A PROGRAMMING PARADIGM FOR BUILDING DISAGGREGATED APPLICATIONS FOR THE HETEROGENEOUS COMPUTING ENVIRONMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Xinwen Wang

August 2023

© 2023 Xinwen Wang

ALL RIGHTS RESERVED

A PROGRAMMING PARADIGM FOR BUILDING DISAGGREGATED APPLICATIONS FOR THE HETEROGENEOUS COMPUTING ENVIRONMENT

Xinwen Wang, Ph.D.

Cornell University 2023

With the rise of cloud computing, many applications have transitioned to the cloud. However, various domain-specific tasks require specialized hardware to expedite computation, rather than relying solely on CPUs. This has transformed the cloud into a heterogeneous computing environment, composed of a variety of domain-specific specialized accelerators. Examples include GPUs for image classification and video processing, TPUs for artificial intelligence and machine learning tasks, and ASICs for blockchain mining. There are also specialized hardware pieces, such as smartNICs and smartSSDs, which can unlock more of the underlying hardware's potential.

Nonetheless, the existing programming paradigm for applications is CPU-centric, meaning that specialized hardware is bound to a CPU host rather than being a first-class programmable abstraction. This can cause issues such as restricted scalability of accelerators on a single host, increased system complexity when managing multiple nodes, among other problems.

In this dissertation, we propose a programming paradigm that consists of actors and shared logs to provide a *resource-egalitarian* abstraction, simplifying the development of applications within a heterogeneous computing environment. We present two frameworks following this programming paradigm to assist application developers in building applications in both a partitioned network IoT context and a resource-disaggregated cloud context.

BIOGRAPHICAL SKETCH

Xinwen Wang was born and raised in Qingdao, Shandong, China. He pursued his undergraduate studies at Stony Brook University and began his Ph.D. research at Cornell University in 2017. Xinwen has a keen interest in conducting research in the fields of distributed systems and operating systems.

For my parents

For my advisors

For all my friends

ACKNOWLEDGEMENTS

First, I would like to convey my deepest appreciation to my advisor, Robbert van Renesse, for his continuous support and guidance throughout the course of my Ph.D. studies. Robbert has taught me how to conduct research and think as a systems researcher. His ideas on research are always inspiring and exciting. Specifically, I will never forget Robbert's understanding and support during the challenging times of the pandemic, when I was grappling with feelings of depression.

Next, I wish to express my thanks to my other advisors, Hakim Weatherspoon and Andrew Myers. Hakim has always been a source of courage and warm support, while Andrew's insights have consistently inspired me.

I also want to extend my gratitude to my other collaborators and colleagues: Isaac Sheff, Danny Adams, Gloire Rubambiza, Haobin Ni, Kushal Babel, and Pablo Fiori. The Vegvisir project could not have been completed without their efforts.

Lastly, I want to sincerely thank my parents and friends who have supported me during my studies at Cornell.

My research was partially funded through generous gifts from Google and Meta.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vi
List of Tables ix
List of Figures x

1 Introduction 1
1.1 Challenges 1
1.2 Research Statement 3
1.3 Background 4
1.3.1 Heterogenity of hardware 4
1.3.2 Shared Log 4
1.3.3 Blockchains 5
1.3.4 Actor Models 5
1.3.5 Mobile Ad-Hoc Network 6
1.3.6 Resource Disaggregation 7
1.4 Approach 8
1.5 Roadmap 9
1.5.1 Vegvisir 9
1.5.2 BloomBox 9
1.5.3 Uniservice 10
1.6 Contributions 10

2 Tamperproof Provenance-Aware Storage for Mobile Ad Hoc Networks 11
2.1 Motivation 13
2.2 System Design 15
2.2.1 Application Tier 16
2.2.2 Pub/Sub Tier 18
2.2.3 Block Tier 18
2.2.4 Reconciliation Tier 27
2.3 Vegvisir Prototype 30
2.3.1 Application Layer 31
2.3.2 Pub-Sub Layer 34
2.3.3 Block Layer 35
2.3.4 Network Layer 36
2.4 CBDC Application 37
2.4.1 Problem Definition 38
2.4.2 Offline Payment Protocol 39
2.4.3 Correctness 43
2.5 Evaluation 44
2.5.1 Network Measurements 44

2.5.2 Scalability 46
2.6 Related work 48
2.7 Conclusion 51

3 BloomBox: Improving Availability and Efficiency in Geographic Hash Tables 52
3.1 Background and System Model 55
3.1.1 Geographic Routing 55
3.1.2 Geographic Hash Tables 55
3.1.3 Failure Detection and Recovery 56
3.2 System Design 57
3.2.1 Failure Detection 57
3.2.2 Failure Recovery 59
3.2.3 Configuration 60
3.2.4 Mergeable Bloom Filter 62
3.2.5 MBF Size Estimation 63
3.3 Implementation 64
3.3.1 Reconciliation 64
3.3.2 Example of Detection & Recovery 66
3.3.3 Optimizations 67
3.4 Evaluation 67
3.4.1 Simulation Model and Metrics 69
3.4.2 Availability versus Epoch Length 72
3.4.3 Comparing BloomBox and Heartbeat 74
3.4.4 Limitations 75
3.5 Related Work 76
3.5.1 Geographic Hash Tables 76
3.5.2 Mobile Ad Hoc Routing Protocols 78
3.5.3 Bloom Filters 78
3.6 Conclusion 80

