components. Third, AMVP is developed on top of a mobile stream processing platform called MStorm [8], where it runs an adaptive scheduler to choose appropriate CNN candidates for different pipeline components and assign these components to appropriate devices to achieve the user desired performance goals. We implement a prototype system of AMVP on Android phones and demonstrate its superiority by running a sample CNN-based multitask video analysis application. The experimental results show that, compared to two baseline solutions, AMVP achieves up to 60% lower latency and 10% higher throughput with comparative accuracy.

In summary, this chapter makes the following contributions:

- It shows the possibility of combing three types of orthogonal strategies (i.e. offloading,

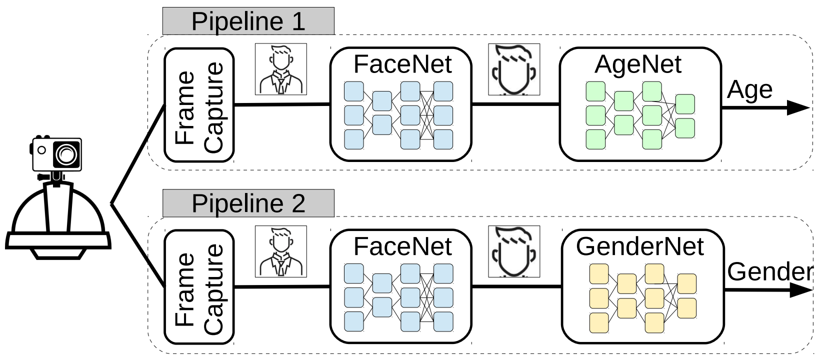


Figure 6.1: An example of CNN-based multitask video processing, which consists of two vision analysis pipelines and four CNNs.

compression and sharing) to support the execution of multiple computational intensive CNNs on resource-constrained mobile devices.

- It designs an adaptive framework which chooses the most appropriate CNNs implementation and running device for each pipeline component based on the user performance goals and actual system resources available.
- It implements a prototype system on Android phones to shown its superiority over other status quo approaches in supporting CNN-based multitask video processing.

6.1 Background and Motivation

6.1.1 CNN-based Multitask Video Processing

Multitask video processing is a new category of applications which applies multiple different analysis pipelines on a given raw video stream to run various vision processing tasks, such as object detection, people re-identification, image classification, activity recognition, etc. CNNs, because of distinguished accuracy and effectiveness, have replaced traditional computer vision methods to be the mainstream technology to implement complex vision tasks. In this case, CNN-based multitask video processing becomes increasingly popular in our lives. Figure 6.1 is an example of CNN-based multitask video processing which consists of two vision analysis pipelines and four CNNs.

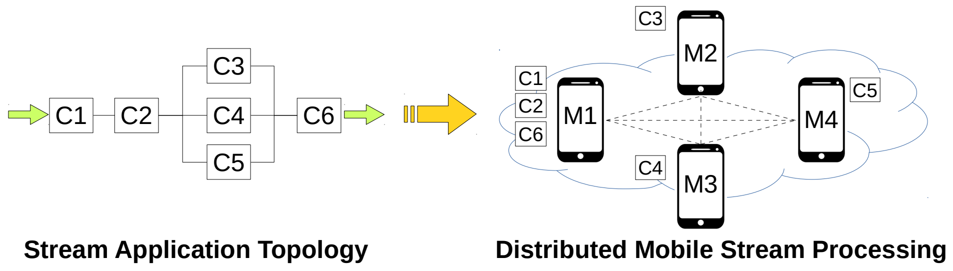

C1
C2
C3
C4
C5
C6
Stream Application Topology
M1
M2
M3
M4
Distributed Mobile Stream Processing

Figure 6.2: An example of Distributed Mobile Stream Processing (DMSP), which assigns six components of a stream application to execute on four wireless-connected mobile devices.

It identifies both age and gender of victims shown up in a video stream from a helmet camera, which helps disaster responders to collect the basic information about victims during recuse.

To deal with issues such as intermittent connectivity, bandwidth limitation, real-time requirements, privacy concern, etc., it is common to run CNN-based multitask video processing at the edge rather than in the cloud. However, executing multiple CNNs concurrently is extremely computational intensive while edge devices usually have very limited computing resources. In addition, these resources are shared with other applications, which makes the actual resources available to run CNNs even fewer. Moreover, the users expect to achieve good application performance and the performance goals may change from time to time. All these factors make running CNN-based multitask video processing at the edge devices challenging.

6.1.2 Distributed Mobile Stream Processing

Distributed Mobile Stream Processing (DMSP) is an emerging computing paradigm [70, 71, 108] that supports online stream processing at the edge. Different from previous systems which offload computation tasks to a nearby edge server, DMSP uses resources of nearby mobile devices to conduct real-time stream processing. In DMSP, a stream application is represented as a graph called topology. Each topology consists of several logical units called component. Each component implements an independent functionality of a whole pipeline. To execute a stream application on a DMSP platform, the user at first needs to submit an application topology. Then, the DMSP platform assigns different application components to proper devices to do distributed

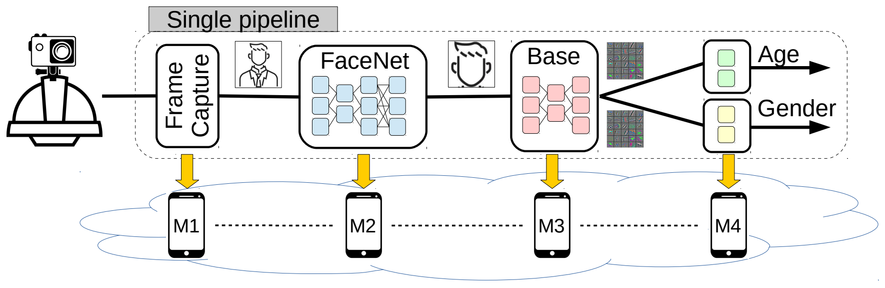


Figure 6.3: Motivation of AMVP: sharing components and layers of multiple CNNs and running them on multiple mobile devices to improve the performance of multitask video processing.

stream processing. Figure 6.2 shows an example of DMSP, where six components C1-C6 of the stream application are assigned to run on four wireless-connected mobile devices M1-M4 to perform distributed stream processing.

Although DMSP makes it easier to deploy and scale stream processing at the edge, there are still some challenges to solve. First, in DMSP, mobile devices are not dedicated to do stream processing, other applications need to execute on them as well. This makes the actual available resources for DMSP uncertain. It would be great if a component of a DMSP application could adjust its workload based on the free resources on the device. Second, mobile devices in DMSP are connected by wireless networks that have dynamic bandwidths. It would be great if the data traffic size between two adjacent components can be adjusted according to the dynamic networks.

6.1.3 Motivation and Challenges

In this chapter, our goal is to develop an adaptive execution framework that enables CNN-based multitask video processing to run on a DMSP platform. On the one hand, we realize that DMSP can provide more computation resources to CNN-based multitask video processing to achieve better performance. On the other hand, we find that CNNs can offer an elastic implementation of application components which perfectly matches the above mentioned adaptive computing and communication requirements of DMSP. Therefore, combing DMSP with CNN-based multitask video processing is a perfect choice.

s

s

s

n

ti

tı

6

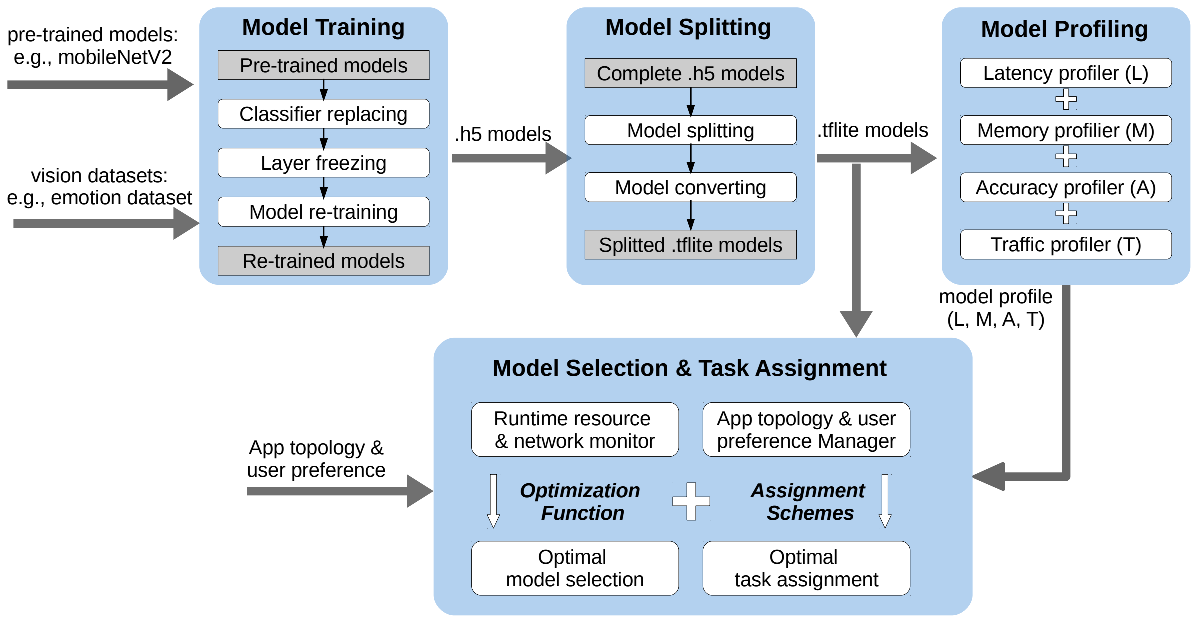


Figure 6.4: AMVP architecture, which includes model training, splitting, profiling, selection and assignment.

Figure 6.4 illustrates the architecture of AMVP, which consists of four stages: model training, model splitting, model profiling and model selection & task assignment. The first three stages are performed offline while the last state is performed online.

Model training. AMVP trains different CNNs for different vision processing tasks through transfer learning. It first takes a well-known pre-trained model, such as mobileNet or ResNet, as vanilla model. Then, it replaces the classifier layers of the vanilla model and freezes some of its base layers. Next, AMVP trains the model with an input dataset (e.g., emotion dataset) and outputs a new CNN (e.g., emotionNet) for a specific vision task (e.g., emotion analysis). By using different vanilla models, replacing with different classifier layers, freezing different base layers and training with different datasets, CNNs with different accuracy and computation workload are obtained for different vision processing tasks. This stage is performed offline, with pre-trained models and datasets as input, .h5 models as output.

Model splitting. Since complete CNNs might be too computational intensive for a resource-constrained mobile device, they need to be split into two parts and run on different mobile devices to achieve the desirable performance. To this end, after getting a group of .h5 CNN models from the model training stage, AMVP splits each .h5 model into two and converts them to .tflite format for mobile devices. To adapt to the uncertain available resources at mobile devices, AMVP splits each .h5 model at different splitting points, resulting in different .tflite model pairs. For each pair, the first part extracts intermediate features from the input and the second part makes inference on the features output by the first part. For different pairs, the computation costs of the first and second parts, as well as the features between them, are different. This stage is performed offline, with .h5 models as input and .tflite models as output.

Model profiling. After getting a set of .tflite model pairs for each vision task, AMVP profiles each pair of .tflite models on mobile devices to get the execution latency, memory footprint, inference accuracy and traffic size of feature transmission. To set a benchmark, when profiling these parameters, the mobile devices are assumed to be in an idle state, i.e., except for the necessary system services, no other applications are running. This stage is also performed offline, with .tflite models as input and diverse profile information as output.

Model selection & task assignment. Finally, after getting a group of .tflite models and corresponding profiles for each vision task, AMVP needs to assign a CNN-based multitask video processing pipeline to execute on a DMSP platform to achieve the desirable performance. To this

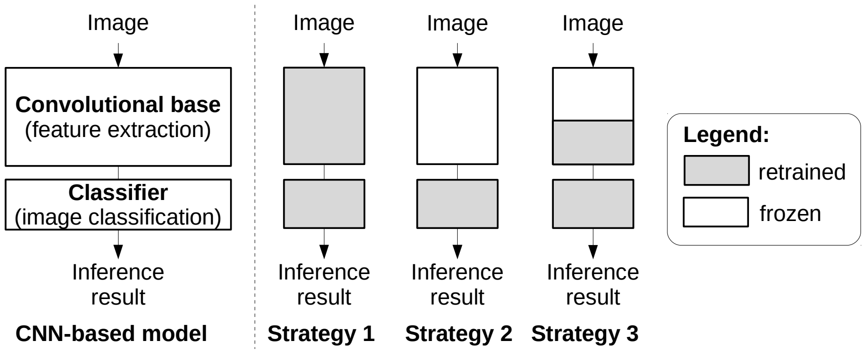


Figure 6.5: Transfer learning strategies for CNN models.

end, AMVP uses a resource and network monitor to acquire the actual available resources and network speed at each mobile device. AMVP also uses a topology and user preference manager to obtain the application topology and performance goals from the user. Then, based on these information, AMVP chooses the optimal CNN candidate for each application component and assgins all the components of application to the optimal devices based on an optimization function and a task assignment scheme. A detailed description about optimization function and scheme are given in the next section. This stage is performed online, with .tflite models and corresponding profiles, actual resource and network condition, applicaiton topology and performance goals as input, with an optimal model selection and task assignment as output.

6.3 Design of AMVP

6.3.1 Transfer Learning and Layer Sharing

In AMVP, CNNs for different vision processing tasks are obtained through transfer learning – a popular approach which builds accurate CNNs for new tasks in a very efficient way. Instead of learning from scratch, transfer learning starts with patterns learned from solving similar problems. In computer vision, these patterns are pre-trained deep CNN models (e.g., VGGNet, MobileNet, ResNet, InceptionNet, etc.) trained on a large benchmark dataset (e.g., ImageNet).

As shown in Figure 6.5, a pre-trained deep CNN model can be divided into two parts: the convolutional base that consists of stacked convolutional and pooling layers and the classifier that consists of fully connected layers. The goal of a convolutional base is to extract features from an input image and the goal of a classifier is to classify the image based on extracted features. An interesting aspect of the convolution base is that it learns hierarchical features. The lower-layer features are general and the higher-layer features are specialised, i.e., there is transition from general to specific in the network [122]. General features can be reused in different tasks while specific features strongly depend on the specific task and dataset.

When performing transfer learning, AMVP first replaces the original classifier in a pre-trained model with a new classifier. Then, it fine tunes the parameters of the new model based on one of three strategies shown in Figure 6.5. Strategy 1 re-trains all the weights of the model from scratch. It requires a large dataset and a lot of computation. Strategy 2 freezes the whole convolutional base and only trains the classifier. It uses the pre-trained model as a fixed feature extractor, which be suitable for training tasks similar to the original one with small datasets. Strategy 3 is a compromise between Strategy 1 and Strategy 2, which chooses how many layers in the convolutional base to freeze. As a rule of thumb, if a new task has a small dataset, freezing more layers avoids overfitting. But if a new task has a large dataset, retraining more layers improves the accuracy.

If CNNs for different tasks happen to be generated from the same pre-trained model, they will have some common frozen layers in the convolutional base. To reduce the computation costs and memory footprint, AMVP can share common frozen layers among multiple CNNs. An interesting fact is that there is a trade-off between the number of frozen layers different CNNs share and the inference accuracy each CNN achieves. In general, the more frozen layers different CNNs share, the more computation costs can be saved, but the less specific each CNN will be and the lower inference accuracy each CNN will achieve. AMVP needs to choose an appropriate frozen point for each CNN model to keep a balance between the computation cost and inference accuracy.

6.3.2 Model Splitting and Feature Compression

In AMVP, to enable distributed CNN execution, a complete CNN model is split into two parts: Feature Extraction (FE) and Feature Inference (FI). FE takes a raw image as input and outputs features extracted from that image; FI takes features extracted from an image as input and outputs

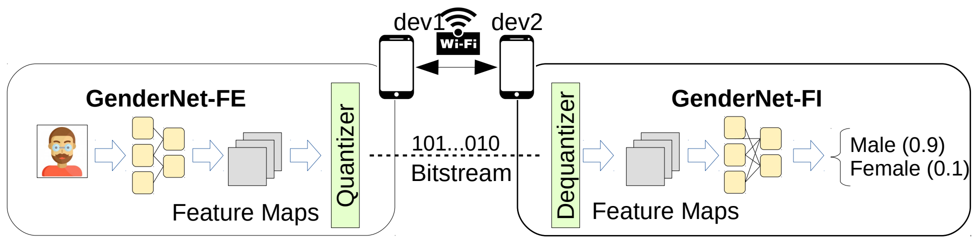


Figure 6.6: Compression and decompression of feature maps.

the inference results. There are multiple splitting points where a CNN model can be split, resulting in a group of (FE, FI) pairs. To improve the processing throughput, FE and FI of a pair are scheduled to run on different mobile devices. However, due to the large data size of intermediate features and dynamic bandwidth of wireless networks, how to efficiently transfer features from FE to FI becomes a key challenge. In AMVP, we use quantization-based compression method to reduce the feature data size. The whole processes is shown in Figure 6.6, where the features output by a FE is quantized by a quantizer before they are sent to a FI. Then FI receives the quantized features, it uses a corresponding dequantizer to recover the original feature.

Splitting: There are multiple splitting points to divide a CNN into two separate parts. In different splitting cases, the traffic sizes for feature transmission between two separate parts are different. Usually, the size of features gradually gets reduced along with the inference process [79]. The reason behind is: features at the lower layers represent lower level information which are more detailed and fragmented; however, features at the higher layers represent higher level information converged from the lower level, which are more abstract and advanced. Usually, the lower level information is more than the higher level. However, this characteristic of CNNs does not mean that AMVP should always split a CNN at a higher layer. Actually, except for the communication costs, AMVP also considers the computation cost of two separated parts: although splitting at a higher layer saves communication costs, computation costs of two parts become less balanced, which is not beneficial for achieving high processing throughput.

Quantization: The features output by FE is quantized to n-bit precision by a uniform quantizer defined as follows:

$$\overline{\mathbf{F}} = \lfloor \frac{\mathbf{F} - \min(\mathbf{F})}{\max(\mathbf{F}) - \min(\mathbf{F})} \cdot (2^n - 1) \rceil \tag{6.1}$$

where $\mathbf{F} \in \mathbb{R}^{H \times W \times C}$ is the tensor containing the feature map data with H as height, W as width, and C as channels. $\min(\mathbf{F})$ and $\max(\mathbf{F})$ denote the minimum and the maximum values in $\mathbf{F}$, respectively. $\overline{\mathbf{F}}$ denotes the quantized feature tensor and $\lfloor \cdot \rceil$ denotes a function rounding to the nearest integer. When the quantized feature tensor $\overline{\mathbf{F}}$, $\min(\mathbf{F})$ and $\max(\mathbf{F})$ are obtained at the FI of a CNN, $\overline{\mathbf{F}}$ is dequantized by a uniform dequantizer:

$$\widehat{\mathbf{F}} = \frac{\max(\mathbf{F}) - \min(\mathbf{F})}{2^n - 1} \cdot \overline{\mathbf{F}} + \min(\mathbf{F}) \tag{6.2}$$

where $\widehat{\mathbf{F}}$ denotes the dequantized feature tensor. It deserves to be mentioned that, although $\widehat{\mathbf{F}}$ is not exactly equal to $\mathbf{F}$, some research [77, 123, 124] shows that, when the n value is above a threshold, the quantization process has a negligible effect on the accuracy of image classification and object detection. As described later in Section 6.4.2, AMVP uses $n = 8$ to compress the feature most while keep the original accuracy.

Evaluation metric: We leverage three metrics to evaluate the quantization-based compression performance. The first metric is compression rate (CR), which is defined as:

$$CR = \frac{Feature\ size\ after\ compression}{Feature\ size\ before\ compression}. \tag{6.3}$$

The second metric is fidelity, which evaluates the information loss of dequantized features for image classification. It is calculated by comparing the original onehot classification results with the outputs inferred from the dequantized features [79]:

$$Fidelity = 1 - \frac{1}{2N} \sum_{i=1}^{N} Hamming(O_i^{og}, O_i^{cp}) \tag{6.4}$$

where O_i^{og} denotes the original onehot classification result of image sample i and O_i^{cp} denotes the onehot output inferred from the dequantized features. $Hamming(\cdot)$ is the hamming distance function and N denotes the total number of samples.

The third metric is compression benefits (CPB), which compares the total latency for transmitting features in compressed format and latency for transmitting features in original format. The concrete definition is as follows:

$$CPB = OTR - (QT + TR + DQ) \tag{6.5}$$

where OTR is the latency for transmitting features in original format, QT is the quantization time for features, TR is the latency for transmitting features in compressed format and DQ is the dequantization time for features.

6.3.3 Model Selection and Task Assignment

In AMVP, a set of .tflite models and profiles are deployed on mobile devices in advance, AMVP needs to choose an appropriate model for each vision task and assign the whole CNN-based multitask video processing pipeline to proper devices of a DMSP platform to achieve the desirable performance. We name this procedure as Model Selection and Task Assignment (MSTA) and formulate it mathematically as the following.

Let S denote the set of vision analytic tasks in a multitask video processing application and let A_s, L_s and T_s denote the user setting goals for inference accuracy, processing latency and processing throughput respectively for task $s \in S$. Let M_s represent the set of all available CNN candidates for task s and let $m_s \in M_s$ denote a specific candidate with specific pre-trained CNN type and frozen layers. In addition, we use $\boldsymbol{m_s} = (m_s^1, m_s^2)$ to denote a specific separation of CNN model m_s, which consists of two separate parts m_s^1 and m_s^2. The size of features transmitted from m_s^1 to m_s^2 is denoted as $f_{m_s^1 m_s^2}$ and the communication bandwidth between devices k_1, k_2 that execute m_s^1, m_s^2 is denoted as $b_{k_1 k_2}$. $D(f_{m_s^1 m_s^2}, b_{k_1 k_2})$ denotes the latency for delivering features from k_1 to k_2. Then, the cost function for task s is defined as follows:

$$\begin{aligned} C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s) = \alpha_s \cdot & \frac{A_s - A(\boldsymbol{m_s})}{A_s} \\ + \beta_s \cdot & \frac{\max(0, L(\boldsymbol{m_s}, \boldsymbol{u_s^k}) - L_s)}{L(\boldsymbol{m_s}, \boldsymbol{u_s^k})} \\ + \gamma_s \cdot & \frac{\max(0, T_s - T(\boldsymbol{m_s}, \boldsymbol{u_s^k}))}{T_s} \end{aligned} \tag{6.6}$$

where $A(\boldsymbol{m_s})$ is the inference accuracy of $\boldsymbol{m_s}$, $L(\boldsymbol{m_s}, \boldsymbol{u_s^k})$ is the processing latency defined as

$$L(\boldsymbol{m_s}, \boldsymbol{u_s^k}) = \frac{l_{m_s^1}^{k_1}}{u_{m_s^1}^{k_1}} + \frac{l_{m_s^2}^{k_2}}{u_{m_s^2}^{k_2}} + D(f_{m_s^1 m_s^2}, b_{k_1 k_2}) \tag{6.7}$$

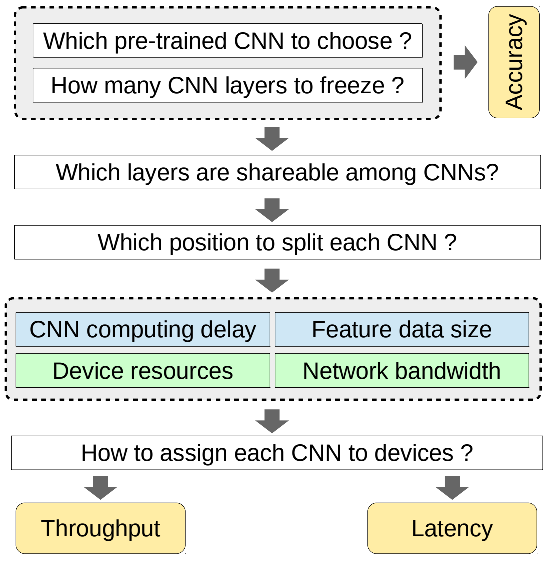
which includes both computation and communication latency and $T(\boldsymbol{m_s}, \boldsymbol{u_s^k})$ is the processing throughput defined as

$$T(\boldsymbol{m_s}, \boldsymbol{u_s^k}) = \min\{u_{m_s^1}^{k_1} t_{m_s^1}^{k_1}, u_{m_s^2}^{k_2} t_{m_s^2}^{k_2}, \frac{1}{D(f_{m_s^1 m_s^2}, b_{k_1 k_2})}\} \tag{6.8}$$

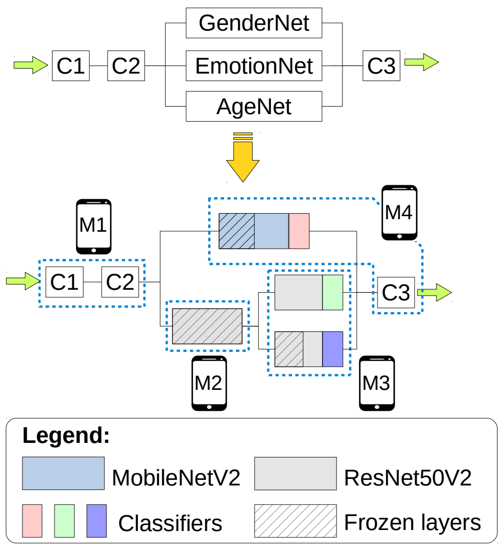
where $\boldsymbol{u_s^k} = (u_{m_s^1}^{k_1}, u_{m_s^2}^{k_2})$, $u_{m_s^i}^{k_j} \in (0, 1]$ denotes the percentage of computing resources allocated to m_s^i at device k_j, $l_{m_s^i}^{k_j}$ and $t_{m_s^i}^{k_j}$ denote the processing latency and throughput of m_s^i when 100% computing resources are allocated to m_s^i at device k_j.

In the cost function, the first term promotes to select a CNN candidate with the highest inference accuracy, the second term penalizes choosing a CNN candidate that achieves a processing latency higher than the latency goal L_s and the third term penalizes choosing a CNN candidate that achieves a processing throughput lower than the throughput goal T_s. However, since the video stream is streamed at a fixed frame rate, there is no reward to achieve a latency lower than L_s and a throughput higher than T_s. To allow trade-off among accuracy, latency and throughput, parameters $\alpha_s, \beta_s, \gamma_s \in [0, 1]$, $\alpha_s + \beta_s + \gamma_s = 1$, can be set by the user to indicate the importance of accuracy, latency and throughput, respectively. The larger a parameter is, the more important the corresponding term is.

Given the cost function of each task, AMVP applies a widely used MinMaxCost scheme [38]


Which pre-trained CNN to choose ?
How many CNN layers to freeze ?
Accuracy
Which layers are shareable among CNNs?
Which position to split each CNN ?
CNN computing delay
Feature data size
Device resources
Network bandwidth
How to assign each CNN to devices ?
Throughput
Latency

(a) Basic workflow of MSTA



(b) An example of MSTA

Figure 6.7: Model Selection and Task Assignment (MSTA) for AMVP.

to perform the model selection and task assignment, which minimizes the cost of the task that has the largest cost. The optimization problem of this scheme is formulated as follows:

$$\underset{\boldsymbol{m_s}, \boldsymbol{u_s^k}}{\text{minimize}} \quad C \tag{6.9}$$

$$\text{subject to:} \quad \forall s : C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s) \leq C, \tag{6.10}$$

$$\forall k : \sum_{\{m_s^i\}} u_{m_s^i}^k \leq U_k, \tag{6.11}$$

$$\forall k : \sum_{\{m_s^i\}} r_{m_s^i}^k \leq R_k \tag{6.12}$$

where the cost of any task k must be smaller than C where C is minimized. $\{m_s^i\}$ is a CNN model set where all the models inside are different. At device k, $u_{m_s^i}^k$ denotes the percentage of computing resources allocated to model m_s^i, U_k denotes the percentage of total available computing resources, $r_{m_s^i}^k$ denotes the runtime memory usage of model m_s^i and R_k denotes the total available memory. With the MinMaxCost scheme, AMVP assigns resources to all the tasks of video processing in a fair way so that there is no obvious bottleneck.

Algorithm 4: MSTA(), a greedy algorithm

```
Input  : $\forall s$: $M_s$, $A_s$, $L_s$, $T_s$, $\alpha_s$, $\beta_s$, $\gamma_s$;
         $\forall k, k'$: $U_k$, $R_k$, $b_{kk'}$; $\forall m_s^i, k$: $l^k_{m_s^i}, t^k_{m_s^i}$
Output: Model selection $\overline{\boldsymbol{Q}}$ and task assignment $\overline{\boldsymbol{X}}$
1  $\overline{\boldsymbol{Q}} \leftarrow \varnothing$ , $\overline{\boldsymbol{X}} \leftarrow \varnothing$
2  $minMaxCost \leftarrow 1$ // based on Eq. 6.6, the maximum cost is 1
   // Initialize with CNNs that have the highest accuracy
3  $P \leftarrow \{\forall s$: $m_s | m_s$ is CNN in $M_s$ with highest accuracy$\}$
4  while $\exists m_s \in P$ *is NOT with the lowest accuracy* do
5      CFL $\leftarrow$ common frozen layers of CNNs in $P$
6      $\boldsymbol{Q} \leftarrow \varnothing$
7      for $m_s \in P$ do
           // split each model based on common frozen layers
8          $\boldsymbol{m_s} \leftarrow (m_s^1, m_s^2)$, separation of $m_s$ based on CFL
           // record separated models as model selection
9          $\boldsymbol{Q}.add(\boldsymbol{m_s})$
10     $\boldsymbol{X} \leftarrow$ TaskAssign($\boldsymbol{Q}$, **Input**) // Algorithm 2
11     $cost \leftarrow 0$, $exit \leftarrow ture$, $accCost \leftarrow 1$, $\overline{s} \leftarrow NULL$
12     for $s \in S$ do
           // calculate cost of each task and find the maximum
13         $cost \leftarrow \max(cost, C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s))$
           // update flag for early exit
14         if $L(\boldsymbol{m_s}, \boldsymbol{u_s^k}) > L_s$ *or* $T(\boldsymbol{m_s}, \boldsymbol{u_s^k}) < T_s$ then
15             $exit \leftarrow false$
           // find task with the minimum accuracy cost
16         if $accCost > \alpha_s \cdot \frac{A_s - A(\boldsymbol{m_s})}{A_s}$ then
17             $\overline{s} \leftarrow s$
18             $accCost \leftarrow \alpha_s \cdot \frac{A_s - A(\boldsymbol{m_s})}{A_s}$
       // update minMaxCost, model selection and task assignment
19     if $cost < minMaxCost$ then
20         $minMaxCost \leftarrow cost$
21         $\overline{\boldsymbol{Q}} \leftarrow \boldsymbol{Q}$, $\overline{\boldsymbol{X}} \leftarrow \boldsymbol{X}$
22     if $exit = ture$ then
23         return $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$
       // replace model for task with the minimum accuracy cost
24     $m'_{\overline{s}} \leftarrow$ CNN in $M_{\overline{s}}$ with accuracy second to $m_{\overline{s}} \in P$
25     $P \leftarrow P \setminus \{m_{\overline{s}}\} \cup \{m'_{\overline{s}}\}$ // update model in $P$ for task $\overline{s}$
26 return $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$
```

In order to solve the above computationally hard nonlinear optimization problem, AMVP uses a greedy heuristic algorithm to get an approximate solution. The key idea of this algorithm is built on a basic workflow of MSTA shown in Figure 6.7(a). First, each task selects a specific CNN implementation from all the available candidates, which implies the following information: 1) the used pre-trained CNN model; 2) the frozen CNN layers. This selection directly determines the inference accuracy that each task can achieve. Then, after each task chooses its own CNN implementation, information about how many common frozen layers are shareable among different CNNs are easy to obtain. To minimize the total computation workload, different CNNs try to share as many common frozen layers as possible. With this shared layer information, each related CNN can be split into two sub-parts. Since the profile of each CNN sub-part is acquired in advance, AMVP finally assigns all the sub-parts to proper devices based on the monitored system status. Based on the assignment, throughput and latency of each task is determined. A detailed description of this greedy algorithm is in Algorithm 4. First, we use $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ to represent the final model selection and task assignment and we set the minimum maximum cost value $minMaxCost$ as 1 (line 1-2). Then, we create a set P by choosing the model with the highest inference accuracy for each task (line 3). Starting with this initial set, a procedure is performed repeatedly until all the models in the set have the lowest accuracy (line 4). In the procedure, we first figure out the common frozen layers among all the models (line 5). Then, base on the common frozen layer, each model is split into two and stored in $\boldsymbol{Q}$ (line 6-9). Next, based on the separated models and the input information, we call a task assignment procedure (Algorithm 5) to get a task assignment $\boldsymbol{X}$ (line 10). Based on the assignment $\boldsymbol{X}$, we calculate the cost function of each task and record the maximum one (line 13). We also calculate the latency and throughput of each task and check if they meet the goals (line 14). If either of them does not meet the goal, the whole procedure cannot exit in advance (line 15). Besides, we also check which task has the minimum accuracy cost and record it for the later update (line 16-18). After we find the maximum cost of all tasks, we compare it with $minMaxCost$ and update $minMaxCost$ with it if it is smaller (line 19-20). Correspondingly, we also update $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ with a better solution (line 21). Then, we check if the procedure can exit early (line 22). If so, we get current $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ as the optimal solution (line 23). Otherwise, we update model set P by replacing the model of task with the minimum accuracy cost and continue the loop procedure (line 24-25). In the end, when all the models in P have the lowest accuracy, the loop procedure stops and the whole algorithm returns $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ as the optimal solution (line 26).

Algorithm 5: TaskAssign(), a genetic algorithm

```
   Input  : $\boldsymbol{Q} = \{\boldsymbol{m_s}\}; \forall s$: $A_s$, $L_s$, $T_s$, $\alpha_s$, $\beta_s$, $\gamma_s$;
            $\forall k, k'$: $U_k$, $R_k$, $b_{kk'}$; $\forall m_s^i, k$: $l_{m_s^i}^k, t_{m_s^i}^k$
   Output: Task assignment $\boldsymbol{X}$
1  $P \leftarrow$ RandomlyInitPopulations$()$
2  $\boldsymbol{X} \leftarrow$ SelectBest$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), P)$
3  $gen \leftarrow 0$;
4  while $gen < MAXGEN$ do
5      $P_{pa} \leftarrow$ SelectParents$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), P)$
6      $P_{ch} \leftarrow$ GenerateChildren$(P_{pa})$
7      $P_{mu} \leftarrow P_{ch}$
8      for $p \in P_{mu}$ do
9          $p \leftarrow$ Mutate$(p, MR)$ // mutation rate: $MR$
           // recombination period: $RP$
10         if $gen\%RP = 0$ then
11             $p \leftarrow$ Recombination$(p)$
       // Filter constraints: $CSTR$
12     $P_{off} \leftarrow$ FilterOffspring$(CSTR, P_{ch} \cup P_{mu})$
13     $P \leftarrow$ SelectPopulations$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), P_{pa} \cup P_{off})$
14     $\boldsymbol{X} \leftarrow$ SelectBest$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), \{\boldsymbol{X}\} \cup P)$
15 return $\boldsymbol{X}$
```

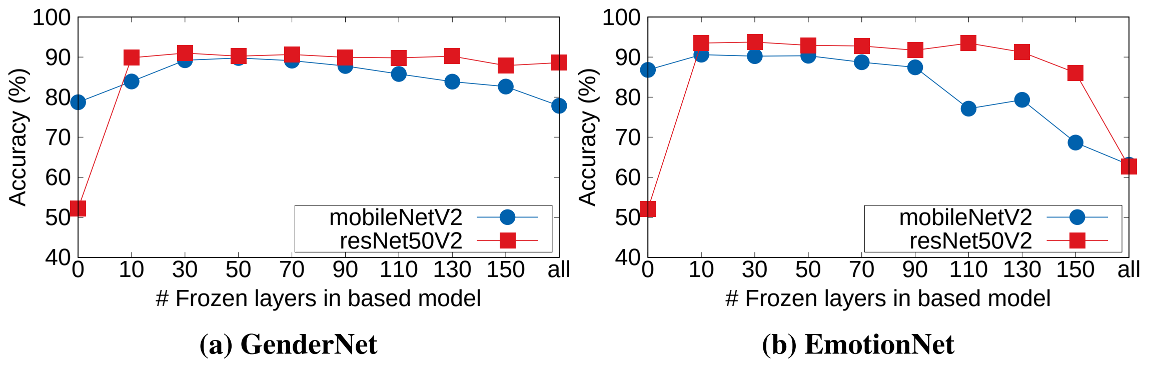
Algorithm 5 describes the task assignment procedure used in Algorithm 4. It is a genetic algorithm which starts with a certain amount of random assignments (line 1) and runs an iterative process (line 4-14) containing the following operations: a) SelectParents, which selects a certain number of candidate assignments as parents based on the cost function (line 5); b) GenerateChildren, which generates children assignments by running uniform crossover (line 6); c) Mutate, which chooses some rows of an assignment and changes the values at some random positions (line 9); d) Recombination, which exchanges two rows of a task assignment to achieve a lower cost (line 11); e) FilterOffSpring, which deletes task assignments that violate constraints (line 12); f) Select-Populations, which selects new populations from all the candidates based on the cost function; g) SelectBest, which selects an assignment with the minimum cost (line 13). After iterating the above operations for some generations, it finally returns an assignment $\boldsymbol{X}$ which achieves a low cost with best efforts (line 15).

A simple example of MSTA is in Figure 6.7(b), where three components of a video processing application depends on CNNs. For each component, there are a group of CNN candidates with two pre-trained CNN types (MobileNetV2, ResNet50V2) and different frozen layers. Based on MSTA, GenderNet chooses a MobileNetV2-based CNN while both EmotionNet and AgeNet choose the ResNet50V2-based CNNs. GenderNet executes the selected CNN at device M4. EmotionNet and AgeNet execute the common frozen layers of the selected CNNs at device M2 and execute the distinct layers of the CNNs at device M3.

6.4 Evaluation

6.4.1 Experimental setup

The implementation of AMVP are divided into two stages: offline stage and online stage. For the offline stage, we first train a group of CNNs for different vision analytic tasks (e.g., gender recognition, emotion recognition, age recognition, etc.) via transfer learning with Keras [125] API of TensorFlow [126]. These CNNs are trained from different pre-trained CNNs with different frozen layers, which result in a group of different .h5 models. The pre-trained CNNs we use are MobileNetV2 and ResNet50V2 trained on ImageNet [127]. The dataset we use to train each vision task contains 750 images for each category: 500 images for training and 250 images for validation. After getting the .h5 models, we split each model at different layers and convert the separate models into .tflite format. This results in a group of .tflite model pairs. Next, we run each .tflite model pair on Android phones to profile critical information such as inference accuracy, processing latency,



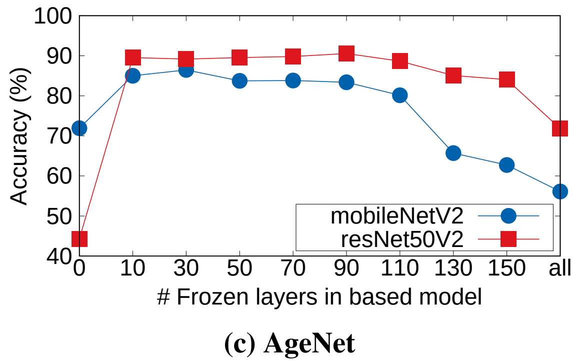
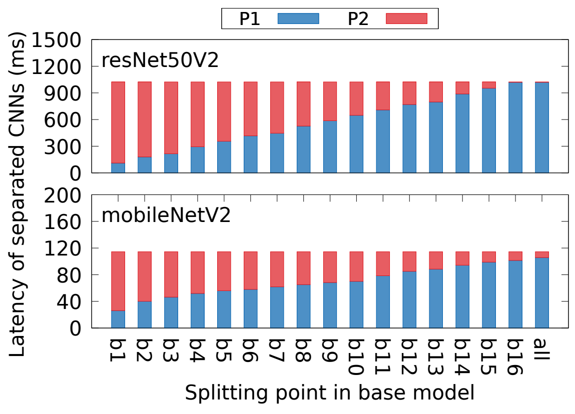


Figure 6.8: The number of frozen layers in the pre-trained CNNs affects the inference accuracy of vision analysis tasks.

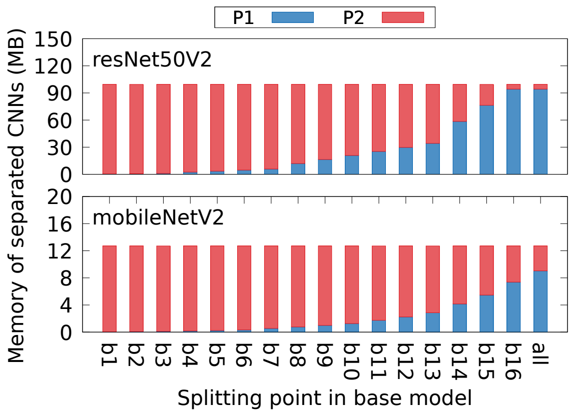
memory footprint, feature traffic size, etc. We gather these information together as a database for the later model selection and assignment.

For the online stage, we implement AMVP on top of a mobile distributed stream processing system [108] running on a test platform shown in Figure 5.9, which consists of a helmet Yi® camera, four Essentail® phones and a wireless manpack. An AMVP client application with a group of .tflite model pairs is installed on each phone in advance. An AMVP server with the profile database is running on the manpack for model selection and task assignment. Before a user executes a multitask video processing application, he/she first sends a request containing the application information and the performance goals to the AMVP server. When the server receives the request, it calls the MSTA algorithm to select an appropriate CNN model for each vision task and assigns them to run on proper devices. When an AMVP client receives the assignment from the server, it loads the selected CNNs to memory to perform video analysis.

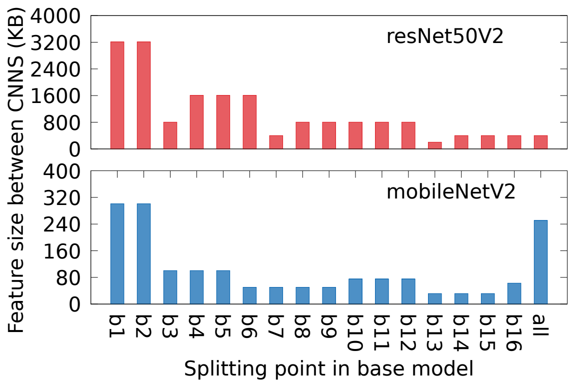
In our experiments, we utilizes a multitask video processing application in Figure 6.7(b) to eval-



(a) Latency



(b) Memory size



(c) Feature size

Figure 6.9: Latency, memory size of and feature size between separated CNNs at different splitting points.

uate AMVP. We compare AMVP with two baseline strategies, i.e., Pure Sharing Strategy (PSS), which shares common frozen layers among multiple CNNs on a single mobile device and Pure Offloading Strategy (POS), which offloads CNNs layers to other devices without sharing. We run extensive experiments with various performance goals under different computing and networking condition to show how AMVP adapts to the dynamic edge environment.