4 Disaggregating Applications Using Uniservices 81
4.1 Motivation 84
4.1.1 Disaggregated Hardware Management 85
4.1.2 Programmability 85
4.1.3 Application Performance 86
4.2 Uniservice Design 88
4.2.1 Functions 92
4.2.2 Data Object Reference and Movement 93
4.2.3 Application Execution Graph 94
4.2.4 Shared Log 98
4.2.5 Scheduling 99
4.2.6 Node Failures and Data Recovery 100
4.3 Implementation 102

4.3.1 Shared Log 102
4.3.2 Function Hub 102
4.3.3 Global Scheduler 103
4.3.4 Invoker 103
4.4 Evaluation 104
4.5 Related Work 109
4.6 Conclusion 111

5 Conclusion **112**

Bibliography **114**

LIST OF TABLES

3.1 Parameter settings for simulations 68

4.1 Comparison between microservices and uniservices 88

LIST OF FIGURES

2.1 From (a) to (d), this diagram shows how two disconnected blockdags can be connected. Each block displays a 4 digit unique hash identifier and the owner device in the lower right corner. In the beginning, in figure (a), two blockdags are created by device X and device Y with different genesis blocks 1215 and 3231. For convenience, we use the genesis block number to identify a blockdag and assume that there is only one WCRDT for each blockdag. Now device Y wants device X to be able to append blocks for blockdag 3231. From (b) to (c), device Y creates a new block with a transaction adding device X to blockdag 3231 WCRDT's ACM with write permission. Then, the newly added block 2187 is synchronized with device X through reconciliation. Finally, device X appends a new block 2133 that connects blockdag 3231 and blockdag 1215. 20
2.2 Three valid blockdags. (a) and (b) are two snapshots that show that device z is misbehaving because it has created two blocks (3503 and 3273) that do not depend on one another. As a result, the WCRDTs in the blockdags may have diverged. In order to merge the blockdags, both x and y have to add a PoM for z before they can create blocks that depend on both blocks from z. (c) shows a merged blockdag with additional blocks from x and y added. Neither x nor y has concurrent blocks. z is no longer able to add blocks to the end. 21
2.3 Screenshots of sample applications 31
2.4 Layer structure for Vegvisir 32
2.5 (a) The structure of a Hashed Vector Timestamp. The example shows a HVT from device A which has witnessed blocks from device B up to height 3 and device C up to height 21. (b) The structure of a transaction. 35
2.6 PREPARE: Generating Balance Lock Attestation 40
2.7 WITNESS: Client A talks with peer infrastructure devices to transfer 25 dollars to B. 41
2.8 FINALIZE: Client A gathers a BUA and unlock the offline balance. 43
2.9 Maximum, median, and mean bandwidth with standard deviation experienced by two Android devices running Vegvisir reconciliation as a function of distance. 45
2.10 Average reconciliation success ratios for various block sizes. 47
2.11 CDF of block spread for various block sizes. 48
2.12 CDF of the dissemination duration from a random block's generation to its ultimate receipt at the most remote participants in the system. . . 48

3.1 A time-space diagram of BloomBox recovery, with time going left to right. There are four devices in the diagram, named A through D. The dotted double-headed arrows represent reconciliation between two devices. The top left corner contains a list of hash mappings from `key:replicaNumber` to a location `(x,y)` on the map. The origin of the map is at the top left corner and the y-axis grows down vertically. The locations and moving directions of the devices in each epoch are shown above the time-space diagram. The rounded boxes next to the device names display data that each device has stored. For example, for device A at time e_0, `key1: (1, val1)` means that device A stores replica 1 of the object with key `key1`. 65
3.2 Block availability for computed epoch lengths varying numbers of replicas and ϵ. 68
3.3 Simulation results for independent-failures. The legend for the first three figures is at the top. We ran simulations with three radio ranges, 10, 15, and 20. In all experiments, replicas were placed pseudo-randomly on the map. 70
3.4 Simulation results for dependent-failures for both random replica placement and, in the case of Heartbeat, nearby replica placement. Results for BloomBox are in solid lines and results for Heartbeat are in dashed lines. A circle marker indicates random placement and a triangle marker represents nearby placement. 71

4.1 Server-centric v.s. disaggregated architecture. 81
4.2 Uniservices Overview 88
4.3 Uniservices code decomposition 99
4.4 Uniservices architecture. (1) A trigger event is inserted into the log. (2) The global scheduler finds the execution graph of the triggered application and assigns functions to invokers. (3) Invokers on different resources receiving either *prewarm* or *invoker* commands, load function bodies from the function hub, and (4) launch the functions. Data movement among functions happens through the underlying RDMA network (5). 101
4.5 Video processing design 105
4.6 M1 memory and CPU usage 106
4.7 C1 memory and CPU usage 106
4.8 Normalized memory utilization of Lambda Function and Uniservice. . 107
4.9 Normalized throughput of Nginx and Uniservice on serving different sizes of objects. 108

CHAPTER 1

INTRODUCTION

Traditionally, software applications developed for commodity hardware are constructed atop layers of abstractions provided by the underlying operating systems. These abstraction layers are beneficial in reducing complexity and enhancing modularity. However, they can hinder applications from fully exploiting the capabilities of the lower layers to obtain optimal performance. This becomes particularly noteworthy as we near the end of Moore's Law. Over the past two decades, the scaling of single-core performance has decelerated significantly [15], with the average Instruction Per Cycle (IPC) improvement per generation hovering around a mere 10 to 20 percent. As a consequence, improving application performance has increasingly steered interests towards *distributed applications* and *specialized hardware*. Applications such as databases, machine learning training and inference tasks, and data processing in data science, are progressively being distributed across server clusters in the cloud. Specialized hardware such as Graphics Processing Units (GPUs) and Application-Specific Integrated Circuits (ASICs) are being harnessed for tasks such as machine learning training and blockchain mining [3, 22].

1.1 Challenges

Despite significant advancements in specialized hardware, current commodity operating systems are still designed based on decades-old models that revolve around Central Processing Units (CPUs). These systems were conceptualized at a time when cloud computing was non-existent. As a result, they offer a CPU-focused abstraction to applications. New hardware components are attached to the host via PCI-E connections.

These hardware devices use the CPU for task initiation and rely on shared memory within the same motherboard for device-specific communication. However, this model is increasingly failing to meet the demands of diverse hardware environments for three reasons.

First, this CPU-centric abstraction does not scale well when host applications require additional specialized hardware. This is because the number of devices that an application can use on a host is confined by the capabilities of the motherboard. For instance, the recent trend towards training large language models (LLMs) typically requires hundreds or even tens of thousands of high-end GPUs to perform training and online inference tasks. But due to the limitations of PCI-E bandwidth, a single host motherboard can only accommodate about eight GPUs at full performance. This constraint poses significant challenges for the design of distributed systems for applications such as model training, which require meticulous planning to fully utilize the power of all GPUs. Furthermore, CPUs can also become the bottleneck [132].

Second, the limitations of a motherboard's capabilities mean that when applications require more specialized hardware, they must inevitably draw on devices from a cluster of machines. This leads to significant network communication overhead compared to using a CPU and shared memory. A typical memory's system access latency via CPU usually takes around 100 nanoseconds (ns) [11], but even with a fast RDMA Infiniband network, the latency can rise to at least 1 microsecond (μs). This issue becomes even more pronounced if the application is running in an Internet of Things (IoT) network, such as a Mobile Ad-hoc Network (MANET), where the network may be partitioned.

Third, the CPU-centric abstraction offers a fate-sharing abstraction. This means if the host crashes, the progress and data of the tasks on the specialized devices are also lost, regardless of the different failure rates of each device.

1.2 Research Statement

Considering the aforementioned challenges of the existing CPU-centric abstraction, this dissertation aims to answer the following research question: *What is an effective programming paradigm for developing applications that can utilize resources efficiently and scale well in a heterogeneous computing environment?* In response, we propose a programming paradigm that consists of actors and shared logs. This pairing offers a *resource-egalitarian* abstraction, which allows applications running on various hardware resources to communicate directly through the shared log, rather than relying on CPUs for coordination. This abstraction enables different applications running on various resources to be loosely coupled, simplifying the development and scalability of applications in a heterogeneous computing environment.

To validate this approach, we design and implement two frameworks — one for Mobile Ad-hoc Networks, and another for disaggregated architectures — that follow this proposed programming paradigm. Applications built on top of these frameworks are evaluated, and our results demonstrate that this resource-egalitarian abstraction leads to better utilization and scalability compared to the existing CPU-centric abstraction.

In the rest of this chapter, we will first introduce some foundational concepts and techniques that we will use, and then present our approach in more detail.

1.3 Background

1.3.1 Heterogenity of hardware

A variety of specialized accelerator hardware is now available in both cloud and IoT networks, each serving different purposes. For instance, GPUs and TPUs [28, 113] are extensively used for machine learning tasks because their parallel processing architecture can efficiently perform matrix operations, such as multiplication and addition. In contrast, CPUs are designed for rapid sequential instructions and do not offer as many cores as GPUs. Beyond machine learning training, some scientific tasks, such as genome sequencing analysis, can be executed over ten times faster on FPGAs than on CPUs [20]. Another domain-specific hardware application is seen in blockchain mining. ASICs, circuits specifically designed for a particular use, are utilized in Bitcoin mining to solve hash puzzles and mine bitcoins profitably. An increasing number of such domain-specific accelerators are shaping heterogeneous cloud and IoT environments.

1.3.2 Shared Log

A *shared log* functions as a distributed ledger, comprising servers and clients. It furnishes Append and Read APIs for clients, facilitating total ordering of every record appended to the log via the Append API. Once recorded on the log, all clients perceive the same sequence of the record via the Read API.