6.4.2 Experimental results

Transfer learning of CNNs: We train CNNs for different vision analysis tasks through transfer learning with different frozen layers in the pre-trained base models. As shown in Figure 6.8, in general, ResNet50V2-based CNNs achieves higher inference accuracy than MobileNetV2-based CNNs because the former models have deeper layers (190 vs. 155) and more parameters (23.56M vs. 2.25M). For each specific vision task, we observe that, as the number of frozen layers increases, the inference accuracy first increases and then decreases. This is because the datasets we use for training different vision analysis tasks are much smaller than the ImageNet [127] dataset used for training ResNet50V2 and MobileNetV2. If we retrain all the layers in the base model (Strategy 1 in Figure 6.5), it is easy to encounter the overfitting problem. Specifically, we find that, the overfitting problem for ResNet50V2-based CNNs is more serious than MobileNetV2-based CNNs because it has more parameters. By contrast, if we freeze all the layers in the based model (Strategy 2 in Figure 6.5), it is difficult to extract specific features for each vision analysis task, which also leads to low accuracy. Therefore, freezing some layers in the base model (Strategy 3 in Figure 6.5) is a wise choice when performing transfer learning on large CNNs with small dataset. Besides, we also observe that, the effect of freezing proper layers is more obvious for complicated applications. For example, as shown in Figure 6.8(a), for gender recognition which is simpler than emotion or age recognition, the accuracy difference of models with different frozen layers is smaller.

CNN model profiling: In Figure 6.9, we choose 17 points as candidate splitting points for MobileNetV2 and ResNet50V2. In fact, these points are the last layer of each CNN block. For each splitting point, we profile the latency and memory size of two separated CNN parts P1 and P2, as well as the feature size between them. As we observe in Figure 6.9(a), the latency of P1 almost increases linearly with the increase of block number. Therefore, to balance the computation workload between P1 and P2, the optimal splitting point should be in the middle. However, the memory size of a CNN is mainly concentrated in the back of the model (Figure 6.9(b)). Meanwhile, the size of the feature between P1 and P2 is decreasing as the block number increases. Therefore, from the perspective of memory balance and communication cost, splitting at a latter layer is better. Moreover, from the layer sharing perspective, if we split different CNNs at a latter layer, these CNNs will have a larger opportunity to share more layers by freezing the layers of their first parts. However, as we observe from Figure 6.8, freezing too many CNN layers decreases the inference accuracy of vision tasks. Overall, from the above profile data, we conclude that, when we split a CNN model into two separated parts, there is a very complicated trade-off among different metrics

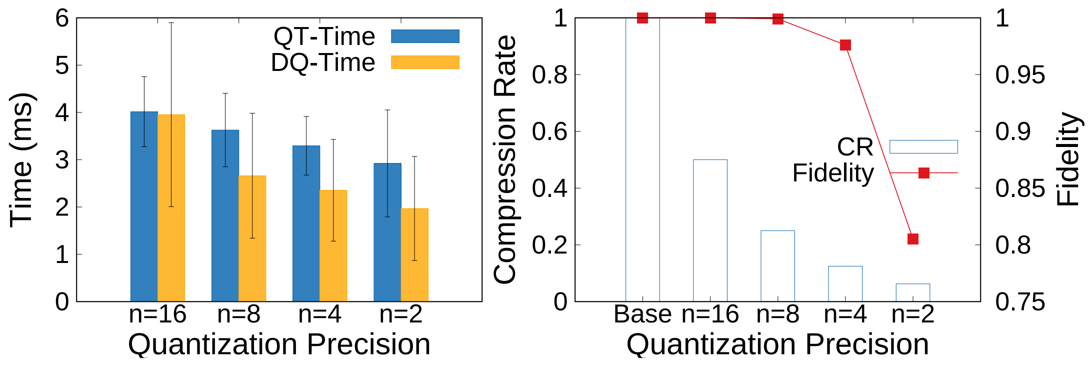


Figure 6.10: An example (MobileNetV2-based AgeNet, splitting at layer 90, feature size $\approx$ 50 KB) that shows how quantization precision affects quantization/dequantization time, compression rate and fidelity.

including computation workload and memory footprint balance, communication costs, shareable common layers, and inference accuracy. This motivates us to design a framework that is able to adaptively choose the most appropriate splitting points for different CNNs based on the user requirements and environment condition.

Feature compression and transmission: After we split a CNN into two separated parts P1 and P2 and deploy them on two devices, P1 needs to transmit its extracted feature to P2 to finish the inference. As mentioned in 6.3.2, in AMVP, we use a quantization-based method to compress the feature before transmitting it. In Figure 6.10, we use a simple example to show how the quantization precision affects the total quantization and dequantization time, compression rate and fidelity. We use a MobileNetV2-based AgeNet, splitting at layer 90, with inter-part feature size around 50KB. As we see from the left side figure, when the quantization precision n decreases, both the quantization and dequantization time also decreases. From the right side figure, we find that, although the compression rate decreases as the quantization precision decreases, the fidelity metric starts to decrease at some point (n=8, fidelity=0.999) as well. To keep the inference accuracy, we adopt n=8 in AMVP.

With a fixed quantization precision n=8, we show in Figure 6.11 that how the network bandwidth and feature data size affect the compression benefits. In the left side figure, we transmit a fixed size feature (50KB) at different network bandwidths with both compressed size (CP) and

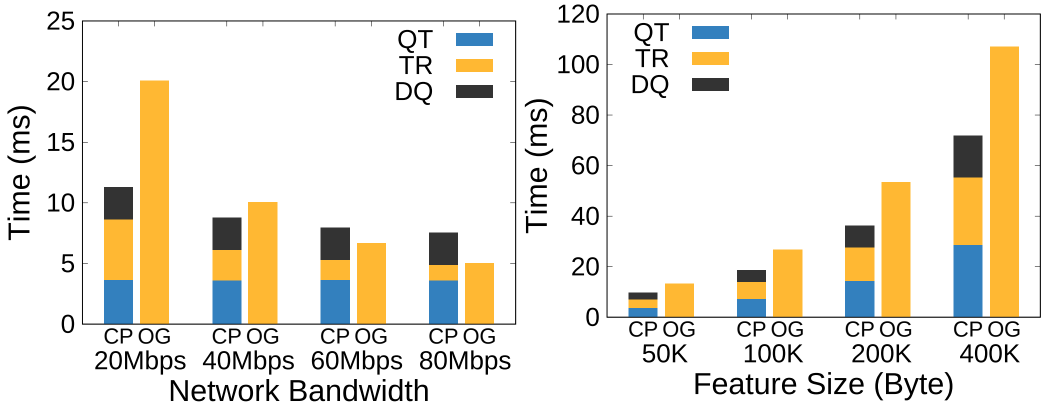


Figure 6.11: An example that shows how network bandwidth (left, with feature size 50 KB) and feature data size (right, with bandwidth 30Mbps) affect compression benefits.

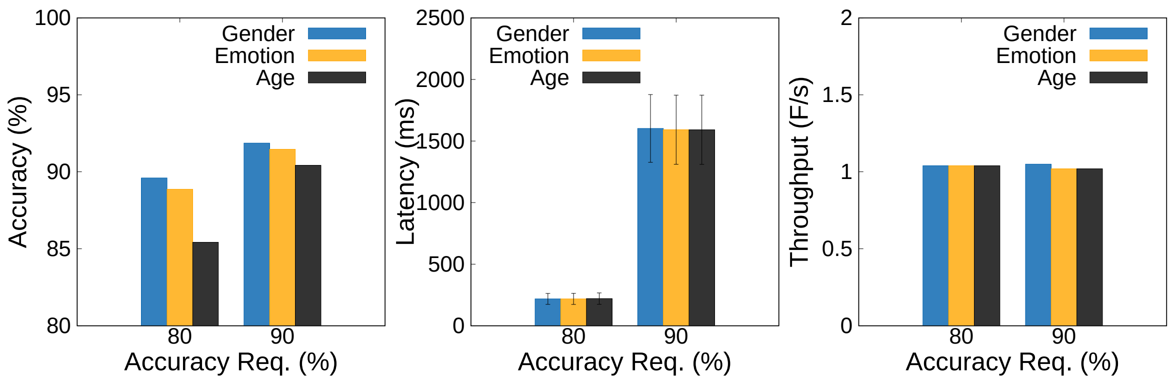


Figure 6.12: AMVP adapts to different accuracy requirement.

original size (OG). When the network bandwidth is low, although (de)quantization takes time, the total amount of time for delivering the feature is still lower than that of delivering it in original size. However, as the network bandwidth increases, transmitting feature in original size becomes a better choice because the (de)quantization time exceeds the saved transmission time. In the right side figure, we show that, when the network bandwidth is low, the larger the feature size is, the more absolute compression benefits we can obtain by using the quantization-based compression. In AMVP, system can adaptively choose to use the quantization-based compression method based on the network condition.

Adapting to different accuracy requirement: The user of AMVP may have different accuracy requirements. In Figure 6.12, we show how AMVP adapts to different accuracy requirements by

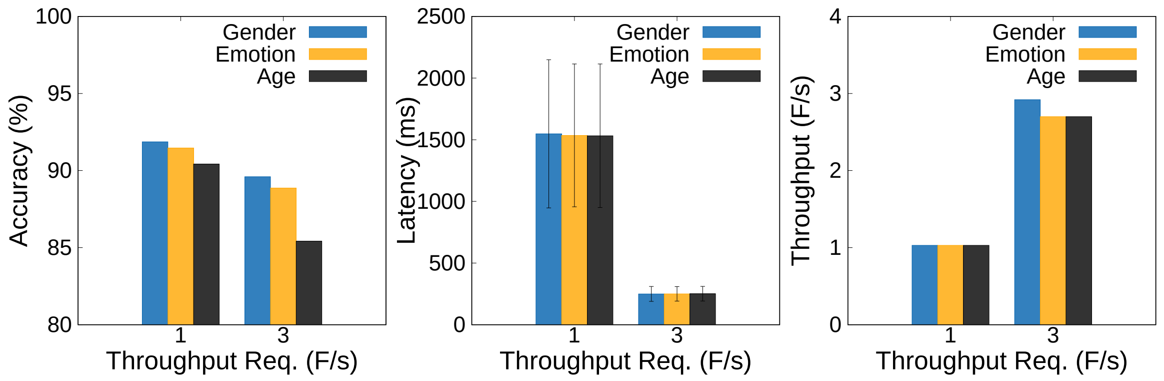


Figure 6.13: AMVP adapts to different throughput requirement.

trading off other metrics. As we can see, the first user has an accuracy goal 80% for all three vision tasks, which is not very high. He also has a latency goal 1000ms and a throughput goal 1F/s. Since he does not have strong preference on the accuracy goal, he sets α, β, γ all equal to 0.33. The second user, however, has a higher accuracy goal 90% and he has a latency goal 1000ms and a throughput goal 1F/s as well. Since he has strong preference on achieving the accuracy goal, he sets $\alpha = 0.8$, $\beta = 0.1$ and $\gamma = 0.1$, respectively. Based on the different accuracy goals and preference, AMVP offers a solution with lower accuracy and lower latency for the first user. For the second user, however, AMVP offers a solution with higher accuracy by trading off the latency. Both solutions meet the throughput goals.

Adapt to different throughput requirement: Video stream can be fed into AMVP at different frame rate, which requires AMVP to provide different throughput. As shown in Figure 6.13, when the input rate is 1F/s, AMVP provides a solution with high accuracy by using the ResNet50V2-based model for each task, which also has higher latency. However, when the input rate becomes 3F/s, AMVP provides another solution by using the MobileNetV2-based models. This solution provides 2.8x higher throughput and 5x lower latency than the original.

Adapting to different latency requirement: Sometimes, an AMVP user wants to instantly get an inference result, which requires AMVP to provide a short latency. As shown in Figure 6.14, when the latency required by the user is high (1500ms), AMVP provides a solution with a high latency and a high accuracy. However, once the latency requirement becomes low (200ms), AMVP will

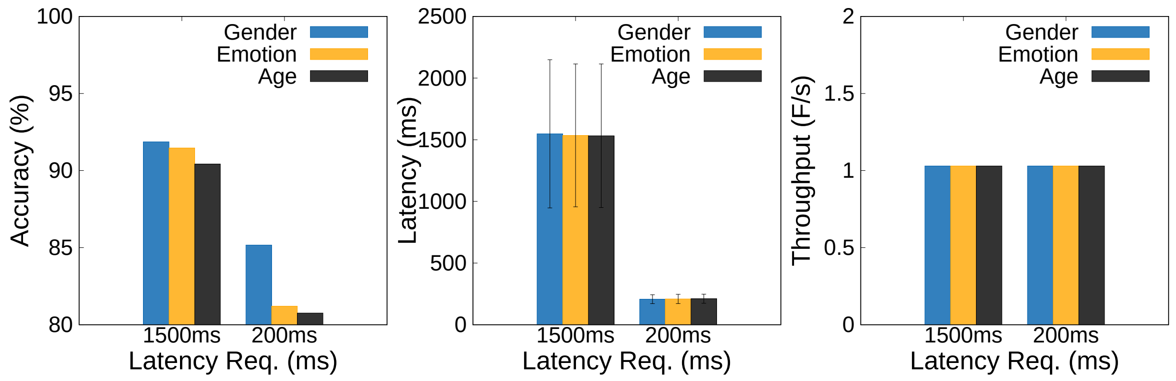


Figure 6.14: AMVP adapts to different latency requirement.

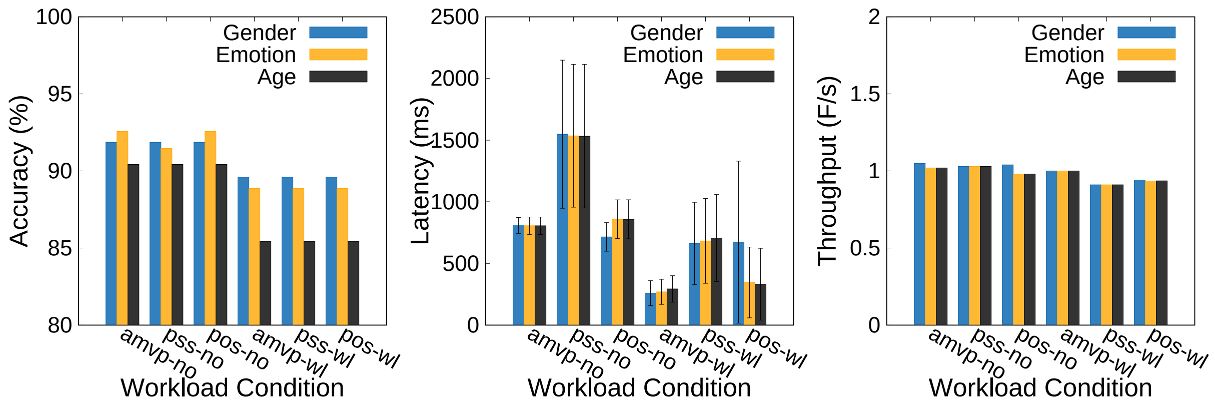


Figure 6.15: AMVP adapts to different computing resource.

choose another solution with lower latency, as well as lower accuracy. The throughput in both solutions are equal to the input rate.

Adapt to different computing resource: The actual available computing resource of a mobile device at the edge is uncertain because other applications also consume it. In Figure 6.15, we show how AMVP adapts to changing computing resources of a mobile device by running a workload application to consume a lot of computing resources. We compare AMVP with two other strategies, i.e., the Pure Sharing Strategy (PSS), which shares layers among multiple CNNs on the same device, and the Pure Offloading Strategy (POS), which offloads CNNs to other devices, to show the superiority of AMVP. As we can observe from the left figure, when there is a workload application running on a device, all the strategies choose to execute CNNs with lower accuracy to reduce the computation cost. From the middle figure, we observe that: (a) when there is no workload,

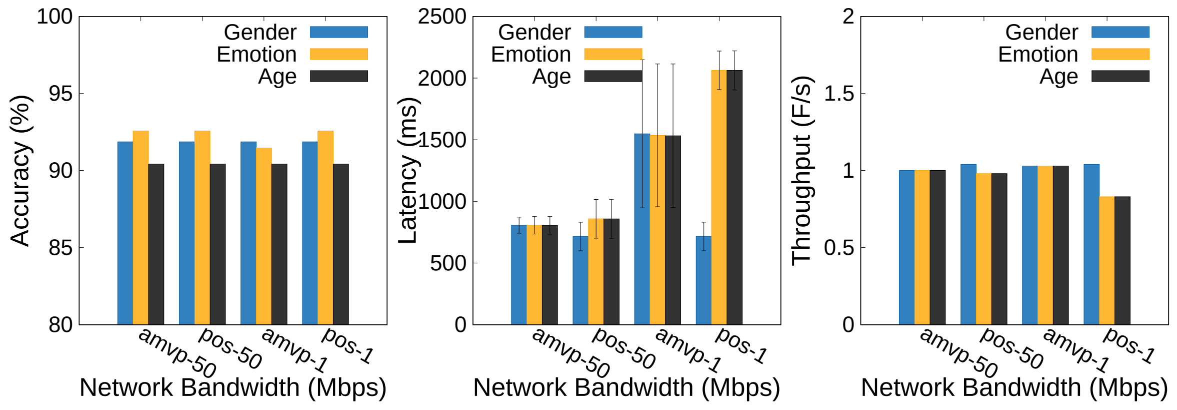


Figure 6.16: AMVP adapts to different network bandwidth.

AMVP achieves up to 48% shorter latency than PSS by offloading CNN layers to other devices and around 6% shorter latency than POS by sharing common layers among CNNs; (b) when there is workload, although all the three strategies choose a lightweight MobileNetV2-based model for each task, AMVP achieves up to 61% shorter latency than PSS and POS; (c) for POS in the workload running case, GenderNet executing on the mobile device with the workload has obviously higher latency than EmotionNet and AgeNet running on other devices. From the right figure, we see that, when there is no workload, all three strategies achieve similar throughput. However, when there is workload running, AMVP achieves around 10% higher throughput than PSS and around 7% higher throughput than POS.

Adapt to different network bandwidth: The edge network is dynamic: sometime, the network bandwidth is large; some other time, it is small. In Figure 6.16, we show how AMVP outperforms POS by running application under different network bandwidth. As we observe in the left figure, when the network bandwidth drops from 50Mbps to 1Mbps, AMVP sacrifices the accuracy of EmotionNet a little by selecting a model with more frozen layers so that different CNNs can share more layers. However, for POS, it still uses the original models because it does not care about layer sharing. From the middle figure, we find that, when the network bandwidth becomes 1Mbps, AMVP chooses to executing all the CNNs locally while POS still offloads two CNNs to other devices, which causes huge communication latency. From the right figure, we find that, AMVP achieves around 1F/s throughput for all tasks in both 50Mbps and 1Mbps network scenarios. However, for POS, since it offloads two tasks to other devices when the network bandwidth is 1Mbps, the throughput of those tasks decreases.

7. EAR: ENERGY-AWARE RISK-AVERSE ROUTING*

The Emergency Operation Center (EOC), which is responsible for scheduling rescue tasks, relies on data collected by first responders at different disaster sites to make informed decisions [128]. However, natural disasters damage the infrastructures of the affected areas, making communication in the disaster area extremely challenging. In this case, Disaster Response Networks (DRNs) are proposed to provide a temporary communication infrastructure. In DRNs, battery-powered wireless routers are deployed at disaster sites [39]. First responders send the processing results back to EOC via the wireless routers. Since the end-to-end connections are not available in DRNs, packets are stored in the routers temporarily. Vehicles (e.g., ambulances, supply vehicles, patrol cars) with on-board routers move around the disaster area, collect packets stored in the routers, carry and forward them to the intermediate (disaster sites or vehicles) or destination (EOC) node [40].

Many routing protocols have been proposed to support the communication in general DTNs [41–44], which improve the routing metrics PDD and PDR by using unlimited level of packet replication [45]. This approach, however, is not energy-efficient, because unlimited level of packet replication costs high TTE, which depletes the batteries of the fixed routers soon. To prolong the lifetime of DRNs, the first research question arises: *how to restrict the level of packet replication to reduce TTE, while maintaining other metrics like PDD and PDR at an adequate level*?

Moreover, most existing routing protocols only address routing metrics such as PDD, PDR and TTE. However, in DRNs, PDS is also significant. For example, a path with the minimum PDD might have a large PDS, which makes the actual delivery delay of some packets much larger than expected. If the first responders prefer a stable delay, the path with the minimum PDS is actually a better choice. To satisfy the preference of different users, the second research question arises: *how to allow first responders to express their preferences on PDD or PDS, when choosing routing paths in DRNs*?

*Reprinted with permission from "EAR: energy-aware risk-averse routing for disaster response networks" by Mengyuan Chao, Harsha Chenji, Chen Yang, Radu Stoleru, Evdokia Nikolova and Ala Altaweel in Adhoc Networks, 2020, Copyright 2020 by Elsevier.

Furthermore, first responders sometimes have the requirements to deliver some urgent packets to the destination before a deadline. Otherwise, the value of these packets will be discounted. To this end, the third research question arises: *how to maximize the probability of delivering packets to the destination before a given deadline*?

To answer the three questions above, we present the Energy-Aware Risk-averse routing framework (EAR). EAR models the mobility in DRNs with a stochastic multigraph [129]: the nodes represent different centers and the weighted edges represent the delivery delay of packets between centers via different types of vehicles. When first responders send data packets, multiple routing paths can be utilized, each with a routing metric formulated by the Mean-Risk model [130], where the PDD and PDS are combined into a single "risk" metric denoted by $PDD + \rho * PDS$. The DRNs users can trade off PDD with PDS by choosing different ρ values. In addition, EAR applies the Max-Probability model [131] to improve the probability of delivering packets to the destination before a given deadline. Furthermore, to improve PDE, which measures *the ratio of the total distinct delivered packets over the total packet transmissions* [132], EAR utilizes a parameter L to restrict the level of packet replication and a differentiated service model to maintain other routing metrics at an adequate level. EAR applies a "λ-optimal" algorithm to find multiple paths with the optimal metrics to perform source routing. Simulation results show that EAR provides flexible control of the routing risk and delivers packets to the destination in a more energy-efficient way (up to 8x higher PDE with 4% lower PDR) than other well-known DTN routing protocols Prophet, MaxProp, RAPID and Spray&Wait.

Out contributions in this chapter are summarized as the following: (1) we apply the risk-aversion concept to address PDS, an important routing metric in DRNs that is overlooked by most previous routing protocols; (2) we use a parameter L to restrict the level of packet replication and analyze its effects on different routing metrics; (3) we apply a differentiated service model to deliver the packets with less TTE while maintaining other metrics at an adequate level; (4) we introduce a "λ-optimal" algorithm [133] to search for the routing path with the lowest risk or the highest probability to deliver the packets before a deadline; (5) we extend the "λ-optimal" algorithm to a multipath version and propose an EAR routing protocol based on it.

7.1 Background

This section first introduces the Post Disaster Mobility (PDM) model [134]. Then, a simple example scenario is used to motivate our research. Finally, the state-of-the-art research is discussed.

7.1.1 Mobility Model

PDM uses two components to model the post disaster scenario: "centers" and "mobile agents (MAs)". Centers are important regions such as EOC, Triage and collapsed buildings, where a wireless router is deployed to support the communication within the range. MAs include supply vehicles (SVs), ambulances (Ambs), patrol cars (Pats) and first responders. They move in the disaster area based on different moving patterns, each with an on-board or hand-held wireless device that collects the buffered packets at one center, carries and forwards them to another.

Each SV is placed at a random center $C_s \in \boldsymbol{C}$ at the beginning. Then, it repeatedly chooses a random center $C_i \in \boldsymbol{C}$ to travel to along the shortest path. Each Amb is initially located at Triage. Then, it chooses a random center $C_i \in \boldsymbol{C}$ to travel to and returns to the Triage upon arriving. This process is repeated, resulting in a series of alternating centers and Triages. Each Pat has a predefined route $\{C_1, C_2, \ldots, C_n, (C_1)\} \in \boldsymbol{C} \times \boldsymbol{C} \times \cdots \times \boldsymbol{C}$. They are placed at a random center C_i initially and travel to C_{i+1} along the shortest path. The process is repeated when arriving at C_{i+1}. For first responders, each member is initially placed at a home center $C_H \in \boldsymbol{C}$. Then, he/she randomly picks a point within the radius r of C_H and travels to it along the shortest path. After arriving at that point, the process is repeated. It should be noted that each category of MAs has its own minimum and maximum speeds. They choose a speed randomly between the minimum and maximum for each leg of travel.

There are two types of data flows in PDM: between centers and within a center. In this chapter, we focus on routing between centers. The routing within a center is regarded as a direct delivery.

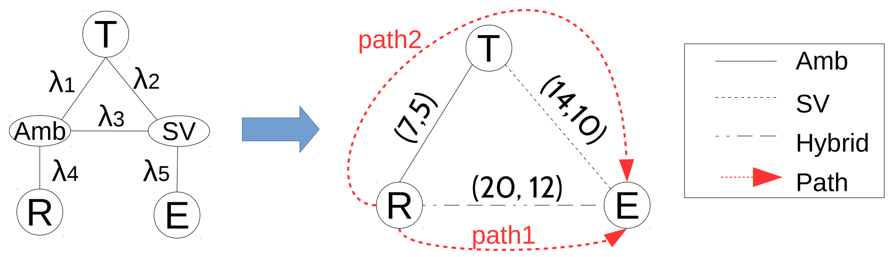


Figure 7.1: A simple example scenario for calculating risk of DRNs routing.

Table 7.1: Two routing paths from a disaster site R to Emergency Operation Center E.

#	Path	Leg1	Leg2	PDD	PDS	P(X<25)%	P(X<45)%
1	R→E	Amb, SV	-	**20**	12	**72.4**	95.9
2	R→T→E	Amb	SV	21	**11.2**	70.2	**96.4**

7.1.2 Risk of DRNs Routing

Based on our PDM, a simple example scenario is depicted in Figure 7.1, where the EOC (E) and a Triage area (T) have been set up and a Rubble pile (R) is identified for search and rescue. Two MAs are moving among these centers: Amb and SV. Due to different moving patterns of MAs, the inter-contact rates between MAs and centers, and between MAs themselves, are different, and are denoted by different λs. Based on the inter-contact information, we construct a stochastic multigraph, where the vertices denote different centers and the edges denote the packet delivery delays between centers via different MAs. The numbers next to each edge indicate (PDD, PDS) of packet delivery delay via different MAs. We illustrate how to construct such graph in Section 3.2.

Now, suppose a first responder located at the Rubble pile R needs to send a report/message to a manager located in EOC E. For this, multiple routing paths can be chosen. In Table 7.1, we list two representative paths: path 1 is through Amb and SV (Hybrid vehicle edge); path 2 is from R to T through Amb and then from T to E through SV. We observe that if the first responder needs a shorter delay, Path 1 with the lower PDD is better. However, if he/she needs a more stable delay, Path 2 with the lower PDS is better. In addition, if he/she wants the message to be delivered in 25 minutes, Path 1 should be chosen to maximize the probability. However, if the deadline is extended to 45 minutes, Path 2 should be chosen instead. As this example shows, based on different preferences, the optimal path is different. In order to adapt to diverse routing requirements of different first responders, an adaptive DTN routing protocol is needed.

7.2 The EAR Routing Framework

This section describes the EAR routing framework. At first, we formulate the routing problem from high level (Section 7.2.1) and construct a stochastic multigraph (Section 7.2.2). Then, we define the risk of DRNs routing by the Mean-Risk model and Max-Probability model [131] (Section 7.2.3). Next, we extend the routing problem from single path routing to multipath routing (Section 7.2.4). Following that we introduce the differentiated service model (Section 7.2.5). Finally, we describe the EAR routing protocol (Section 7.2.6).

7.2.1 Problem Formulation

Given a disaster response scenario with centers $\boldsymbol{C}$ and mobile agents categories $\boldsymbol{M}$, the problem of finding an optimal routing path p from all available paths $\boldsymbol{P}$ for routing from $C_s \in \boldsymbol{C}$ to $C_d \in \boldsymbol{C}$ is formulated as:

$$\begin{aligned} & \min_{p \in \boldsymbol{P}} R_p \;/\; \max_{p \in \boldsymbol{P}} Pr(D_p \leq D), \\ \text{s.t.} \quad & p = (C_s, M_1, C_1, ..., M_i, C_i, ..., M_n, C_d), \\ & C_i \in \boldsymbol{C}, M_i \in \boldsymbol{M}, \end{aligned} \tag{7.1}$$

where $p \in \boldsymbol{P}$ is a routing path consisting of alternating C_i and M_i, R_p is the risk of path p and $Pr(D_p \leq D)$ is the probability that a packet is delivered to the destination before D along path p. The detailed definition of R_p and $Pr(D_p \leq D)$ is given in Section 7.2.3.

7.2.2 Stochastic Multigraph Construction

To describe $R(p)$ and $Pr(D_p \leq D)$ in detail, we at first transfer the PDM model into a stochastic multigraph. In Figure 7.2, centers in the PDM model are mapped to vertices like C_1 and C_2, the delay from C_1 to C_2 via vehicle type V is mapped to a directed edge from C_1 to C_2. The

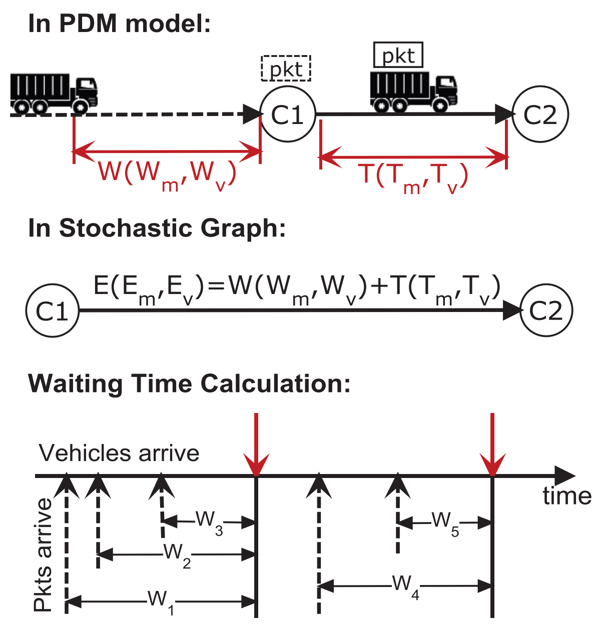


Figure 7.2: Constructing stochastic multigraph for EAR through simulation.

delay consists of two parts: the traveling delay T from C_1 to C_2 via V and the waiting delay W at C_1 for V. The traveling delay $T(T_m, T_v)$, where T_m and T_v represent the mean and variance, respectively, can be calculated by dividing the length of the shortest path between C_1 and C_2 by a random speed in the speed range of V. The waiting delay $W(W_m, W_v)$, however, is not easy to obtain: packets arrive at C_1 at any time, and the time when a vehicle arrives at C_1 and heads to C_2 is determined by its mobility pattern. To calculate $W(W_m, W_v)$, we utilize a new concept called *"Shortest Path Nodes (SPN)"*, which is defined as: for any C_i, if C_2 lies on the shortest path from C_1 to C_i, then $C_i \in SPN(C_1, C_2)$. The idea behind SPN is: if a vehicle goes from C_1 to any center in $SPN(C_1, C_2)$ along the shortest path, it must go through C_2. In other words, if a vehicle of type V arrives at C_1 and heads to C_i, the packets at C_1 with C_2 as the next hop should be forwarded to the vehicle if $C_i \in SPN(C_1, C_2)$. Then, the waiting time at C_1 for V heading to C_2 is calculated as follows: run a simulation in the ONE simulator [46] for a period of time and record all times that vehicles of type V arrive at C_1 and head to $C_i \in SPN(C_1, C_2)$. We assume that packets arrive at C_1 with a Poisson process [82] and record all the arriving times. After getting all the vehicle and packet arrivin

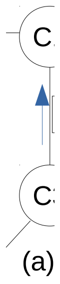


Figure 7.

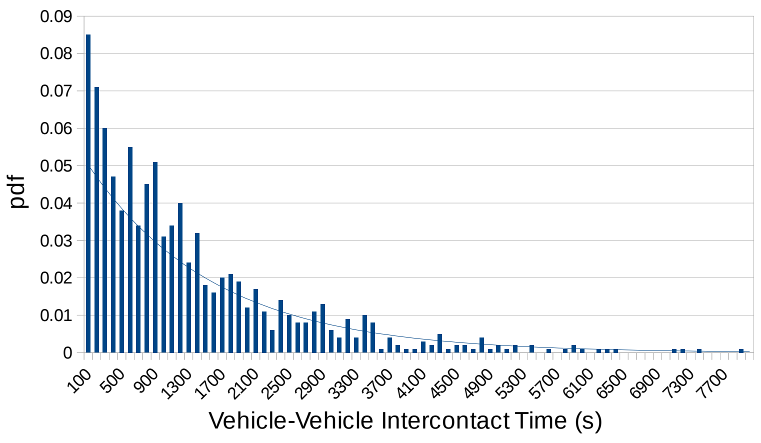


Figure 7.4: Vehicle-center and vehicle-vehicle inter-contact time in DRNs.

Indeed, there are other approaches to build such stochastic graph. For example, one can obtain the inter-contact time (ICT) among vehicles and centers (including vehicle-to-vehicle), either through simulations or by learning from history. Then, the ICT can be used to easily construct such a stochastic graph. We illustrate the process via an example shown in Figure 7.3. In our scenario, we focus on three centers C1, C2, C3 and two mobile agents m1, m2. The mobile agents travel among centers and meet each other, in an opportunistic manner. Through simulations (with results depicted in Figure 7.4) in the ONE simulator, we find that the ICT between vehicles-and-centers and vehicles-and-vehicles in DRNs follows an exponential distribution, which matches the observation in other DTN networks such as social opportunistic networks [135, 136] or VANETs [137–139].

Suppose through simulation (or learning from history), we obtain the inter-contact rate λ_{m2C1} between m2 and C1 and λ_{m2C3} between m2 and C3. Then, the expectation of ICT between m2 and C1 and between m2 and C3 are $1/\lambda_{m2C1}$ and $1/\lambda_{m2C3}$, respectively. Since the inter-contact time between two nodes follows exponential distribution, the packet delivery delay between them follows the same exponential distribution [140, 141]. If the delay of each hop is independent, the sum of these delays will follow a hypo-exponential distribution [142], with mean equal to $\sum_i 1/\lambda_i$ and variance equal to $\sum_i 1/\lambda_i^2$, where λ_i is the inter-contact rate between two nodes of each hop. Then, an edge from C1 to C3 via m2 can be represented as (M_{C1m2C3}, V_{C1m2C3}), where $M_{C1m2C3} = 1/\lambda_{m2C1} + 1/\lambda_{m2C3}$ and $V_{C1m2C3} = 1/\lambda^2_{m2C1} + 1/\lambda^2_{m2C3}$. The opportunistic nature of communication is also employed by allowing vehicle-to-vehicle contacts to relay packets. More specifically, a "virtual vehicle" between two centers can be created whenever multiple vehicles meet in an opportunistic manner. This is illustrated in Figure 7.3(c) by the edge $m2m1$ between $C3$ and $C2$, where $M_{C3m2m1C2} = 1/\lambda_{m2C3} + 1/\lambda_{m1m2} + 1/\lambda_{m1C2}$ and $V_{C3m2m1C2} = 1/\lambda^2_{m2C3} + 1/\lambda^2_{m1m2} + 1/\lambda^2_{m1C2}$.

7.2.3 Mean-Risk and Max-Probability Models

This subsection introduces how to use the Mean-Risk model and the Max-Probability model to formulate R_p and $Pr(D_p \leq D)$ defined in Section 7.2.1.

The Mean-Risk Model: A path is regarded as "risky" if it has a high probability of realizing a much larger delay than expected [129]. To quantify this "risk", EAR adopts the Mean-Risk model proposed in [131]: the risk R_p of a path p is defined as $R_p = \mu_p + \rho * \sigma_p$, where μ_p is the expectation of the delay, σ_p^2 is the variance of the delay and $\rho(\rho \geq 0)$ is a coefficient representing the importance of variance. $\rho = 0$ helps choosing the path with the lowest mean while a $\rho = \infty$ helps choosing the path with the lowest variance. However, it should be noted that the risk of the path does not equal to the sum of its edges' risks, i.e.,

$$R_p = \sum_{e \in p} \mu_e + \rho * \sqrt{\sum_{e \in p} \sigma_e^2} \neq \sum_{e \in p} R_e = \sum_{e \in p} \mu_e + \rho * \sum_{e \in p} \sigma_e. \tag{7.2}$$

Due to lack of sub-optimality, finding path with the lowest risk is challenging, because any dynamic programming approach based on substructure would fail. To deal with this, we need a λ-optimal algorithm introduced in Section 7.3.

The Max-Probability Model: To maximize the probability of delivering a packet to a destination before a deadline D, EAR adopts the Max-Probability model [131], which defines $max\{Pr(D_p \leq D)\}$ as the objective, where D_p represents the end-to-end delay of path p. According to *one side Chebyshev inequality*, we get the upper bound for the probability $Pr(D_p \leq D)$ as follows:

$$\begin{cases} Pr(D_p \leq D) \leq 1 - \frac{1}{1+\varphi_p^2}, & \text{if } D \geq \sum_{e \in p} \mu_e, \\ Pr(D_p \leq D) \leq \frac{1}{1+\varphi_p^2}, & \text{if } D \leq \sum_{e \in p} \mu_e, \end{cases} \tag{7.3}$$

$$\varphi_p = \frac{D - \mu_p}{\sigma_p} = \frac{D - \sum_{e \in p} \mu_e}{\sqrt{\sum_{e \in p} \sigma_e^2}}. \tag{7.4}$$

To achieve a larger probability $Pr(D_p \leq D)$, a higher bound is preferred. For both cases, the highest bounds are achieved when φ_p achieves the maximum value. Similar to R_p, φ_p does not have an optimal substructure. Therefore, a λ-optimal algorithm is needed to find the path with the maximum φ_p as well. However, since the algorithm for the Max-Probability model is already given in [133], we omit it in this chapter.

7.2.4 K-Safest Paths Problem (KSfP)

In EAR, the "safest" path on the stochastic multigraph is the one with the lowest R_p for the Mean-Risk model or the one with the highest φ_p for the Max-Probability model. To further improve the performance, multiple routing paths should be utilized, which leads to the K-Safest Paths Problem (KSfP). The objective of KSfP is to choose the K "safest" paths from the stochastic graph $\mathcal{S}$ for a pair of source and destination nodes.

We solve KSfP from the $K = 1$ case. As described above, neither R_p nor φ_p has the sub-optimality and, thus, no dynamic programming approaches based on substructure work for them. To deal with this issue, we utilize a λ-optimal algorithm [133] to search for the paths with the optimal nonlinear metrics by iteratively using a linear metrics $\mu_p + \lambda_i * \sigma_p^2$, which starts from the $\lambda_0 = 0$, $\lambda_1 = \infty$ cases and updates λ_i towards the one achieving a better original nonlinear metric. The λ-optimal algorithm for the Max-Probability model is already given in [133]. In this chapter, we prove that it works for the Mean-Risk model as well (Section 7.3.2). Then, to solve the $K > 1$ case, we extend the simple path λ-optimal algorithm to a multipath version. To avoid repetition, we only illustrate how it works for the Mean-Risk model in Section 7.3.3 but omit the corresponding part for the Max-Probability model.

7.2.5 Differentiated Service Model

To prolong the lifetime of DRNs, EAR improves PDE by restricting the level of packet replication, while maintaining other routing metrics (such as PDD, PDS and PDR) at an adequate level with a differentiated service model. In this model, we classify packets into four priority classes, namely *urgent*, *regular*, *non-urgent* and *custom*: The *urgent* packets prefer low PDD; the *non-urgent* packets prefer low PDS and less TTE; the *regular* packets, however, prefer a balance among PDD, PDS and TTE; the *custom* packets, specifically, allows the user to set a specific deadline D. When a user sends a packet, he/she needs to choose a specific class. Meanwhile, we classify the

Table 7.2: Differentiated service model in EAR.

	urgent	**regular**	**non-urgent**	**custom**
sufficient	$\rho = 0, K = L$	$\rho = 1, K = \frac{L}{2}$	$\rho = 10, K = 1$	$D, K = f(D)$
insufficient	$\rho = 0, K = 1$	$\rho = 1, K = 1$	$\rho = 10, K = 1$	$D, K = 1$

energy condition of DRNs into two classes, namely *sufficient* and *insufficient*: In *sufficient* case, the average remaining battery of fixed routers is more than 20%, while in *insufficient* case, the average remaining battery of fixed routers is less than 20%. The energy condition of DRNs determines how many packet replicas can be used for the packet transmission.

When transmitting a non-custom packet, the risk-averse parameter ρ and the number of replicas K are chosen based on the rules in Table 7.2, where $L(L \geq 2)$ is a system parameter. When transmitting a custom packet, EAR chooses the number of packet replicas based on the length of D. More specifically, it divides D into two cases. One is the long deadline case, which means $D \geq 2 * PDD_{st}$; the other is the short deadline case, which means $D < 2 * PDD_{st}$. PDD_{st} is the delay of the shortest path obtained from the Dijkstra's algorithm. In the long deadline case, EAR can set $K = L$ to make PDR approaching 100% or set $K = 1$ to save energy. In the short deadline case, EAR can set $K = L$; otherwise, PDR becomes too low. However, if the energy condition is insufficient, EAR will set $K = 1$ to save energy.

7.2.6 The EAR Routing Protocol

The EAR routing protocol performs source routing: when K "safest" paths are obtained, each path is stored in the head of each packet replication at the source node. Then, K replicated packets are transmitted along the K paths to the destination. Based on the current carrier of packets, the forwarding strategies are classified into two cases:

Case (a): the packets are currently at center C_i. Then, only when 1) the next hop C_j of vehicle V is decided, 2) "$C_i - V - C_k$" are in the routing paths of the packets, 3) $C_j \in SPN(C_i, C_k)$ and 4) these packets do not exist in vehicle V, will the packets be sent to the vehicle V when it passes by.

Case (b): the packets are currently carried by vehicle V. Then, only when "$V - C_i$" is a segment of the routing paths (or C_i is in the remaining paths) and these packets do not exist in center C_i,

Algorithm 6: The EAR routing protocol

```
Input  : PDM scenario $\mathcal{P}$, source $s$, destination $d$
Output: None
1  if $s$ and $d$ are Centers then
2      Build the stochastic multigraph $\mathcal{S}$ using $\mathcal{P}$
3      $K, \rho \leftarrow$ Differentiated Service Model
4      Paths $\mathbb{K} \leftarrow \lambda$-optimal alg. with input $\mathcal{S}, K, \rho, s, d$
5      Source routing along the paths in $\mathbb{K}$
6  else if $s$ is a first responder, $d$ is a Center then
7      $c_s \leftarrow$ the Center around which $s$ works
8      Direct delivery from $s$ to $c_s$
9      Apply EAR at $c_s$ with input $\mathcal{P}, c_s, d$
10 else if $s$ is a Center, $d$ is a first responder then
11     $c_d \leftarrow$ the Center around which $d$ works
12     Apply EAR at $s$ with input $\mathcal{P}, s, c_d,$
13     Direct delivery from $c_d$ to $d$
14 else
15     $c_s, c_d \leftarrow$ the Centers around which $s, d$ work
16     if $c_s = c_d$ then
17         Directly delivery from $s$ to $c_s$ then to $d$
18     else
19         Direct delivery from $s$ to $c_s$
20         Apply EAR at $c_s$ with input $\mathcal{P}, c_s, c_d$
21         Direct delivery from $c_d$ to $d$
```

will the packets be sent to center C_i when the vehicle V passes by it.