Recently, the study of shared log-based applications has surged due to the advent of novel fast logs [39, 40, 59, 83] and blockchains [13, 92]. Tango [41], a framework that eases the creation of distributed data structures over a shared log, simplifies the devel-

opment of decentralized systems. Boki extends this concept further, using a shared-log *metalog* [71] to construct stateful serverless functions.

1.3.3 Blockchains

First introduced by Nakamoto in 2008, a blockchain is a distributed shared log managed by all network participants. A blockchain is a data structure that consists of blocks, each containing a data payload and a cryptographic reference to its predecessor. Blockchain users can verify block integrity by tracing references back to the genesis block. While most blockchains employ consensus to append transactions sequentially, some such as IoTA [23], use a block Directed Acyclic Graph (DAG).

Blockchains are predominantly used for cryptocurrencies and supply chain accountability. Some blockchains are programmable, allowing the creation and execution of smart contracts (like Ethereum smart contracts [124]) — code segments that maintain the program's state on-chain. A universal programming language typically associated with a smart contract can be applied to all services.

1.3.4 Actor Models

The Actor model [69], a computation model pioneered by Carl Hewitt in the 1970s, defines an actor as the fundamental computation unit. An actor can interact with others via message passing and specify actions for incoming messages [68]. Actors can exclusively modify their own states and are designed to process one message at a time. The actor model excels in systems requiring intense concurrency as actors do not share

state, eliminating the need for concurrency control (like employing locks) to maintain system-wide consistency.

1.3.5 Mobile Ad-Hoc Network

A *Mobile Ad Hoc Network* (MANET) is a collection of devices within a specific geographical region, such as a building, forest, or city. Communication between devices is exclusively through peer-to-peer radio communication, with a limited range. As devices in a MANET are mobile and can enter or exit the geographical area at will, there is constant *churn*. Devices can also fail or be turned off and on.

To simplify MANET reasoning, we model it as follows:

- Each device has a specific location at any time;
- The MANET is defined by the devices within a specified geographic area;
- Devices can join and leave the MANET at will. We model device failures as leaving the MANET;
- Devices can directly communicate with a certain group of neighbors within a specific range [103];
- When two devices are within range, they can exchange an arbitrary amount of data;
- Devices have (approximately) synchronized clocks.

1.3.6 Resource Disaggregation

Disaggregated architecture requires a rapid interconnect between resources to compete with server-centric architectures in performance. Emerging device interconnects such as RDMA, Gen-Z, and CXL, can deliver bandwidths exceeding 100 Gbps [43, 46, 94]. As long as latencies are restricted to a few microseconds, most data center applications can run on a disaggregated architecture without significant performance loss [64]. Several hardware disaggregation designs have been proposed [36, 63, 81].

From the software perspective, LegoOS [109] offers a split-kernel operating system design to manage the underlying disaggregated hardware components while presenting a POSIX interface to applications for backward compatibility. FractOS [122] employs a distributed microkernel design and offers resource service APIs for each resource. It is designed specifically to minimize redundant data movement among devices in disaggregated systems. For application interaction, FractOS uses a continuation-based RPC request object to invoke RPC calls.

To effectively utilize the memory pool, systems such as [30–32, 65, 85, 95] either deploy new specialized hardware or design new data structures to efficiently leverage remote memory. In terms of disaggregated storage, industry has already made substantial progress [123]. [119] proposes a method that utilizes remote persistent memory, such as Non-Volatile Memory (NVM), without necessitating a computing unit at the storage server. [35] advocates for exposing hardware disaggregation to applications with features like explicit failure notifications and memory grants and steals. [96] notes the similarities between serverless computing and resource disaggregation, proposing that they could be co-designed to shape the future of the cloud.

1.4 Approach

Our proposal is to construct an appropriate programming paradigm that facilitates the development of heterogeneous applications using the actor model. Briefly, this paradigm will comprise a shared log and a set of specialized actors.

Drawing inspiration from actor models and smart contracts, an application is decomposed into a set of specialized actor functions for domain-specific accelerators. These specialized actors interact via the shared log, which serves as a substitute for the CPU in a CPU-centric abstraction. Each actor is assigned an address and a mailbox. The address is obtained by registering the actor to the log via a log record, while the mailbox is a queue of records addressed to the actor.

An actor is triggered by an invocation record from the shared log, and subsequently performs the actor action asynchronously in relation to other actors. The output of an action is then placed on the log as a record and can further trigger other actors.

This programming abstraction addresses the problems mentioned at the beginning of this chapter in some aspects. First, hardware is attached to the system through the network, thus it is not restricted by a single motherboard and applications can scale out with more specialized actors on specialized hardware devices under the same abstraction. Second, since actors operate asynchronously to each other, the network overhead would not block a running actor, thereby improving resource utilization. Lastly, the progress of tasks can be shared on the log so that the progress can be restored in the event of a host crash. Specifically, in a disaggregated environment, data can be backed up on a remote memory pool, allowing for fast recovery from a host crash by reconstructing the data from the in-memory data.