Algorithm 6 shows the complete EAR routing protocol. Based on whether the source node s and the destination node d are first responders or centers, EAR divide the routing into four cases. The first case is that both s and d are centers (step 1). In such case, a stochastic multigraph $\mathcal{S}$ is constructed from PDM scenario $\mathcal{P}$ at first (step 2). Then, it chooses proper K and ρ values based on the Differentiated Service Model (step 3). Next, it uses the λ-optimal algorithm (step 4) to find the K safest paths. Finally, the packets are routed along these paths to the destination d (step 5). If s is a first responder (step 6), the packet is first delivered directly to the nearest center (steps 7-8), then the EAR protocol is applied as if the packet was created at the center (step 9). The steps are similar if d is a first responder (steps 10-13). If both s and d are first responders working at the same center (steps 14-16), the source sends the packets to the center and the center sends the packets to the destination (step 17) directly; otherwise, a combination of the previous strategies is applied (steps 18-21).

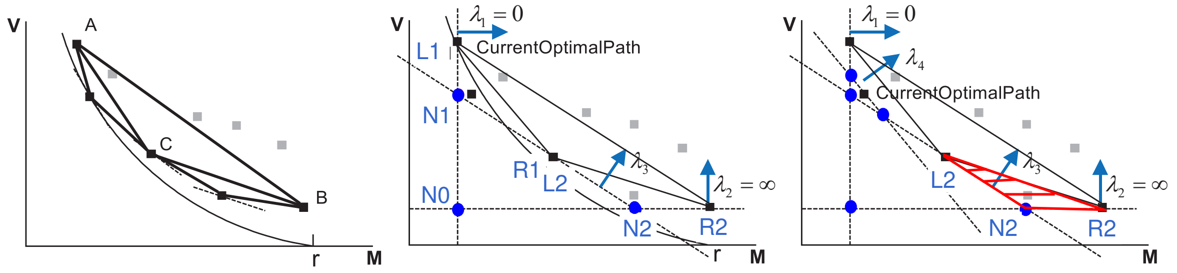


Figure 7.5: Enumerate extreme points (left) and skip unnecessary enumerations (middle, right) to search for the λ-optimal path.

7.3 λ-optimal Algorithm For Mean-Risk Model

In this section, we introduce a λ-optimal algorithm for the Mean-Risk model to find the routing path(s) with the minimum risk(s). At first, we introduce the basic version [143] to explain the high-level idea. Then, we introduce the enhanced version that reduces the time complexity of the basic version by applying three theorems from [133]. Finally, we extend the λ-optimal algorithm from the single path version to a multipath version to find K paths with the minimum risks.

7.3.1 Basic Version

In [143], the authors proposed a basic version λ-optimal algorithm for the Max-Probability Model based on the following two facts: 1) the optimal path only exists in the extreme points of the convex hull for all the candidate paths in the Mean-Variance (M-V) plane; 2) there is a one-to-one correspondence between the extreme points and the breakpoints of the parametric shortest path problem [144] with edge weights $M_e + \lambda * V_e$ (breakpoints mean the λ_is at which the shortest path changes). In [131], the author showed that the two facts hold for the Mean-Risk Model as well. Therefore, we develop a similar λ-optimal algorithm for the Mean-Risk Model as follows:

Step 1: Search two optimal paths for the linear metric $M_p + \lambda V_p$ with $\lambda = 0$ and $\lambda = \infty$, respectively. Their projections on the M-V plane are points $A(M_0, V_0)$ and $B(M_\infty, V_\infty)$ in Figure 7.5 (left). Compare these two paths with the original metric $M_p + \rho\sqrt{V_p}$, choose the one with better metric as the current optimal path.

Step 2: Calculate the slope k_{AB} of line AB, which generates a new $\lambda_{AB} = \frac{-1}{k_{AB}}$. Then, find the optimal path for $M_p + \lambda_{AB} V_p$ by assigning $M_e + \lambda_{AB} V_e$ as the edge weights. Since $M_p + \lambda_{AB} V_p = \sum_{e\in p}(M_e + \lambda_{AB} V_e)$, classic shortest path algorithms can be used to find the optimal path (point C). Next, calculate the original metric $M_p + \rho_{AB}\sqrt{V_p}$ of path C and replace the current optimal path with C if C's metric value is lower than that of the current optimal path.

Step 3: Recursively find new λ-optimal paths between A,C and between C,B with λ_{AC} and λ_{CB}. If the path is identical to the endpoints, the recursive process stops; otherwise, update the current optimal path and λ to do more searching. When the whole procedure stops, return the current optimal path as the optimal for the original metric.

The essence of the algorithm above is a quasi-binary search which exhaustively enumerates all the extreme points with different breakpoints. Therefore, its time complexity depends on the total number of extreme points (breakpoints). In [144], the author proved that there are $N^{\Theta(logN)}$ breakpoints in total for the worst case. Therefore, the time complexity of this λ-optimal algorithm is $O(N^{\Theta(logN)})$, where N is the number of nodes in the network.

7.3.2 Enhanced Version

Although the basic version λ-optimal algorithm can find the optimal path eventually, it is time consuming because it needs to recursively exhaust all the extreme points (breakpoints). In [133], the authors proposed three theorems to eliminate some unnecessary enumerations, which greatly reduces the time complexity. However, *all these theorems are built upon and proved against the Max-Probability model, whether they hold for the Mean-Risk Model remains unclear*. In this chapter, we prove that these theorems hold for the Mean-Risk Model as well. In the following part, we will first give these theorems, then prove the correctness and finally illustrate their benefits.

To begin with, we introduce some technical terms with an example in Figure 7.5 (middle). A parabola $M + \rho\sqrt{V} = r$ intersects with axis M at $M = r$. The square points on the M-V plane represent the candidate paths. Points L_1 and R_2 represent the λ-optimal paths p_0, p_∞ for $\lambda = 0$ and $\lambda = \infty$, respectively. Currently, since the metric value of p_0 is better than that of p_∞, we name p_0 as the current optimal path and its metric value as the current best metric value. We name the line with slope equal to $\frac{-1}{\lambda}$ and going through the λ-optimal path point as the "λ-optimal search line". In Figure 7.5, the 0-optimal search line and the ∞-optimal search line intersect at N_0. Then, with L_1, R_2, we calculate a new $\lambda_{L_1R_2}$ and find the $\lambda_{L_1R_2}$-optimal path at point R_1/L_2. The $\lambda_{L_1R_2}$-optimal search line intersects with the 0-optimal search line and ∞-optimal search line at N_1, N_2, respectively. We name the triangular region $\triangle L_iN_iR_i$ where paths with better metrics might exist as a "candidate region" and the point N_i as the "probe point" of region $\triangle L_iN_iR_i$.

Theorem 1: *If the metric value $M + \rho\sqrt{V}$ of a probe point is larger than the current best, its corresponding candidate region contains no global optimal path.*

Proof 1: Suppose a probe point happens to be a path point, then no other path points in its corresponding candidate region can be the optimal, since the optimal path only occurs among extreme points. If the probe point is not a path point, we can add an imaginary path there. If the metric value of the imaginary path is larger than the current best, the metric values of those real path points in the candidate region can only be larger. Therefore, they can not be global optimal.

Benefit 1: Theorem 1 can reduce the time complexity of the basic version λ-optimal algorithm by removing the candidate region whose probe point satisfies the condition above from recursive searching. Figure 7.5 (right) illustrates this case, where the right candidate region $\triangle L_2N_2R_2$ is removed without searching because the metric value of N_2 is larger than that of the current optimal path (p_0). On the contrary, the left candidate region $\triangle L_1N_1R_1$ is searched since N_1 has a smaller metric value than the current best. Therefore, paths with better metric values might be found.

Theorem 2: *The optimal path can only be found with λ upper bounded by λ_u, the negative inverse of the slope of the tangent to the parabola $M + \rho\sqrt{V} = M_0 + \rho\sqrt{V_\infty}$ at the intersection of the 0-optimal and ∞-optimal search lines. When $\lambda > \lambda_u$, the metric value $M + \rho\sqrt{V}$ of the λ-optimal path can not be smaller than that of the λ_u-optimal path.*

Proof 2: Suppose $\lambda > \lambda_u$, if the λ-optimal path is the same as the λ_u-optimal path, finding the optimal path with λ_u is enough; if the λ-optimal path is different from the λ_u-optimal path, $M_{\lambda_u} \neq M_\lambda$ or $V_{\lambda_u} \neq V_\lambda$. From the definition of λ_u, we get:

$$\lambda_u = \frac{\rho}{2\sqrt{V_\infty}}, \tag{7.5}$$

From the definition of λ-optimal, we get:

$$M_{\lambda_u} + \lambda_u V_{\lambda_u} < M_\lambda + \lambda_u V_\lambda, \tag{7.6}$$

$$M_\lambda + \lambda V_\lambda < M_{\lambda_u} + \lambda V_{\lambda_u}. \tag{7.7}$$

Since $\lambda > \lambda_u$, we get:

$$V_{\lambda_u} > V_\lambda,\ M_\lambda > M_{\lambda_u}. \tag{7.8}$$

Then, according to (5)(6)(8), we get:

$$\frac{M_\lambda - M_{\lambda_u}}{V_{\lambda_u} - V_\lambda} > \lambda_u = \frac{\rho}{2\sqrt{V_\infty}} > \frac{\rho}{2\sqrt{V_\lambda}}. \tag{7.9}$$

By multiplying two sides with $\sqrt{V_{\lambda_u}} + \sqrt{V_\lambda}$, we get:

$$\frac{M_\lambda - M_{\lambda_u}}{\sqrt{V_{\lambda_u}} - \sqrt{V_\lambda}} > \frac{\rho}{2\sqrt{V_\lambda}} \left(\sqrt{V_{\lambda_u}} + \sqrt{V_\lambda}\right) > \rho, \tag{7.10}$$

$$M_\lambda + \rho\sqrt{V_\lambda} > M_{\lambda_u} + \rho\sqrt{V_{\lambda_u}}. \tag{7.11}$$

Therefore, when $\lambda > \lambda_u$, even if the λ-optimal path is different from the λ_u-optimal path, the risk value of it is larger than that of the λ_u-optimal path.

Benefit 2: Theorem 2 can reduce the time complexity of searching the λ-optimal path from $O(N^2)$ to $O(1)$ if λ is larger than λ_u. If a lot of λs satisfy this property, the time complexity of the basic λ-optimal algorithm can be greatly reduced.

Theorem 3: *The optimal path can only be found with λ lower bounded by λ_l, the negative inverse of the slope of the tangent to the parabola $M+\rho\sqrt{V} = M_{copt}+\rho\sqrt{V_{copt}}$ at the intersection of the parabola and 0-optimal search line. When $\lambda < \lambda_l$, the metric value $M + \rho\sqrt{V}$ of the λ-optimal path can not be smaller than either that of the λ_l-optimal path or that of the current optimal path.*

Proof 3: Suppose $\lambda < \lambda_l$, if the λ-optimal path is the same as the λ_l-optimal path, finding the optimal path with λ_l is enough; if the λ-optimal path is different from the λ_l-optimal path, $M_{\lambda_l} \neq M_\lambda$ or $V_{\lambda_l} \neq V_\lambda$. From the definition of λ_l, we get:

$$\lambda_l = \frac{\rho}{2\sqrt{V_d}}, \tag{7.12}$$

where V_d is the value of V at the intersection point. Following similar steps as (6)(7)(8)(9), we get:

$$\frac{M_\lambda - M_{\lambda_l}}{\sqrt{V_{\lambda_l}} - \sqrt{V_\lambda}} < \frac{\rho}{2\sqrt{V_d}}\left(\sqrt{V_{\lambda_l}} + \sqrt{V_\lambda}\right). \tag{7.13}$$

Next, let us consider two cases: (a) $\sqrt{V_\lambda} < \sqrt{V_d}$; (b) $\sqrt{V_\lambda} \geq \sqrt{V_d}$. In case (a), since $\sqrt{V_\lambda} < \sqrt{V_d}$ and similar to (8) we have $V_{\lambda_l} < V_\lambda$, we get:

$$(\sqrt{V_{\lambda_l}} + \sqrt{V_\lambda}) < 2\sqrt{V_\lambda} < 2\sqrt{V_d}. \tag{7.14}$$

Combine (13)(14), we get:

$$\frac{M_\lambda - M_{\lambda_l}}{\sqrt{V_{\lambda_l}} - \sqrt{V_\lambda}} < \rho \Rightarrow M_\lambda + \rho\sqrt{V_\lambda} > M_{\lambda_l} + \rho\sqrt{V_{\lambda_l}}. \tag{7.15}$$

In case (b), since $\sqrt{V_\lambda} \geq \sqrt{V_d}$, $M_\lambda \geq M_0$ and $M_0 = M_d$, we can easily get:

$$M_\lambda + \rho\sqrt{V_\lambda} \geq M_d + \rho\sqrt{V_d} = M_{copt} + \rho\sqrt{V_{copt}} \tag{7.16}$$

Therefore, when $\lambda < \lambda_l$, even if the λ-optimal path is different from the λ_l-optimal path, the risk value of it is either larger than that of the λ_l-optimal path or not smaller than that of the current optimal path.

Benefit 3: Theorem 3 can reduce the time complexity of searching the λ-optimal path from $O(N^2)$ to $O(1)$ if λ is smaller than λ_l. If a lot of λs satisfy this property, the time complexity of the basic λ-optimal algorithm can be greatly reduced.

We combine Theorem 1-3 with the basic version λ-optimal algorithm to get an enhanced version as Algorithm 7 as follows (the underlined part is for the multipath-version we describe later): First, it uses p, q and Q to present the current optimal path, the region being searched and the regions that have not been searched, respectively (step 1). Then, it searches the λ-optimal path for $\lambda = 0$ (path p_0) and for $\lambda = \infty$ (path p_∞) and store the one with a lower risk as the current optimal path (steps 2-5). Next, it puts the region enclosed by p_0, p_∞ and the intersection ($p_0.M$, $p_\infty.V$) into Q as an candidate region (step 6) and updates λ_l, λ_u respectively (step 7). Following that, it goes over each candidate region in Q by: 1) applying Theorem 1 (steps 9-10); 2) calculating a new λ (step 11); 3) applying Theorem 2 (steps 12-16); 4) applying Theorem 3 (steps 17-21); 5) calculating the λ-optimal path for the current λ (steps 22); 6) checking whether the λ-optimal path is different from the left and right points of the region (step 23); 7) updating the current optimal path and λ_l if the λ-optimal path has a lower risk than the current optimal path (step 24-26); 8) adding new candidate regions into Q (steps 27-29). Finally, when all the candidate regions in Q are checked, it will return the current optimal path as the path with the minimum risk (step 32).

In [133], the authors proved that the time complexity of the enhanced λ-optimal algorithm for the Max-Probability model is $O(N^2log^4N)$ on the average. Similarly, the algorithm has an average $O(N^2log^4N)$ time complexity for the Mean-Risk model. The reader can refer to [133] for details.

7.3.3 Multipath Version

In the above section, a λ-optimal algorithm for finding the path with the minimum risk is introduced. However, to further improve routing metrics such as PDD and PDR, multiple routing paths should be used to transmit several replicas for a packet simultaneously. This leads to the "K-safest paths problem" mentioned before, which aims at finding "K paths with the least risks"

Algorithm 7: λ-optimal alg. for Mean-Risk model

```
Input  : multigraph $\mathcal{S}$, source $s$, destination $d$, $\rho$
Output: path $p$ with the least risk
path $p \leftarrow \varnothing$, region $q \leftarrow \varnothing$, regions $Q \leftarrow$ FIFO queue
$p_0 \leftarrow \lambda$-optimal-path(0), $p_\infty \leftarrow \lambda$-optimal-path($\infty$)
if $p_0 = p_\infty$ then
    return $p_0$
else
    $p \leftarrow$ ($p_0$.risk $<$ $p_\infty$.risk) ? $p_0$ : $p_\infty$
$Q$.enqueue(region(l: $p_0$, r: $p_\infty$, prob: ($p_0$.M, $p_\infty$.V)))
calculate $\lambda_l$ and $\lambda_u$
while $q \leftarrow Q.dequeue() \neq \varnothing$ do
    // Theorem 1
    if $q.prob.risk > p.risk$ then
        continue
    $\lambda \leftarrow -\frac{q.l.M - q.r.M}{q.l.V - q.r.V}$
    // Theorem 2
    if $\lambda \geq \lambda_u$ then
        if $\lambda_u$ was not searched then
            $\lambda \leftarrow \lambda_u$
        else
            continue
    // Theorem 3
    if $\lambda \leq \lambda_l$ then
        if $\lambda_l$ was not searched then
            $\lambda \leftarrow \lambda_l$
        else
            continue
    $p_\lambda \leftarrow \lambda$-optimal-path($\lambda$)
    if $p_\lambda \neq q.l$ and $p_\lambda \neq q.r$ then
        if $p_\lambda.risk < p.risk$ (and $p_\lambda \notin \mathbb{K}$) then
            $p \leftarrow p_\lambda$ and update $\lambda_l$
        locate $prob_l$ and $prob_r$ with $\lambda$-optimal line
        $Q$.enqueue(region(l: $q$.l, r: $p_\lambda$, prob: $prob_l$))
        $Q$.enqueue(region(l: $p_\lambda$, r: $q$.r, prob: $prob_r$))
return $p$
```

for the Mean-Risk model. In the following, we illustrate how to solve this problem by extending the λ-optimal algorithm to a multipath version.

First of all, by analyzing the procedure of λ-optimal algorithm, we find that it only searches the triangle region enclosed by p_0, p_∞ and $(p_0.M, p_\infty.V)$. This region is enough for finding the path with the minimum risk because the least risky path must occur on the left side of the line connecting p_0 and p_∞. However, when the number of required paths increases to $K(K > 1)$, algorithms that only search the triangle region might fail, because there might be no sufficient paths in that region. In the following, we consider the K-safest paths problem in two cases.

Case 1: *$N \geq K$ paths exist in the triangle region $L_1N_0R_2$*

We observe that the reason Theorem 1-3 can speed up the basic version λ-optimal algorithm is, it only finds the global optimal solution. Under this target, regions that do not contain the global optimal solution can be ruled out. However, when we extend it to multipath version, those regions can not be ruled out any more, because path points there might be the $2_{nd} - K_{th}$ optimal. With this in mind, we propose to call Algorithm 7 with a slight modification (the underlined part) for K times to find "K paths with the least risks", which works as the following for each iteration i:

When a λ-optimal path is found, first check whether it has already been recorded in the path container $\mathbb{K}$. If it is not recorded and has a better metric value than the current best, replace the current optimal path with the λ-optimal path to continue the searching process; otherwise, just use the λ-optimal path to continue the searching process but do not replace the current optimal path. In the end, add the current optimal path as the i_{th} optimal path into the path container $\mathbb{K}$.

Case 2: *$N < K$ paths exist in the triangle region $L_1N_0R_2$*

In such case, the algorithm above can find N paths with the minimum risks at most, which do not meet the demand. Therefore, instead of trying to find K paths with the minimum risks accurately, we propose an approximate algorithm that uses a similar idea with the λ-optimal algorithm, which works as follows: First, find $P(P \gg K)$ paths that have the smallest $M_p + \lambda V_p$ values with $\lambda = \frac{M_0 - M_\infty}{V_\infty - V_0}$. Then, sort these P paths by the risk metric $M_p + \rho\sqrt{V_p}$ in ascending order. Finally, add the first $(K - N)$ paths that are not in $\mathbb{K}$ from the sorted P paths to finally get "K paths with the minimum risks."

Algorithm 8: Approximate multipath λ-optimal alg.

```
Input  : multigraph $\mathcal{S}$, source $s$, destination $d$, $\rho$, $K$
Output: set $\mathbb{K}$ containing $K$ paths with the least risks
1  path set $\mathbb{K} \leftarrow \varnothing$
2  for $i \leftarrow 1$ to $K$ do
3      $p_i \leftarrow \lambda$-optimal alg. with a tiny adjustment (the underlined red part)
4      $\mathbb{K}$.add($p_i$) // Set $\mathbb{K}$ does not allow duplicated paths
5  if $\mathbb{K}$.size $< K$ then
6      $\mathbb{P} \leftarrow P$ paths with least $M_p + \rho * V_p$ by Dijkstra's alg.
7      sort $\mathbb{P}$ by $M_p + \rho * \sqrt{V_p}$ in ascending order
8      for $j \leftarrow 1$ to $P$ and $\mathbb{K}$.size $\neq K$ do
9          if $p_j \notin \mathbb{K}$ then
10             $\mathbb{K}$.add($p_j$)
11 return $\mathbb{K}$
```

By combining Case 1 and Case 2, we get a complete multipath λ-optimal algorithm as Algorithm 8. As we mentioned eariler, the time complexity for the λ-optimal algorithm is $O(N^2log^4N)$. Then, the time complexity for Case 1 is $K * O(N^2log^4N) = O(KN^2log^4N)$. For Case 2, since it calls the Dijkstra's algorithm for P times, the time complexity is $P * O(N^2) = O(PN^2)$. Then, the time complexity for the whole algorithm is $O(KN^2log^4N + PN^2)$.

7.4 Evaluation

In this section, we evaluate the performance of EAR by both mathematical analysis and simulations. At first, we analyze the effects of risk-aversion and level of replication on different routing metrics. Because of space limitations, we only present the analysis for the Mean-Risk model. Then, we introduce the experimental setup and present the experimental results for both Mean-Risk and Max-Probability models with different parameters. Finally, we compare the performance of EAR with other 4 well-known DTN routing protocols.