1.5 Roadmap

The rest of this dissertation presents three projects that leverage this programming paradigm to build applications for partitioned networks and disaggregated architectures.

1.5.1 Vegvisir

In chapter 2, I introduce Vegvisir, a novel blockchain that is aware of network partitions and supports double-spend-free offline payments. Vegvisir utilizes the proof-of-witness from trusted infrastructure nodes and local data storage for balances to prevent double spending. I also provide an informal proof and case studies to demonstrate offline payments using Vegvisir.

1.5.2 BloomBox

In chapter 3, I address a storage overhead issue inherent in Vegvisir. Vegvisir's strategy of storing all blocks everywhere to ensure availability creates substantial storage overhead, especially for mobile or Internet of Things (IoT) devices. BloomBox is a failure detection protocol for geographic hash tables which stores blocks only at certain geographically mapped locations. I introduce a new variant of bloom filters to efficiently detect failures of block replicas.

1.5.3 Uniservice

In chapter 4, I discuss Uniservice, a programming paradigm for building applications that are aware of resource disaggregation. I first describe the structure of a Uniservice, then present the creation of a Uniservice serverless platform. I demonstrate its application by implementing a video processing service and a machine learning training application on the platform.

1.6 Contributions

In this dissertation, we introduce a resource-egalitarian programming paradigm that incorporates actors and shared logs, aimed at enhancing the scalability and utilization of applications in a heterogeneous computing environment. We develop and assess two frameworks under this programming paradigm in the context of mobile ad hoc networks and resource disaggregated systems.

In the mobile ad hoc network setting, we design, implement, and evaluate Vegvisir, a blockchain that supports the development of applications using actors on heterogeneous Internet of Things (IoT) devices. To address the issue of Vegvisir's storage overhead, we design and evaluate BloomBox, a failure detection protocol for block replications within mobile ad hoc networks.

As for resource disaggregated systems, we create and assess the Uniservice framework. This framework allows cloud applications to be broken down into fine-grained uniservices that are tailored for a specific resource server and communicate via a distributed shared log.

CHAPTER 2
TAMPERPROOF PROVENANCE-AWARE STORAGE FOR MOBILE AD HOC NETWORKS

Recently, there is renewed interest in mobile ad hoc networks, given new applications that cannot rely on continuous internet availability. Such application areas include emergence response, self-driving cars (Vehicular Autonomous Networks), distributed robotic applications, digital farming, and the Internet-of-Things (IoT). Such applications require a convenient and secure communication and storage infrastructure. It is also often desirable that such systems support *accountability*, so that events can be traced and the causes of actions can be attributed [126].

Blockchains are attracting considerable research attention as a distributed computing model with the potential to revolutionize finance, transport, and supply chain management [23, 38, 44, 54, 88, 92, 114, 124]. Blockchains are desired for their decentralization, consistency, and tamperproof properties and seem at the face of it ideal to address the concerns raised above. However, existing blockchain protocols heavily use power and networking resources by design. The heavy use of power prevents deployment on IoT devices while the heavy use of networking prevents deployment in partitionable networks.

This chapter introduces a new blockchain design called *Vegvisir* designed specifically for mobile ad hoc networks. The blockchain's integrity is based on a combination of Conflict-free Replicated Data Types (CRDTs) [111] and a guaranteed "happens-before" order on transactions that operate on one or more CRDTs. CRDTs allows continuous availability despite network partitions. Moreover, Vegvisir guarantees that transactions are tamperproof—once a correct device has acted on a transaction it can no longer be lost.

Vegvisir is not suitable to supporting cryptocurrencies: it can detect but not prevent double-spending. However, we believe that for many applications the CRDT consistency properties combined with tamperproofness is sufficient. While Vegvisir only provides a partial "happens-before" ordering on transactions, Vegvisir can be used to hold devices and their users *accountable* because blocks are tamperproof. Through accountability, Vegvisir incentivizes users to behave well. Also, the properties of CRDTs prevent inconsistencies in the data structures used by applications.

The contributions of this chapter include the following:

- Design of a middleware for providing mobile ad hoc network (MANET) devices in low-power and partitionable network environments with tamperproof provenance-aware storage, even when some fraction of devices can be Byzantine.
- Analysis of the system design that allows Vegvisir to operate in low-power and network partition-tolerant environments.
- Experiments evaluating performance differences between a mining-based protocol and Vegvisir in low-power and network partitionable environments.
- Discrete event time simulations using realistic traces that evaluate the scalability of Vegvisir.

The remainder of the chapter is organized as follows. Section 2.1 provides background and motivation for our work. In Section 2.2, we describe the design of Vegvisir, especially its tamperproof provenance-aware storage that can tolerate a fraction of Byzantine devices. We describe Vegvisir's implementation in Section 2.3 and applications built and deployed using Vegvisir. In Section 2.5, we evaluate the scalability of Vegvisir and compare its performance to a mining based approach. We discuss related work in Section 2.6 and conclude in Section 2.7.