7.4.1 Mathematical Analysis

Suppose set P has n random variables, which represent the delays of all paths from a given source to a given destination in the stochastic multigraph. Each of these random variables P_i has an associated mean and standard deviation $(M_i, \sqrt{V_i}) = (PDD_i, PDS_i)$. According to the Mean-Risk model, each P_i is assigned with a scalar risk metric $(R_i = PDD_i + \rho * PDS_i)$ for a risk-aversion coefficient $\rho \geq 0$. This set of n variables is ordered according to the risk metric, resulting in a set $P = \{P_1, P_2, \ldots, P_n\}$, where for $\forall i < j$, $R_i < R_j$. If there is no packet replication, data is sent on P_1 only. However, with K replications, the first K paths of P, i.e. $\{P_1, P_2, \ldots, P_K\}$ are chosen. When any of these replications arrives at the destination, the packet is delivered. Therefore, the delivery delay is $mean(\rho, K) = E[W]$, and the standard deviation of delivery delay is $\sqrt{var(\rho, K)} = \sqrt{V[W]}$, where the random variable $W = min\{P_1, P_2, \ldots, P_K\}$. These P_is are supposed to be independent distributed because of the following assumptions: 1) the storage size at each node is assumed to be infinite, so there is no conflict in packet storage; 2) the waiting and traveling delays, rather than packet transmission delays between vehicle and center, dominate the packet delivery delay, so there is no conflict in the bandwidth usage. Therefore, the Cumulative Distribution Function (CDF) of W is

$$F(W \leq x) = 1 - \prod_{i=1}^{K}(1 - F(P_i \leq x)) \tag{7.17}$$

Once we get CDF of W, its mean and variance are:

$$mean(\rho, K) = E[W] = \int_{-\infty}^{\infty} x f(x)\, dx \tag{7.18}$$

$$var(\rho, K) = V[W] = \int_{-\infty}^{\infty} (x - E[W])^2 f(x)\, dx \tag{7.19}$$

where $f(x)$ is the probability density function of $F(x)$.

Based on the above equations, we get some analysis results in Table 7.3, where *an intentional effect changes a metric by design and an incidental effect does so implicitly* [44]. In the following,

Table 7.3: Effects of ρ and K on routing metrics in EAR.

	PDD	**PDS**	**PDR**	**Energy**
ρ	Intentional	Intentional	Incidental	Incidental
K	Intentional	Incidental	Intentional	Intentional

we describe them in detail.

Result 1: When $K = 1$, as the value of ρ increases from ρ_1 to ρ_2, $var(\rho, K)$ decreases while $mean(\rho, K)$ increases. This result is proved by using the following two inequalities:

$$m_1 + \rho_1 * \sqrt{v_1} \leq m_2 + \rho_1 * \sqrt{v_2}$$
$$m_2 + \rho_2 * \sqrt{v_2} \leq m_1 + \rho_2 * \sqrt{v_1}$$

where $m_i = mean(\rho_i, K = 1)$ and $v_i = var(\rho_i, K = 1)$ represents the mean and variance of the delay of the optimal path for ρ_i. We get that $(\rho_1 - \rho_2)(\sqrt{v_1} - \sqrt{v_2}) \leq 0$. Since $\rho_1 < \rho_2$, we get $v_1 \geq v_2$, i.e., $var(\rho_1, K = 1) \geq var(\rho_2, K = 1)$. Given this, we can get $m_1 \leq m_2$, i.e., $mean(\rho_1, K = 1) \leq mean(\rho_2, K = 1)$. Therefore, when $K = 1$, ρ has an intentional effect on PDD and PDS.

Result 2: When $K \geq 2$, there is no guarantee that the conclusion in Result 1 still holds. For example, suppose ρ_1 chooses the path set $P^{\rho_1} = \{P_1^{\rho_1}, P_2^{\rho_1}, \ldots, P_K^{\rho_1}\}$ and ρ_2 chooses the path set $P^{\rho_2} = \{P_1^{\rho_2}, P_2^{\rho_2}, \ldots, P_K^{\rho_2}\}$. Although the risk relationship among paths in the path sets P^{ρ_1}, P^{ρ_2} and between the optimal paths $P_1^{\rho_1}$ and $P_1^{\rho_2}$ are clear, the risk relationships among other paths are indeterminable. Therefore, it is difficult to infer the relationship of means and variances among these paths. In addition, it is more difficult to infer the relationship between $W^{\rho_1} = min\{P_1^{\rho_1}, P_2^{\rho_1}, \ldots, P_K^{\rho_1}\}$ and $W^{\rho_2} = min\{P_1^{\rho_2}, P_2^{\rho_2}, \ldots, P_K^{\rho_2}\}$, because the variance relationship between W^{ρ_i} and $\{P_1^{\rho_i}, P_2^{\rho_i}, \ldots, P_K^{\rho_i}\}$ is indeterminable. Furthermore, as K increases, the effect of adjusting ρ becomes less and less pronounced. It is difficult to reduce risk by adjusting ρ when K is large, because the order of $F(P_i \leq x)$ in Equation 7.17 does not matter since it is a product, and ρ only changes the order of selected random variables.

Result 3: From Equation 7.18, we claim that as K increases, $mean(\rho, K)$ decreases, which means K has an intentional effect on PDD. We provide an informal proof below. First, we transfer the continuous expression of $mean(\rho, K)$ to a discrete expression $\lim_{N\to\infty} \frac{\sum_{j=1}^{N} mean(\rho,K)_j}{N}$, where $mean(\rho, K)_j$ represents a sample of $mean(\rho, K)$. Then, according to the meaning of $mean(\rho, K)_j$, we have $mean(\rho, K)_j = min\{P_1^j, P_2^j, \ldots, P_K^j\}$, where P_i^j is a sample of P_i. Similarly, we can define $mean(\rho, K+1)$ as $\lim_{N\to\infty} \frac{\sum_{j=1}^{N} mean(\rho,K+1)_j}{N}$, where $mean(\rho, K+1)_j = min\{P_1^j, P_2^j, \ldots, P_K^j, P_{K+1}^j\}$. Given the expressions of $mean(\rho, K+1)_j$ and $mean(\rho, K)_j$, it is easy to prove that $mean(\rho, K+1)_j < mean(\rho, K)_j$. Therefore, $mean(\rho, K+1) < mean(\rho, K)$, i.e., $mean(\rho, K)$ decreases as K increases. Surprisingly, there is no guarantee that such a claim always holds for PDS [145], which means K has an incidental effect on PDS.

Result 4: From Equation 7.17, we observe that as K increases, $F(W \leq x)$ increases as well, which means K has an intentional effect on PDR. Similarly, K has an intentional effect on the TTE, because as K increases, the number of packet relays, and hence TTE increases as well. However, we can not say how PDR or TTE will change with different ρ, because ρ only changes the order of paths but not the number of paths. Thus, ρ has an incidental effect on PDR and TTE.

7.4.2 Experimental Setup

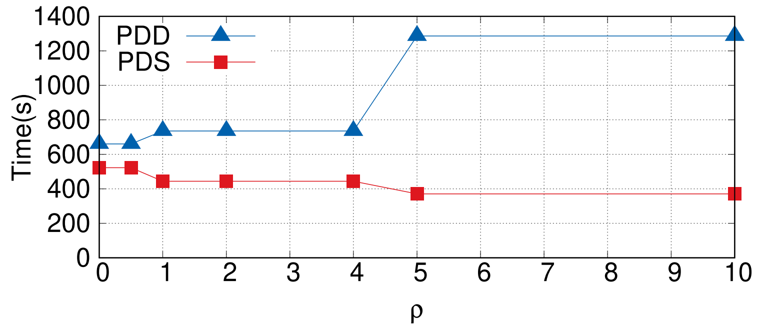
Experiments are performed using The ONE simulator [46] with the PDM mobility model on the Helsinki street map. An EOC and a Triage are set up, with 8 collapsed buildings located for search and rescue. 5 ambulances and 15 supply vehicles move in the city by following corresponding mobility models with the speed at $12-15m/s$ and $15-20m/s$, respectively. 2 patrol cars move along the pre-planned routes with the speed at $6-9m/s$. The data traffic is generated as a Possion process [82] at collapsed buildings sent to the EOC during time $10,000s$–$50,000s$. The packet size is $40KB$, and the contact bandwidth is 80 Mbps. The simulation lasts for $180,000s$ and each point in the figures represents the average value of 32 random runs. The metrics we measure include PDD, PDS, PDR, PDE and TTE. The DTN protocols chosen for comparison include Prophet, MaxProp, RAPID and SprayWait (with 3 replicas, SW3), where Prophet and MaxProp attempt to achieve high PDR through unlimited level of packet replication, RAPID aims to intentionally reduce PDD based on the marginal utility of increasing packet replicas and SprayWait improves PDE by restricting the level of packet replication.

For the Mean-Risk model, since we want to study the end-to-end delay of all packets, we set the Time To Live (TTL) value as $36,000s$, which is long enough for the packets to arrive at the destination. We study the effects of K and ρ on different metrics with different settings. At first, we measure the effects of ρ on PDD and PDS for a fixed K. Then, for a fixed ρ, we increase the value of K to see how it affects PDD and PDS. Finally, we choose three representative (K, ρ) settings to compare EAR with other DTN protocols in terms of PDD, PDS and TTE. For the Max-Probability model, we conduct experiments with different deadlines. At first, we measure the effects of K on PDR and PDE. Then, we compare EAR with other DTN protocols in terms of PDR, PDE and TTE.

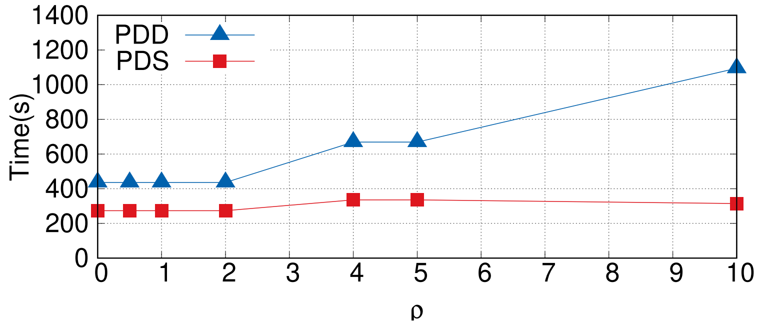
7.4.3 Evaluation of Mean-Risk Model

For a given K, increasing ρ. In Figure 7.6 (a), as $K = 1$ and ρ increases, PDS decreases while PDD increases, which matches the analysis result from Result 1 that ρ has an intentional effect on PDD and PDS when $K = 1$. This property is paramount for disaster response, because it enables First responders to achieve desirable PDD and PDS based on their preference. From the changing trend of PDD and PDS, we can observe that they are not strictly increasing/decreasing with the increase of ρ. This is because, when two ρ values are close to each other, the λ-algorithm returns the same path. Another interesting observation in Figure 7.6 (b) is that when $K \geq 2$, even if PDD increases with increasing ρ, PDS does not decrease as in the $K = 1$ case. This proves our analysis result from Result 2 that when $K \geq 2$, there is no guarantee how ρ will affect PDD and PDS. Moreover, from the PDD/PDS results in Figure 7.6 (c), we find that when K is large, the effects of ρ on PDD and PDS become small, which matches our analysis result from Result 2 as well.

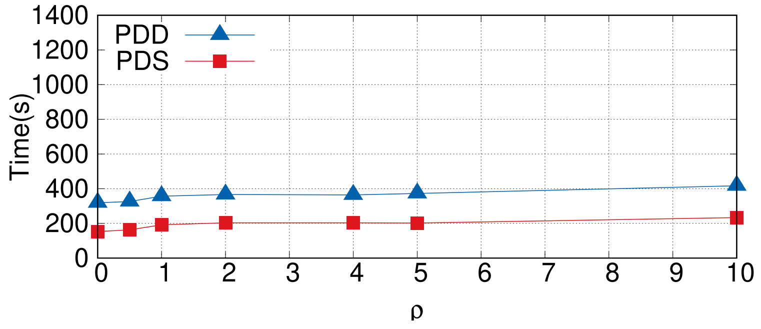
For a given ρ, increasing K. In Figure 7.7, no matter what the value of ρ is, increasing K always decreases PDD, just as our analysis in Result 3. This implies that no matter what the user's preference is, increasing the level of packet replication (the number of routing paths) always helps delivering data packets to the destination in a shorter time on average. This property is important as it allows first responders to achieve better PDD by simply increasing the level of



(a) $K = 1$



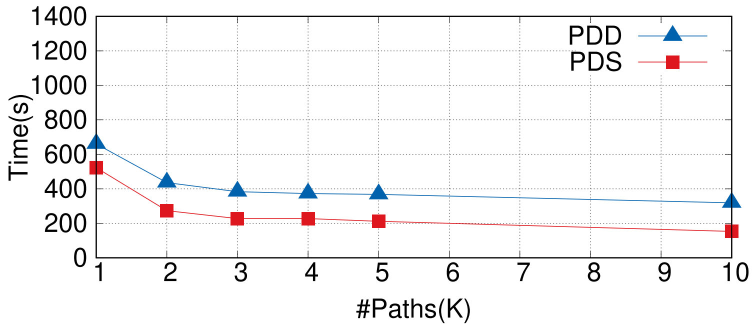
(b) $K = 2$



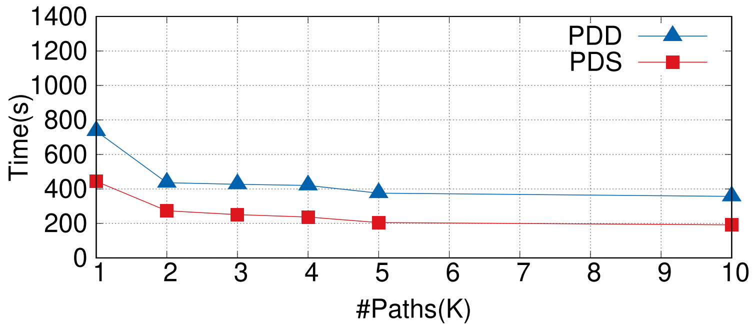
(c) $K = 10$

Figure 7.6: The effects of increasing ρ on PDD and PDS for a given K in EAR.

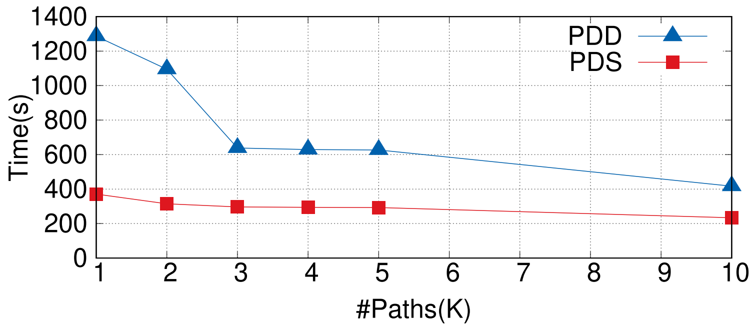
packet replication. However, as we mentioned before, increasing the level of packet replication increases energy consumption as well. Therefore, an upper bound should be adopted to guarantee the packet transmission efficiency. From the changing trend of PDD in Figure 7.7, we observe that the benefit of increasing K, in terms of decreasing PDD, becomes smaller as K becomes larger. This indicates the rationality of setting an upper bound for the level of packet replication. In the following part, when we compare EAR (Mean-Risk model) with other DTN protocols, we set $K = 3$ as an upper bound. The other fact we observe in Figure 7.7 is that, increasing K does



(a) $\rho = 0$



(b) $\rho = 1$



(c) $\rho = 10$

Figure 7.7: The effects of increasing K on PDD and PDS for a given ρ in EAR.

not always help decreasing PDS, especially when the value of ρ is large. This matches what we mentioned in Result 3, K only has an incidental effect on PDS.

EAR (Mean-Risk) vs. other DTN protocols. We compare EAR (Mean-Risk) with four other DTN protocols (Prophet, MaxProp, Rapid and SW3) in terms of PDD, PDS and TTE. Three parameter settings for EAR are utilized, i.e., MRk1r0, MRk1r10 and MRk3r0, which represent ($K = 1$, $\rho = 0$), ($K = 1$, $\rho = 10$) and ($K = 3$, $\rho = 0$), respectively. MRk1r10 and MRk1r0 represent two extreme cases in which the lowest PDS and PDD are preferred, respectively. MRk3r0 is

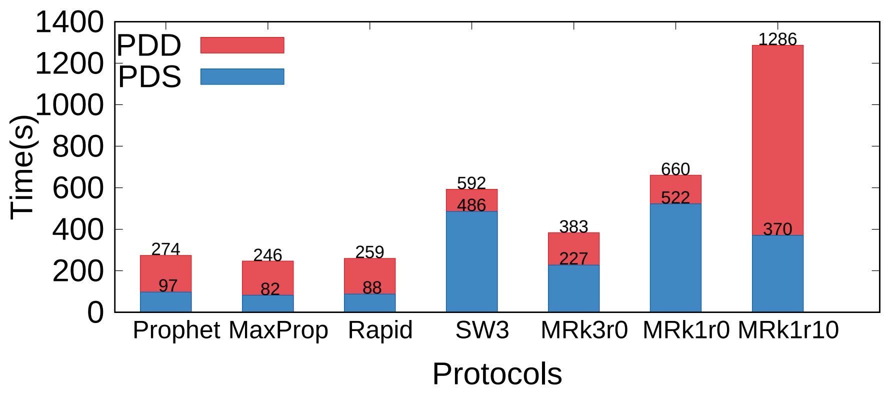


Figure 7.8: PDD and PDS comparison between EAR (Mean-Risk model) and other routing protocols for DRNs.

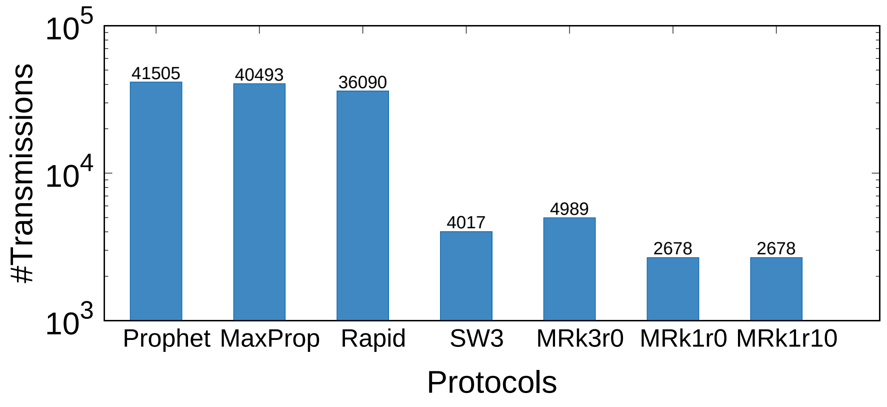
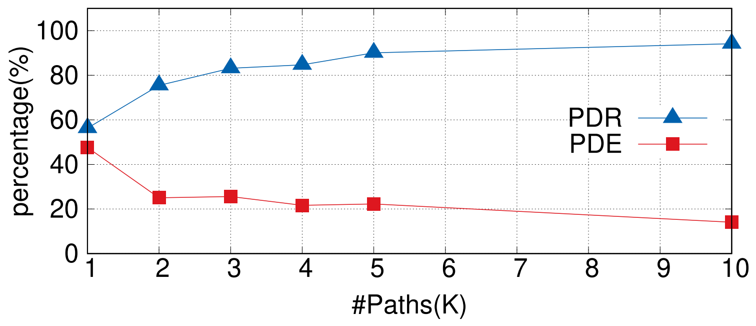
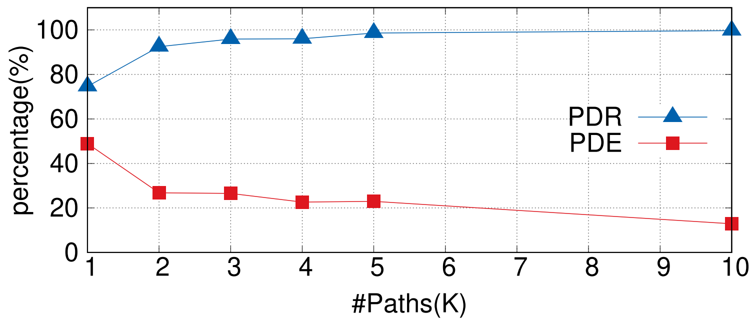


Figure 7.9: TTE (in a log scale) comparison between EAR (Mean-Risk model) and other routing protocols for DRNs.

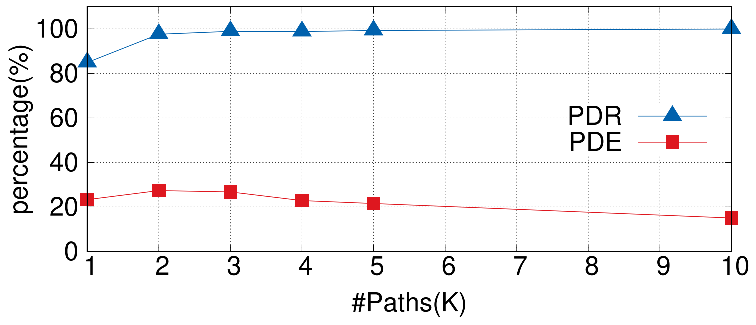
chosen to compare with SW3 in a fair way. From Figure 7.8 and Figure 7.9, we find that MRk3r0 achieves 42% lower PDD and 57% lower PDS than MRk1r0, and 70% lower PDD and 39% lower PDS than MRk1r10 by using two more replicas per packet (which increases packet transmissions by just 86%). This means EAR allows trade-off among PDD, PDS and TTE. Besides, we find that existing DTN routing protocols Prophet, MaxProp and Rapid achieve much lower PDD and PDS than EAR, because they use up to 9x packet transmissions, and hence 9x TTE. Taking Rapid as an example, MRk3r0 consumes 6x less energy by increasing PDD only by 48% and PDS only by 158%. This trade-off is beneficial in DRNs, because the batteries of the fixed routers have a limited capacity. Quite different trade-offs exist between MRk3r0 and SW3. MRk3r0 achieves 35% lower PDD and 53% lower PDS than SW3 with only 24% more packet transmissions. Therefore, MRk3r0 is a good compromise when taking PDD, PDS and TTE all into account.