2.1 Motivation

Today's blockchain protocols are resource-hungry and MANET devices generally are resource-constrained. So-called permissionless blockchains use much energy. Individual participants ("miners") have to continuously work to attempt to solve so-called cryptopuzzles. Bitcoin currently consumes 88.3 TeraWatt-Hours in aggregate per year [2,56], more than the rate of consumption of Switzerland (8.3 million citizens). It is estimated that all of revenue from Bitcoin goes back to paying the electricity that it consumes and is pure waste [1], not to mention the greenhouse gases produced. Even if MANET devices had access to a cheap source of continuous energy (say wind or solar) and could compete with the ASIC devices that Bitcoin miners deploy, their participation in a permissionless blockchain would further explode the amount of energy waste and possibly pollution.

Both permissionless and permissioned blockchains require significant networking resources. Typically every transaction has to be stored by at least half of the participants before it can become finalized and actionable. But this knowledge dissemination requires a quadratic communication load, as each participant has to learn if most of the other participants has stored the information. This is slow and scales terribly. Permissionless blockchains typically use a gossip network for this, while permissioned blockchains, based on Byzantine consensus or voting protocols, will have to use some kind of dissemination tree to be able to scale to more than a few hundred participants. But even so, the protocols would likely not scale to more than a few thousand participants, well shy of the potential number of MANET devices.

Blockchains do not only consume a lot of network bandwidth and require that the network is available continuously. In the case of permissionless blockchains, network

partitions could lead to so-called “forks” in which the blockchain splits into multiple separate chains and inconsistencies would arise that are difficult to reconcile. Also, such forks could drastically reduce the rate of transactions, as in each partition there would be fewer participants to solve the cryptopuzzles while the difficulty of those puzzles take significant time to adjust. In the case of permissioned blockchains, a network partition that isolates more than a third of participants would bring the blockchain to a halt.

Unfortunately, MANET devices are often deployed in challenging conditions and may even be mobile, and their connectivity may be intermittent. In a regular blockchain, the rate of transactions is generally independent of the number of participants in the blockchain. In Bitcoin, if you double the number of miners, the transaction rate remains constant but the difficulty of the cryptopuzzles increases. So even if that doubles the amount of energy waste and quadruples the load on the network, the rate of storage growth remains the same. However, each MANET device may generate data. If you double the number of MANET devices, you double the rate of data generated. Ideally all this data should be stored for longitudinal studies. Moreover, the data may be needed long-term for chain of custody.

This chapter demonstrates a design, implementation, and initial evaluation of a blockchain specifically for the low-connectivity, low-power, high data rate MANET setting. The blockchain techology should be tamperproof, but it should also tolerate network partitions well and use a low-power consensus mechanism. Instead of resolving forks, it will need to permit them, resulting in a Directed Acyclic Graph (DAG) structure of the blockchain rather than a linear one. There are other blockchain designs that have embraced a DAG structure. However, they do so to increase the transaction rate, not to tolerate partitions. The cost of this partition tolerance is that the types of applications that can be implemented with the blockchain are limited to ones that only require

a partial ordering of logged events. To this end, we will explore applications based on Conflict-free Replicated Data Types (CRDT) that can work with partial orders.

2.2 System Design

In Vegvisir, devices store state and communicate through a shared storage layer. Because the network is partitionable but consistency is still desired, Vegvisir embraces Conflict-free Replicated Data Types (CRDTs) [111]. A CRDT has the property that two states of a CRDT can be merged into a new CRDT with intuitive semantics. A good example of a CRDT is an append-only set. Elements can be added concurrently in partitions of a network, and upon reconciliation the sets can be merged by taking the union of the sets. There is an extensive variety of CRDTs defined in the literature.

In Vegvisir, a transaction is an operation on a CRDT. Vegvisir maintains a partial ordering between transactions and can define CRDTs that exploit this partial ordering. A Vegvisir CRDT (WCRDT) is an object whose state is uniquely determined by a partially ordered set of transactions. Vegvisir maintains a *block DAG*, that is, a Directed Acyclic Graph of blocks. Each block contains a sequence of transactions on possibly multiple WCRDTs. The order of transactions within a block and the directed edges between blocks encode the partial order between all transactions. More precisely, a transaction t_1 on a WCRDT is before another transaction t_2 if and only if:

- t_1 and t_2 are in the same block and t_1 comes before t_2 in the block; or
- t_1 and t_2 are in different blocks and there is a path of the block containing t_2 to the block containing t_1.

Therefore, the Vegvisir Block DAG (blockdag) exactly determines the state of each WCRDT stored in it.

The blockdag is maintained by a collection of devices, but not all of them are expected to behave correctly—they may crash or even deviate from prescribed protocols arbitrarily (i.e., Byzantine behavior). Maintaining the integrity of the blockdag is therefore a major challenge. Devices can try to create cycles in a blockdag or add new blocks relentlessly and create a blockdag that wildly branches instead of approximating a linear blockchain. Another complexity is how to ensure *tamperproofness*: blocks should never get dropped from the blockdag.

To manage the complexity of Vegvisir, it has four tiers of abstraction. From top to bottom, these tiers are:

1. **The Application Tier** runs applications built using CRDTs to maintain their state.
2. **The Pub/Sub Tier** provides communication channels between processes running on devices.
3. **The Block DAG Tier** maintains the blockdag with byzantine fault tolerance.
4. **The Reconciliation Tier** contains an efficient algorithm to reconcile state among devices.

2.2.1 Application Tier

Vegvisir applications are built on top of CRDTs. CRDTs enable devices to independently update their states without any remote synchronization and guarantee consistency as long as their concurrent transactions commute.

A simple example of a CRDT is a *2P-Set* [111]. It is implemented by two append-only sets: an ADD set and a REMOVE set. The set difference between these two sets determines the current state of the CRDT state machine.

Leveraging causal relationships between transactions, we refine the notion of a 2P set and introduce a new CRDT, a *2P+* set. When an add and delete on the same element are executed concurrently in normal 2P sets, then delete wins. However, if an add happens causally after a delete of the same element, the element is added back to the set, unlike a regular 2P set. This can be formalized as follows: with each $+x$ (add x) and $-x$ (remove x) transaction in the causal graph of 2P+ transactions (where each transaction may depend on a set of other transactions), we associate an *add* set and a *delete* set. They are defined as follows:

$$+x.add = \bigcup_{t \in +x.deps} t.add \cup \{x\}$$

$$+x.delete = \bigcup_{t \in +x.deps} t.delete \setminus \{x\}$$

$$-x.add = \bigcup_{t \in -x.deps} t.add \setminus \{x\}$$

$$-x.delete = \bigcup_{t \in -x.deps} t.delete \cup \{x\}$$

The first of these specifies that the add set after some particular `add`(x) transaction is the union of the add sets of its ancestors and x itself. The others are defined similarly. The application-visible content of the 2P+ set after a transaction is then the difference between its add set and its delete set.

The 2P+ can be further generalized in various ways. For example, it is trivial to define a 2P− set in which addition wins instead of delete to resolve concurrent conflicting transactions. Going further, we can define nP+ and nP− sets for values of $n \geq 2$.

One can think of these as prioritized sets, with n priorities. The priority of elements can be changed simply by moving them. We have found these very useful for building applications (Section 2.3).

We envision doing similar generalizations for a variety of other existing CRDT objects.

2.2.2 Pub/Sub Tier

The Publish-Subscribe, or Pub-Sub, tier is a multiplexing layer between the application tier and the underlying block tier. It allows multiple applications to cohabitate on a device while using the same blockdag. The application and Pub-Sub layer communicate transactions via *channels*. A channel is queue where applications can publish new transactions and subscribe to incoming transactions. An application can subscribe to multiple channels.

2.2.3 Block Tier

In this section we describe the defenses built into the blockdag design to keep it manageable and to encourage good behavior by devices.

Ensuring an Acyclic blockdag

The blockdag is maintained by a collection of devices, each identified by a public key. Each block in the blockdag is created and signed by a device and is uniquely identified by a cryptographic hash. Each block contains the hashes of its *ancestor blocks*, exactly

forming the edges in the DAG. *The properties of cryptographic hashes prevent cycles in the blockdag.* A correct device will only accept (i.e., store) a block if it has all the ancestor blocks. One or more blocks do not have ancestor blocks—it is up to each device to decide which of those blocks it will accept. Typically a device is configured with one such block, called the *genesis block*, and thus there is a path from each block on the device to the configured genesis block.

Permissioned Growth

The blockdag contains an authorization mechanism. Each WCRDT defines an Access Control Matrix (ACM) that itself must be a WCRDT. An ACM specifies which principals are allowed to do which transactions on which CRDTs. Its state is uniquely determined by the partially ordered set of transactions on the ACM contained in the blockdag. In its simplest form, an ACM is a *2P+* set that determines which devices are allowed to operate on the WCRDT and its ACM. A device is authorized to delegate rights that it has to other devices, but cannot create new rights.

There are special WCRDTs called "registries." A registry describes a set of WCRDTs maintained by the blockdag. A registry defines transactions to add and remove WCRDTs from that set and also contains an ACM specifying the devices that may execute those transactions. A registry is always created in a genesis block. A transaction is permissible in a block if the author has a valid standing within the ACM and the transaction is in the set of allowable transactions. A block with a disallowed transaction is considered invalid. Since a transaction is signed by its owner, this is considered a *Proof-of-Misbehavior* (PoM) of that owner.

Note that while it may appear that the blockdag is a *permissioned* block DAG, any device can create a new self-signed genesis block and participate. Two disconnected blockdags can be connected by creating transactions that depend on blocks in both blockdags, as shown in Figure 2.1. However, each individual WCRDT is protected through the relevant ACMs.