(a) $D = 10min$

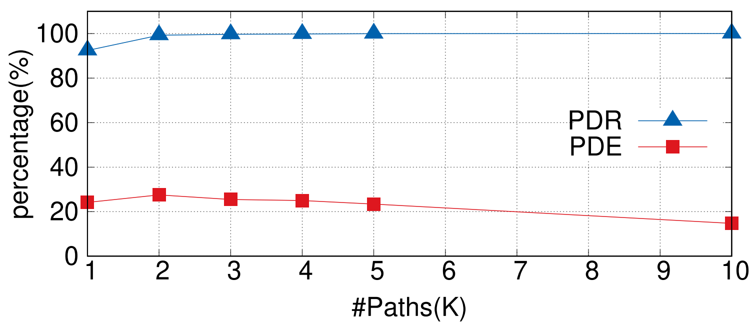


(b) $D = 15min$

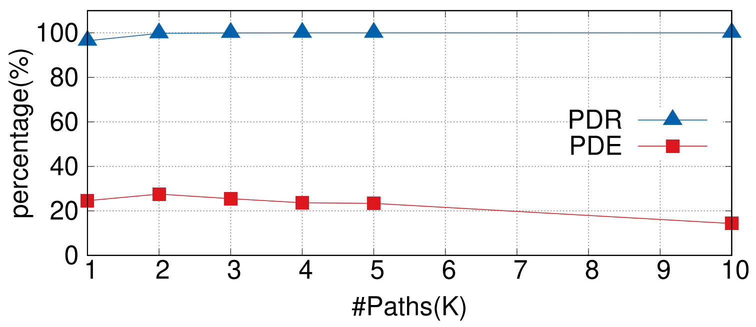


(c) $D = 20min$

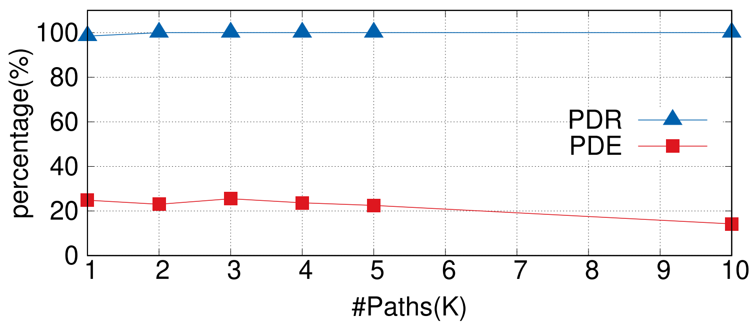
Figure 7.10: The effects of increasing K on PDR and PDE for D = 10, 15, 20min in EAR.



(a) $D = 25min$



(b) $D = 30min$



(c) $D = 35min$

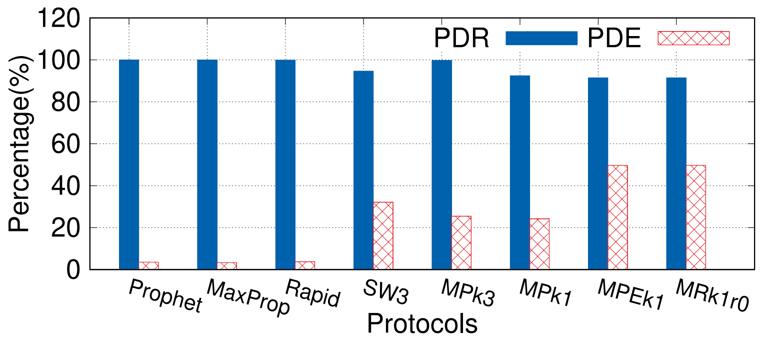
Figure 7.11: The effects of increasing K on PDR and PDE for D = 25, 30, 35min in EAR.

7.4.4 Evaluation of Max-Probability Model

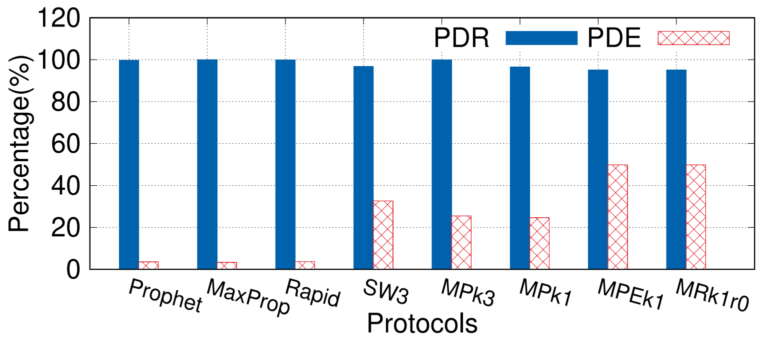
For a given D, increasing K. As shown in Figure 7.10 and Figure 7.11, as K increases, PDR always increases. This indicates that increasing packet replicas is an effective way to increase the probability of delivering packets to the destination in time, as long as the energy condition allows. Here, "in time" means the packet delivery delay is less than a user allowed maximum delay, which is also named as deadline D. From the PDR results shown in Figure 7.10, we find that when the deadline is short ($D = 10min/15min/20min$), increasing K from 1 to 3 increases

PDR much faster than increasing K from 3 to 10. This indicates that the benefit of increasing packet replicas becomes smaller as K increases. Therefore, we can set an upper bound for the number of packet replicas to improve the packet transmission efficiency. However, from Figure 7.11, we observe that when the deadline is long ($D = 25min/30min/35min$), even with $K = 1$, the PDR value is higher than 93%. In such case, we can send a single packet to save energy while maintaining PDR at an adequate level. Since there are plenty of packets in DRNs that do not have an urgent delivery requirement, this "deadline-based" replication is very useful for saving energy. Therefore, in the following, when we compare EAR with other DTN protocols, we set $K = 3$ for $D = 10min/15min/20min$ and $K = 1$ for $D = 25min/30min/35min$.

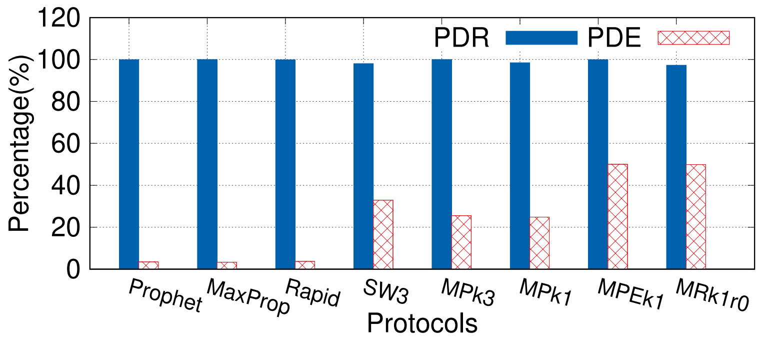
In addition to PDR, we also show how PDE is affected by the increase of K. PDE measures the ratio of total number of distinct delivered packets over the total number of packet transmissions. Figure 7.10 and Figure 7.11 show that, in general, PDE decreases as K increases. This is because, although increasing K enables more distinct packets to be delivered to the destination in time, the packet transmissions increase even faster. In order to eliminate useless packet transmission, we set up a time-to-live (TTL) value for each packet. The TTL value keeps decreasing as time goes and indicates a packet becoming expired as its value decreases to 0. The replicated packets will be discarded as soon as they become expired. Therefore, the increase of packet transmissions is not linear to the increase of K (but much slower). That is the reason PDE does not drop dramatically when K increases. Another observation from Figure 7.11 is that in the case $D = 20min/25min/30min$, when K increases from 1 to 2, PDE even increases. The reason behind is that when $K = 1$, EAR chooses a path which has 4 hops (but with the highest PDR to deliver the packets before D), while when $K = 2$, the newly added path only has 2 hops. In such case, the growth rate of distinct delivered packets might be faster than that of packet transmissions, which makes PDE increasing. This leads to another potential trade-off: *shall we choose a path with higher PDR but more hops or a path with lower PDR but fewer hops*? To answer this question, we extend the Max-Probability model to the Max-Probability Efficient model for $K = 1$ case and compare its performance with other protocols in the following section.



(a) D=25min



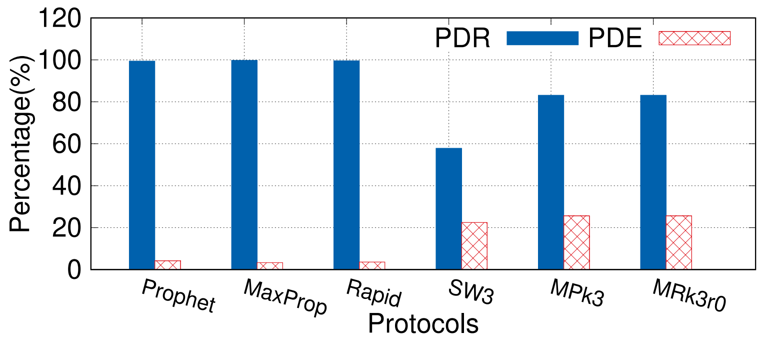
(b) D=30min



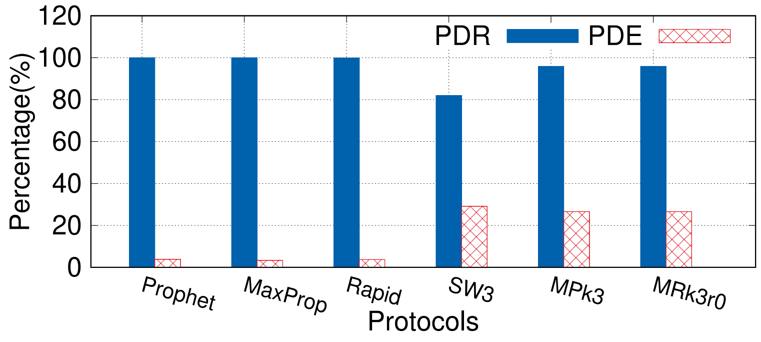
(c) D=35min

Figure 7.12: EAR (Max-Probability) vs. other protocols for D = 25, 30, 35min.

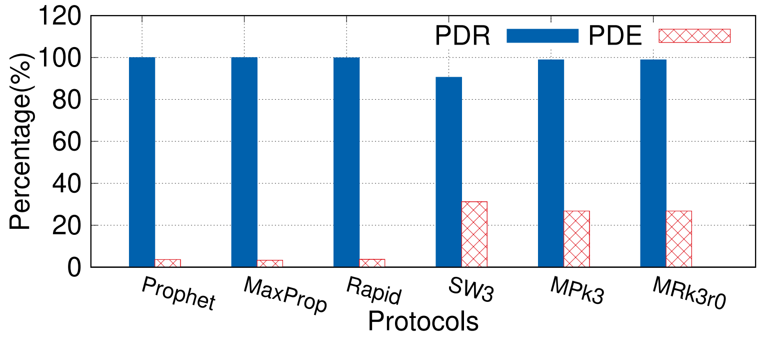
EAR (Max-Probability) vs. other DTN protocols. We compare EAR (Max-Probability) with four other DTN protocols in terms of PDR, PDE and TTE. For the long deadline case ($D = 25min/30min/35min$), we adopt two typical parameter settings MPk1 and MPk3, which represent $K = 1$ and $K = 3$ in the Max-Probability model, respectively. Besides, we also add MRk1r0 ($K = 1$, $\rho = 0$ in the Mean-Risk model) into comparison to show the difference between the



(a) D=10min



(b) D=15min



(c) D=20min

Figure 7.13: EAR (Max-Probability) vs. other protocols for D = 10, 15, 20min.

Mean-Risk and Max-Probability models. Moreover, we compare MPk1 with its extension MPEk1 (Max-Probability Efficient model with $K = 1$) by replacing φ_p with φ_p/H (H is the number of hops of a path). As shown in Figure 7.12, Prophet, MaxProp and Rapid achieve a little bit higher PDR than EAR. However, the cost is much higher. For example, MaxProp improves PDR of MPk3 only by 0.3%, 0.05%, 0% for $D = 25min/30min/35min$, respectively, at the cost of 6.5x more packet transmissions (Figure 7.14). Therefore, when the deadline is long and PDR is already very

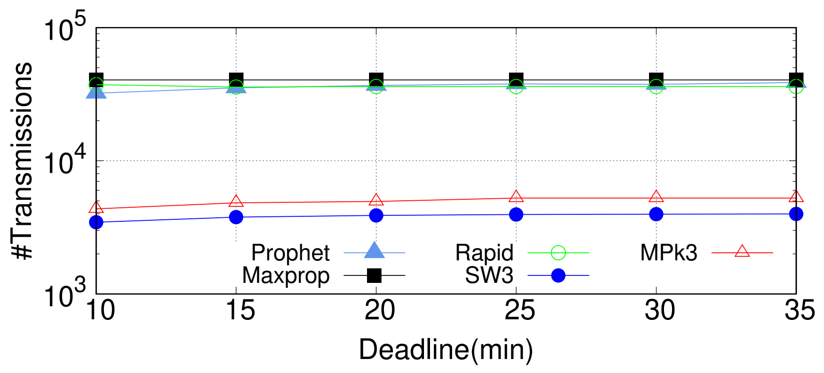


Figure 7.14: TTE (in a log scale) comparison between EAR (Mean-Probability model) and other routing protocols for DRNs.

high with just a single packet, the gain of increasing the level of packet replication is very limited. That is the reason that PDE of Prophet, MaxProp and Rapid is much lower than that of EAR in Figure 7.12. Another interesting observation is that MPk1 achieves a slightly higher PDR but over 2x lower PDE than MRk1r0. The reason behind is, MPk1 chooses a path with higher probability but more hops than MRk1, which leads to more packet transmissions. This result reveals that the path with the lowest PDD is not always the path that maximizes PDR. That's why we need both Mean-Risk and Max-Probability models in EAR. From Figure 7.12, we observe that with MPEk1, EAR increases PDE by choosing the path with fewer hops. The side effect is that PDR of the path chosen by MPEk1 might be a bit lower than that chosen by MPk1. Therefore, EAR should choose MPEk1/MPk1 based on whether PDE/PDR is more important in a specific scenario. When we compare SW3 and MPk3, we find that the advantages of MPk3 seems to be not obvious in the long deadline case, because although MPk3 achieves a bit higher PDR than SW3, its PDE is lower. However, when we compare them in the short deadline case, the advantage of MPk3 becomes clear.

Next, we compare EAR (Max-Probability) and other protocols in terms of PDR, PDE and TTE. For the short deadline case ($D = 10min/15min/20min$), in order to achieve high PDR, EAR just uses MPk3. For completeness, we also add MRk3r0 ($K = 3$, $\rho = 0$ in the Mean-Risk model) for reference. As shown in Figure 7.13, Prophet, MaxProp and Rapid achieve much higher PDR than

EAR by using more packet transmissions (Figure 7.14). From Figure 7.13 (a), we find that although MPk3 achieves 16% lower PDR than Prophet, MaxProp and Rapid when $D = 10min$, its PDE is 6x-8x higher. When D increases to $15min$ and $20min$, MPk3 achieves a PDR only 4% lower than Prophet, MaxProp and Rapid, but 6x-8x higher PDE. This means that EAR is more energy-efficient than Prophet, MaxProp and Rapid. In addition, we find that MPk3 outperforms SW3 by achieving 25% higher PDR but comparable PDE when $D = 10min$. The reason behind is that the mobility pattern information used by EAR becomes critical in the short deadline case. However, when $D = 15min/20min$, although MPk3 still achieves higher PDR than SW3, PDE of the SWs is higher. This is because, paths with more hops are used by MPk3 when $D = 15min/20min$, which increases packet transmissions and decreases PDE.

8. CONCLUSIONS AND FUTURE WORK

In this section, we conclude this dissertation and present the future work.

8.1 Conclusions

To reduce the impact of natural disasters, more and more new techniques are applied to assist disaster response. As a key indicator of effective disaster response, efficient gathering and processing of data from different disaster sites of the disaster area is very important to help the rescue dispatchers at the Emergency Operation Center (EOC) to gain an overview of the whole disaster area to make the right decision. In this dissertation, we present an adaptive edge computing and communication framework for disaster response, which helps processing a large amount of sensing data at each disaster site and sending the most important processing results back to EOC through disaster response networks.

To enable executing computation intensive stream processing tasks on the resource-constrained mobile devices, we design and implement a distributed mobile stream processing platform that enables one mobile device to offload some computation tasks to its nearby mobile devices to perform distributed mobile stream processing together. And in order to deal with the dynamic computing resources at mobile devices and the dynamic wireless networks connecting them, we propose two important modules inside MStorm, namely F-MStorm and R-MStorm. F-MStorm adopts the feedback-based approach in the configuration, scheduling and execution levels of system design and R-MStorm implements resilient mobile stream processing by assigning tasks to the most available mobile devices and assigning tasks of the same application component to different devices to increase the diversity of physical stream paths. We implement both F-MStorm and R-MStorm on Android phones and demonstrate their effectiveness through some test applications.

To extract different information of interests from the video stream taken by the first responders' on-body camera, we design and implement an adaptive execution framework for CNN-based multitask video processing called AMVP. AMVP enables multiple CNNs to share some common frozen layers to reduce the total computation workload and automatically divides each CNN into separated parts so that they can be scheduled to run on the MStorm platform based on the specific application performance goals and actual available computing resources. We implement AMVP on Android phones and show that it can adapt to different application performance goals, computing resources and network conditions.

To transmitting the processing results at each disaster site back to EOC, we propose an energy-aware risk-averse routing protocol for DRNs called EAR. EAR applies the risk-aversion model to address PDS and the max-probability model to deliver the packets to EOC before a deadline. It also applies a differentiated service model to deliver packets with less TTE while maintaining other metrics at an adequate level. We evaluate EAR through extensive simulations and show that, EAR provides flexible control of the routing risks and delivers packets to the destinations in a more energy-efficient way than some well-known DRN routing protocols.

8.2 Future Work

In this section, we present a few ideas for future work.

8.2.1 Distributed Stream Processing on Heterogeneous Devices

In current MStorm, all mobile devices are assumed to be homogeneous. However, in practice, except for the mobile phones carried by the first responders, there are also many other mobile devices at disaster sites such as UAVs, robots, wireless routers, which can also be used to perform distributed mobile stream processing. These mobile devices have totally different hardware/software architectures with Android phones, which makes the current MStorm fail to work directly on them. In the future, we plan to implement MStorm on some other common systems different from Android phones, so that it can use as much resources as possible at the disaster site.

8.2.2 In-order Mobile Stream Processing

In current MStorm, we only consider applications that can tolerate partial data missing and out-of-order processing. However, in real word, there are a lot of applications which require 100% in-order processing. To enable this feature in MStorm, we consider to use an end-to-end acknowledgement and some retransmission mechanisms in the future to ensure in-order processing and application integrity.

8.2.3 Reduce Cost of Task Rescheduling

In current MStorm, when task rescheduling happens, the executor will first stop and kill all the old tasks and then runs up the new one. Although this method is simple and convenient, it has some drawbacks: 1) the switching time between two schedules is very long; 2) some stream packets might get lost during the switching; 3) some unnecessary cost is incurred. In the future, instead of stopping and killing all the old tasks before running the new ones, we first check if there is some "common parts" between two schedules. If there are, those common part tasks will not be killed but just hung up. Then, when the actual new tasks run up, those hung up tasks will be notified to recover to run.

8.2.4 Energy Efficiency in Mobile Stream Processing

In current MStorm, we mainly focus on dealing with dynamic computing resources at mobile devices and dynamic wireless networks connecting them. Although the communication energy cost is considered in F-MStorm, how to reduce the computation energy cost of MSP is still not considered yet. In the future, we will delve into the detailed implementation of MStorm to figure out its energy bottleneck. We will replace the bottleneck code with more energy efficient one to prolong the battery life time of first responders' mobile phones.

8.2.5 More Accurate Model to Describe the Device Availability

In current R-MStorm, we assume that the availability of each mobile device can be obtained by analyzing the RSSI trace during a rescue operation. However, the availability of mobile devices is not only decided by the network condition but also by some other factors such as battery level or system crash. In the future, we plan to introduce some more accurate models to describe the availability of each mobile device.

8.2.6 Supporting different DNNs in AMVP

In current AMVP, we only support image classification tasks. In the future, we plan to support some other vision processing tasks such as objection detection, event detection, instance segmentation, etc. Moreover, we also plan to support some speech and voice recognition tasks to better assist first responders to conduct their rescue work.

REFERENCES

[1] "Effective disaster management strategies in the 21st century." http://www.govtech. com/ em/disaster/Effective-Disaster-Management-Strategies. html, 2010. Accessed: 03-may-2017.

[2] "Effects of hurricane maria in puerto rico." https://en.wikipedia.org/wiki/ Effects_of_Hurricane_Maria_in_Puerto_Rico, 2019. Accessed: 01-May-2020.

[3] E. M. Trono, M. Fujimoto, H. Suwa, Y. Arakawa, and K. Yasumoto, "Milk carton: Family tracing and reunification system using face recognition over a dtn with deployed comput-ing nodes," in *Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services*, pp. 24–28, 2016.

[4] S. M. Preum, S. Shu, J. Ting, V. Lin, R. Williams, J. Stankovic, and H. Alemzadeh, "Towards a cognitive assistant system for emergency response," in *Proceedings of 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 347–348, 2018.

[5] D. Zhang, Y. Zhang, Q. Li, T. Plummer, and D. Wang, "Crowdlearn: A crowd-ai hybrid system for deep learning-based damage assessment applications," in *Proceedings of 39th In-ternational Conference on Distributed Computing Systems (ICDCS)*, pp. 1221–1232, 2019.

[6] H. Chenji, W. Zhang, R. Stoleru, and C. Arnett, "Distressnet: A disaster response sys-tem providing constant availability cloud-like services," *Ad Hoc Networks*, vol. 11, no. 8, pp. 2440–2460, 2013.

[7] "Hurricane maria communications status report for sept. 29." https://www.fcc.gov/ document/hurricane-maria-communications-status-report-sept-29, 2017. Accessed: 01-May-2020.

[8] Q. Ning, C. Chen, R. Stoleru, and C. Chen, “Mobile storm: Distributed real-time stream processing for mobile clouds,” in *Proceedings of IEEE International Conference on Cloud Networking*, pp. 139–145, 2015.

[9] M. Satyanarayanan, G. Lewis, E. Morris, S. Simanta, J. Boleng, and K. Ha, “The role of cloudlets in hostile environments,” *IEEE Pervasive Computing*, vol. 12, no. 4, pp. 40–49, 2013.

[10] M. Aazam and E.-N. Huh, “E-hamc: Leveraging fog computing for emergency alert service,” in *Proceedings of 2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pp. 518–523, 2015.

[11] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, “Odessa: enabling interactive perception applications on mobile devices,” in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp. 43–56, 2011.

[12] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *Proceedings of 2012 IEEE Infocom*, pp. 945–953, 2012.

[13] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, “Cosmos: computation offloading as a service for mobile devices,” in *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, pp. 287–296, 2014.

[14] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, “Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 123–136, 2016.

[15] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[16] J. Zhao, Q. Li, Y. Gong, and K. Zhang, "Computation offloading and resource allocation for cloud assisted mobile edge computing in vehicular networks," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 8, pp. 7944–7956, 2019.

[17] M. Huang, W. Liu, T. Wang, A. Liu, and S. Zhang, "A cloud-mec collaborative task offloading scheme with service orchestration," *IEEE Internet of Things Journal*, 2019.

[18] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, pp. 145–154, 2012.

[19] K. Toda, K. Yamato, T. Kodachi, M. Shimizu, T. Nishimura, T. Yoshida, and T. Fruta, "Camera arm system for disaster response robots," in *Proceedings of International Conference on Design Engineering and Science*, pp. 80–85, 2014.

[20] B. Song, A. T. Kamal, C. Soto, C. Ding, J. A. Farrell, and A. K. Roy-Chowdhury, "Tracking and activity recognition through consensus in distributed camera networks," *IEEE Transactions on Image Processing*, vol. 19, no. 10, pp. 2564–2579, 2010.

[21] B. Song, C. Ding, A. T. Kamal, J. A. Farrell, and A. K. Roy-Chowdhury, "Distributed camera networks," *IEEE Signal Processing Magazine*, vol. 28, no. 3, pp. 20–31, 2011.

[22] C. Ding, B. Song, A. Morye, J. A. Farrell, and A. K. Roy-Chowdhury, "Collaborative sensing in a distributed ptz camera network," *IEEE Transactions on Image Processing*, vol. 21, no. 7, pp. 3282–3295, 2012.

[23] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger, "Mainstream: Dynamic stem-sharing for multi-tenant video processing," in *Proceedings of 2018 USENIX Annual Technical Conference*, pp. 29–42, 2018.

[24] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi, "Videochef: efficient approximation for streaming video processing pipelines," in *Proceedings of 2018 USENIX Annual Technical Conference*, pp. 43–56, 2018.

[25] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, G. Maghanath, and S. Bagchi, "Approxnet: Content and contention aware video analytics system for the edge," *arXiv preprint arXiv:1909.02068*, 2019.

[26] K.-J. Hsu, K. Bhardwaj, and A. Gavrilovska, "Couper: Dnn model slicing for visual analytics containers at the edge," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pp. 179–194, 2019.

[27] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *Proceedings of 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1396–1401, 2017.

[28] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *Proceedings of the 36th International Conference on Computer-Aided Design*, pp. 751–756, 2017.

[29] Z. Xu, Z. Qin, F. Yu, C. Liu, and X. Chen, "Direct: Resource-aware dynamic model reconfiguration for convolutional neural network in mobile systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, p. 37, 2018.

[30] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[31] R. Hadidi, J. Cao, M. Woodward, M. Ryoo, and H. Kim, "Musical chair: Efficient real-time recognition using collaborative iot devices," *arXiv preprint arXiv:1802.02138*, 2018.

[32] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[33] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized neural networks: Training neural networks with low precision weights and activations.,” *Journal of Machine Learning Research*, vol. 18, pp. 187–1, 2017.

[34] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.

[35] M. Wang, B. Liu, and H. Foroosh, “Factorized convolutional neural networks,” in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pp. 545–553, 2017.

[36] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.

[37] S. Changpinyo, M. Sandler, and A. Zhmoginov, “The power of sparsity in convolutional neural networks,” *arXiv preprint arXiv:1702.06257*, 2017.

[38] B. Fang, X. Zeng, and M. Zhang, “Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision,” in *Proceedings of Annual International Conference on Mobile Computing and Networking (MobiCom)*, pp. 115–127, 2018.

[39] Y. Nakayama, K. Maruta, T. Tsutsumi, R. Yasunaga, K. Honda, and K. Sezaki, “Recovery node layout planning for wired and wireless network cooperation for disaster response,” in *Proceedings of the IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, 2018.

[40] J. Z. Moghaddam, M. Usman, and F. Granelli, “A device-to-device communication-based disaster response network,” *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 2, pp. 288–298, 2018.

[41] A. Vahdat and D. Becker, “Epidemic routing for partially connected ad hoc networks,” Tech. Report CS-200006, Duke University, 2000.

[42] A. Lindgren, A. Doria, and O. Schelen, “Probabilistic routing in intermittently connected networks,” in *Proceedings of International Workshop on Service Assurance with Partial and Intermittent Resources*, pp. 239–254, 2004.

[43] J. Burgess, B. Gallagher, D. Jensen, and B. N. Levine, “Maxprop: Routing for vehicle-based disruption-tolerant networks,” in *Proceedings of the 25th International Conference on Computer Communications (Infocom)*, pp. 1–11, 2006.

[44] A. Balasubramanian, B. Levine, and A. Venkataramani, “Dtn routing as a resource allocation problem,” *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 373–384, 2007.

[45] H. Chenji, L. Smith, R. Stoleru, and E. Nikolova, “Raven: Energy aware qos control for drns,” in *Proceedings of the 9th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 464–471, 2013.

[46] A. Keränen, J. Ott, and T. Kärkkäinen, “The one simulator for dtn protocol evaluation,” in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, p. 55, 2009.

[47] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62, 2010.

[48] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the sixth conference on Computer systems*, pp. 301–314, 2011.

[49] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, “Just-in-time provisioning for cyber foraging,” in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pp. 153–166, 2013.

[50] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, “Dsp. ear: Leveraging co-processor support for continuous audio sensing on smartphones,” in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, pp. 295–309, 2014.

[51] M.-M. Moazzami, D. E. Phillips, R. Tan, and G. Xing, “Orbit: a smartphone-based platform for data-intensive embedded sensing applications,” in *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pp. 83–94, 2015.

[52] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, “Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources,” in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pp. 320–333, 2016.

[53] E. E. Marinelli, “Hyrax: cloud computing on mobile devices using mapreduce,” tech. rep., CMU, 2009.

[54] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, “You can teach elephants to dance: agile vm handoff for edge computing,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pp. 1–14, 2017.

[55] L. Chaufournier, P. Sharma, F. Le, E. Nahum, P. Shenoy, and D. Towsley, “Fast transparent virtual machine migration in distributed edge clouds,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pp. 1–13, 2017.

[56] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, *et al.*, “An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pp. 1–14, 2017.

[57] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, “Edge-centric computing: Vision and challenges,” *SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.

[58] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156, 2014.

[59] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pp. 207–218, 2013.

[60] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proceedings of 2014 IEEE 34th International Conference on Distributed Computing Systems*, pp. 535–544, 2014.

[61] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of Annual Middleware Conference*, pp. 149–161, 2015.

[62] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 1579–1588, 2014.

[63] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: dynamic resource scheduling for real-time analytics over fast streams," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, pp. 411–420, 2015.

[64] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 69–80, 2016.

[65] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters.," in *Proceedings of 4th USENIX Workshop on Hot Topics in Cloud Computing*, vol. 12, pp. 10–10.

[66] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[67] "Apache Samza." http://samza.apache.org/, 2017. Accessed: 14-Dec-2017.

[68] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: a better stream processing engine for the edge," in *Proceedings of USENIX Annual Technical Conference*, pp. 929–946, 2019.

[69] H. Wang and L.-S. Peh, "Mobistreams: A reliable distributed stream processing system for mobile devices," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, pp. 51–60, 2014.

[70] S. Fan, T. Salonidis, and B. Lee, "Swing: Swarm computing for mobile sensing," in *Proceedings of IEEE International Conference on Distributed Computing Systems*, pp. 1107–1117, 2018.

[71] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: resilient edge processing for the internet of things," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1178–1191, 2018.

[72] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[73] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Proceedings of European Conference on Computer Vision*, pp. 525–542, 2016.

[74] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

[75] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 389–400, 2018.

[76] Z. Chen, W. Lin, S. Wang, L. Duan, and A. C. Kot, “Intermediate deep feature compression: the next battlefield of intelligent sensing,” *arXiv preprint arXiv:1809.06196*, 2018.

[77] H. Choi and I. V. Bajić, “Deep feature compression for collaborative object detection,” in *Proceedings of the 25th IEEE International Conference on Image Processing (ICIP)*, pp. 3743–3747, 2018.

[78] H. Choi and I. V. Bajić, “Near-lossless deep feature compression for collaborative intelligence,” in *Proceedings of the 20th IEEE International Workshop on Multimedia Signal Processing (MMSP)*, pp. 1–6, 2018.

[79] Z. Chen, K. Fan, S. Wang, L.-Y. Duan, W. Lin, and A. Kot, “Lossy intermediate deep learning feature compression and evaluation,” in *Proceedings of the 27th ACM International Conference on Multimedia*, pp. 2414–2422, 2019.

[80] Y. Cao and Z. Sun, “Routing in delay/disruption tolerant networks: A taxonomy, survey and challenges,” *IEEE Communications surveys & tutorials*, vol. 15, no. 2, pp. 654–677, 2013.

[81] T. Matsuda and T. Takine, “(p, q)-epidemic routing for sparsely populated mobile ad hoc networks,” *IEEE Journal on Selected Areas in Communications*, vol. 26, no. 5, pp. 783–793, 2008.

[82] M. Y. S. Uddin, H. Ahmadi, T. Abdelzaher, and R. Kravets, “Intercontact routing for energy constrained disaster response networks,” *IEEE transactions on mobile computing*, vol. 12, no. 10, pp. 1986–1998, 2013.

[83] Y. Zhu, B. Xu, X. Shi, and Y. Wang, “A survey of social-based routing in delay tolerant networks: Positive and negative social effects,” *IEEE Communications Surveys & Tutorials*, vol. 15, no. 1, pp. 387–401, 2013.

[84] H. Zhou, V. C. Leung, C. Zhu, S. Xu, and J. Fan, “Predicting temporal social contact patterns for data forwarding in opportunistic mobile networks,” *IEEE Transactions on Vehicular Technology*, vol. 66, no. 11, pp. 10372–10383, 2017.

[85] K. Liu, Z. Chen, J. Wu, Y. Xiao, and H. Zhang, "Predict and forward: An efficient routing-delivery scheme based on node profile in opportunistic networks," *Future Internet*, vol. 10, no. 8, p. 74, 2018.

[86] M. Radenkovic, V. S. H. Huynh, and P. Manzoni, "Adaptive real-time predictive collaborative content discovery and retrieval in mobile disconnection prone networks," *IEEE Access*, vol. 6, pp. 32188–32206, 2018.

[87] Y. Yan, Z. Chen, J. Wu, L. Wang, K. Liu, and P. Zheng, "An effective transmission strategy exploiting node preference and social relations in opportunistic social networks," *IEEE Access*, vol. 7, pp. 58186–58199, 2019.

[88] P. Hui, J. Crowcroft, and E. Yoneki, "Bubble rap: Social-based forwarding in delay-tolerant networks," *IEEE Transactions on Mobile Computing*, vol. 10, no. 11, pp. 1576–1589, 2011.

[89] X. Wang, Y. Lin, and S. Zhang, "A social activity and physical contact-based routing algorithm in mobile opportunistic networks for emergency response to sudden disasters," *Enterprise Information Systems*, vol. 11, no. 5, pp. 597–626, 2017.

[90] K. Chen and H. Shen, "Smart: Lightweight distributed social map based routing in delay tolerant networks," in *Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP)*, pp. 1–10, 2012.

[91] V. K. Shah, S. Roy, S. Silvestri, and S. K. Das, "Ctr: Cluster based topological routing for disaster response networks," in *Proceedings of the IEEE International Conference on Communications (ICC)*, pp. 1–6, 2017.

[92] T. Spyropoulos, K. Psounis, and C. S. Raghavendra, "Efficient routing in intermittently connected mobile networks: The multiple-copy case," *IEEE/ACM Transactions on Networking (ToN)*, vol. 16, no. 1, pp. 77–90, 2008.

[93] M. Khouzani, S. Eshghi, S. Sarkar, N. B. Shroff, and S. S. Venkatesh, "Optimal energy-aware epidemic routing in dtns," in *Proceedings of the 13th ACM international symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pp. 175–182, 2012.

[94] D. M. Chen, S. S. Tsai, R. Vedantham, R. Grzeszczuk, and B. Girod, “Streaming mobile augmented reality on mobile phones,” in *Proceeding of the 2009 8th IEEE International Symposium on Mixed and Augmented Reality*, pp. 181–182, 2009.

[95] H. Lu, D. Frauendorfer, M. Rabbi, M. S. Mast, G. T. Chittaranjan, A. T. Campbell, D. Gatica-Perez, and T. Choudhury, “Stresssense: Detecting stress in unconstrained acoustic environments using smartphones,” in *Proceedings of the 2012 ACM conference on ubiquitous computing*, pp. 351–360, 2012.

[96] Y. Lee, C. Min, C. Hwang, J. Lee, I. Hwang, Y. Ju, C. Yoo, M. Moon, U. Lee, and J. Song, “Sociophone: Everyday face-to-face interaction monitoring platform using multi-phone sensor fusion,” in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pp. 375–388, 2013.

[97] G. Chen, C. Parada, and G. Heigold, “Small-footprint keyword spotting using deep neural networks,” in *Proceedings of the 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4087–4091, 2014.

[98] N. D. Lane, P. Georgiev, and L. Qendro, “Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning,” in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pp. 283–294, 2015.

[99] I. Damian, C. S. Tan, T. Baur, J. Schöning, K. Luyten, and E. André, “Augmenting social interactions: Realtime behavioural feedback using social signal processing techniques,” in *Proceedings of the 33rd annual ACM conference on Human factors in computing systems*, pp. 565–574, 2015.

[100] J. Tang and T. Q. Quek, “The role of cloud computing in content-centric mobile networking,” *IEEE Communications Magazine*, vol. 54, no. 8, pp. 52–59, 2016.

[101] J. Dean and S. Ghemawat, “Mapreduce: a flexible data processing tool,” *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[102] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, “Drizzle: Fast and adaptable stream processing at scale,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 374–389, 2017.

[103] L. Sun, R. K. Sheshadri, W. Zheng, and D. Koutsonikolas, “Modeling wifi active power/energy consumption in smartphones,” in *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, pp. 41–51, 2014.

[104] “Listen to current network traffic in the app and categorize the quality of the network.” https://github.com/facebook/network-connection-class, 2016. Ac-cessed: 14-Dec-2016.

[105] “API: CPLEX.” https://www.ibm.com/, 2016. Accessed: 14-Dec-2016.

[106] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[107] G. P. Srinivasa, R. Begum, S. Haseley, M. Hempstead, and G. Challen, “Separated by birth: Hidden differences between seemingly-identical smartphone cpus,” in *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, pp. 103–108, 2017.

[108] M. Chao, C. Yang, Y. Zeng, and R. Stoleru, “F-mstorm: Feedback-based online distributed mobile stream processing,” in *Proceedings of IEEE/ACM Symposium on Edge Computing*, pp. 273–285, 2018.

[109] T. Clausen, P. Jacquet, C. Adjih, A. Laouiti, P. Minet, P. Muhlethaler, A. Qayyum, and L. Viennot, “Optimized Link State Routing Protocol (OLSR),” 2003. Network Working Group.

[110] K. P. Yoon and C.-L. Hwang, *Multiple attribute decision making: an introduction*, vol. 104. Sage publications, 1995.

[111] T. Berthold, S. Heinz, and M. E. Pfetsch, "Nonlinear pseudo-boolean optimization: relaxation or propagation?," in *Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, pp. 441–446, 2009.

[112] D. Le Berre and A. Parrain, "The sat4j library, release 2.2, system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010.

[113] K. Muhammad, J. Ahmad, Z. Lv, P. Bellavista, P. Yang, and S. W. Baik, "Efficient deep cnn-based fire detection and localization in video surveillance applications," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 7, pp. 1419–1434, 2018.

[114] M. Ravanbakhsh, M. Nabi, H. Mousavi, E. Sangineto, and N. Sebe, "Plug-and-play cnn for crowd motion analysis: An application in abnormal event detection," in *Proceedings of the 2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1689–1698, 2018.

[115] M. Babaee, D. T. Dinh, and G. Rigoll, "A deep convolutional neural network for video sequence background subtraction," *Pattern Recognition*, vol. 76, pp. 635–649, 2018.

[116] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors," in *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, pp. 1–16, 2019.

[117] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks," *IEEE Signal Processing Letters*, vol. 23, no. 10, pp. 1499–1503, 2016.

[118] R. Ranjan, V. M. Patel, and R. Chellappa, "Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 1, pp. 121–135, 2017.

[119] D. C. Luvizon, D. Picard, and H. Tabia, “2d/3d pose estimation and action recognition using multitask deep learning,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5137–5146, 2018.

[120] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[121] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[122] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, “How transferable are features in deep neural networks?,” in *Advances in neural information processing systems*, pp. 3320–3328, 2014.

[123] S. Luo, Y. Yang, Y. Yin, C. Shen, Y. Zhao, and M. Song, “Deepsic: Deep semantic image compression,” in *Proceedings of the International Conference on Neural Information Processing*, pp. 96–106, 2018.

[124] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, “Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services,” *IEEE Transactions on Mobile Computing*, 2019.

[125] “Keras: The python deep learning library.” https://keras.io/. Accessed: 10-Apr-2020.

[126] “Tensorflow: An end-to-end open source machine learning platform.” https://www.tensorflow.org/. Accessed: 10-Apr-2020.

[127] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *Proceedings of the 2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, 2009.

[128] B. Nowell, T. Steelman, A.-L. K. Velez, and Z. Yang, “The structure of effective governance of disaster response networks: Insights from the field,” *The American Review of Public Administration*, vol. 48, no. 7, pp. 699–715, 2018.

[129] R. P. Loui, “Optimal paths in graphs with stochastic or multidimensional weights,” *Communications of the ACM*, vol. 26, no. 9, pp. 670–676, 1983.

[130] W. Ogryczak and A. Ruszczynskia, “Dual Stochastic Dominance and Quantile Risk Measures,” *International Transactions in Operational Research*, vol. 9, pp. 661–680, 2002.

[131] E. Nikolova, “Approximation algorithms for offline risk-averse combinatorial optimization,” *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pp. 338–351, 2010.

[132] T. Spyropoulos, K. Psounis, and C. S. Raghavendra, “Spray and wait: an efficient routing scheme for intermittently connected mobile networks,” in *Proceedings of ACM SIGCOMM workshop on Delay-tolerant networking*, pp. 252–259, 2005.

[133] S. Lim, H. Balakrishnan, D. Gifford, S. Madden, and D. Rus, “Stochastic motion planning and applications to traffic,” *The International Journal of Robotics Research*, vol. 30, no. 6, pp. 699–712, 2011.

[134] M. Y. S. Uddin, D. M. Nicol, T. F. Abdelzaher, and R. H. Kravets, “A post-disaster mobility model for delay tolerant networking,” in *Proceedings of Winter Simulation Conference*, pp. 2785–2796, 2009.

[135] G. Xie and N. Chen, “A social-aware opportunistic network routing protocol based on the node embeddings,” in *Proceedings of the 89th IEEE Vehicular Technology Conference (VTC)*, pp. 1–5, 2019.

[136] X. Wang, M. Chen, V. C. Leung, Z. Han, and K. Hwang, “Integrating social networks with mobile device-to-device services,” *IEEE Transactions on Services Computing*, 2018.

[137] H. Zhu, L. Fu, G. Xue, Y. Zhu, M. Li, and L. M. Ni, "Recognizing exponential inter-contact time in vanets," in *Proceedings of the International Conference on Computer Communications (Infocom)*, pp. 1–5, 2010.

[138] T. Le, Q. Zhao, and M. Gerla, "Fragmented data routing based on exponentially distributed contacts in delay tolerant networks," in *Proceedings of the International Conference on Computing, Networking and Communications (ICNC)*, pp. 1039–1043, 2019.

[139] G. Ahani and D. Yuan, "On optimal proactive and retention-aware caching with user mobility," in *Proceedings of the 88th IEEE Vehicular Technology Conference (VTC)*, pp. 1–5, 2018.

[140] X. Tie, A. Venkataramani, and A. Balasubramanian, "R3: robust replication routing in wireless networks with diverse connectivity characteristics," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pp. 181–192, 2011.

[141] C. Yang and R. Stoleru, "Hybrid routing in wireless networks with diverse connectivity," in *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pp. 71–80, 2016.

[142] A. Altaweel, R. Stoleru, G. Gu, and A. K. Maity, "Collusivehijack: A new route hijacking attack and countermeasures in opportunistic networks," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, pp. 73–81, 2019.

[143] E. Nikolova, J. A. Kelner, M. Brand, and M. Mitzenmacher, "Stochastic shortest paths via quasi-convex maximization," in *Proceedings of the European Symposium on Algorithms*, pp. 552–563, 2006.

[144] P. J. Carstensen, "The complexity of some problems in parametric linear and combinatorial programming, university of michigan," *PhD dissertation, Michigan University*, 1983.

[145] A. P. Ker, "On the maximum of bivariate normal random variables," *Extremes*, vol. 4, no. 2, pp. 185–190, 2001.