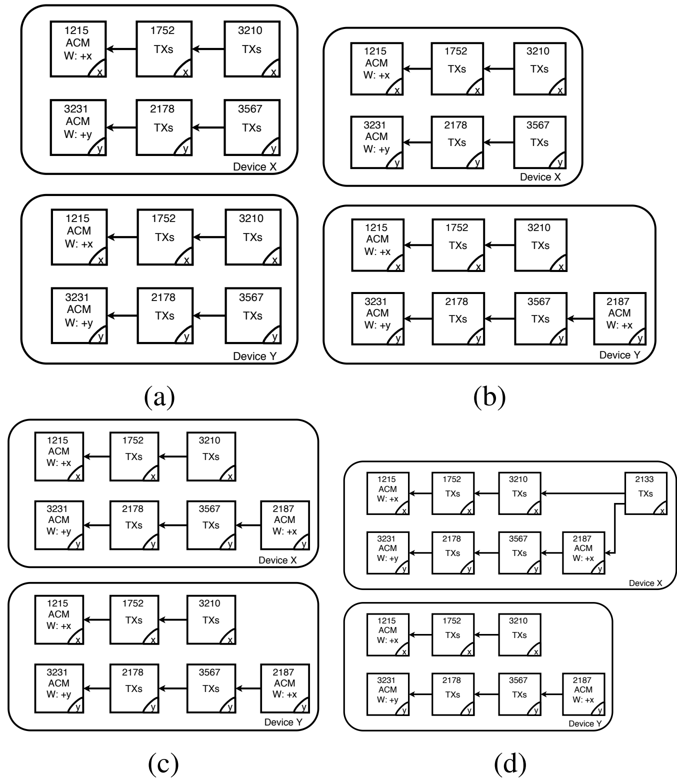


Figure 2.1: From (a) to (d), this diagram shows how two disconnected blockdags can be connected. Each block displays a 4 digit unique hash identifier and the owner device in the lower right corner. In the beginning, in figure (a), two blockdags are created by device X and device Y with different genesis blocks 1215 and 3231. For convenience, we use the genesis block number to identify a blockdag and assume that there is only one WCRDT for each blockdag. Now device Y wants device X to be able to append blocks for blockdag 3231. From (b) to (c), device Y creates a new block with a transaction adding device X to blockdag 3231 WCRDT's ACM with write permission. Then, the newly added block 2187 is synchronized with device X through reconciliation. Finally, device X appends a new block 2133 that connects blockdag 3231 and blockdag 1215.

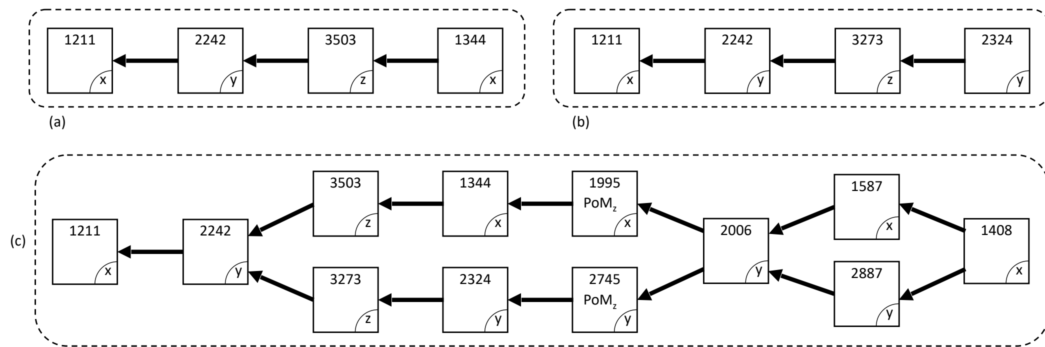


Figure 2.2: Three valid blockdags. (a) and (b) are two snapshots that show that device z is misbehaving because it has created two blocks (3503 and 3273) that do not depend on one another. As a result, the WCRDTs in the blockdags may have diverged. In order to merge the blockdags, both x and y have to add a PoM for z before they can create blocks that depend on both blocks from z. (c) shows a merged blockdag with additional blocks from x and y added. Neither x nor y has concurrent blocks. z is no longer able to add blocks to the end.

Limiting Branching

Ideally, the blockdag would be strictly linear like a conventional blockchain, providing a total order on all transactions. Given the partitionable nature of the environment, this goal is not achievable while allowing progress at all times. However, we impose various constraints to encourage the blockdag to conform to a shape which is “long and thin”. In particular, for any two blocks of a correct device, there is always a path from one of the blocks to the other. In other words, a correct device does not generate concurrent transactions.

On the other hand, a Byzantine device might create concurrent transactions. Concurrent blocks are another form of PoM. The evidence of this misbehavior is traceable back to the signature the device used to sign these blocks. A PoM can be used as a transaction to remove a device from all ACMs in the blockdag. See Figure 2.2 for an illustration.

We impose the following additional constraints on the blockdag to force devices to recognize PoMs and prevent further transactions from the delinquent devices:

- each device is only allowed to add one PoM transaction for any other device;
- a block can only have concurrent ancestor blocks of some device if the path to each such block contains a PoM transaction for that device.

A PoM creates a chain of events within the system. First, the access rights to any WCRDT that the device had are revoked. This includes any CRDTs that the device created. Additionally, the PoM revokes all delegated rights the device has granted to other devices. PoMs incentivize correct behavior by devices and their effect cannot be circumvented by devices delegating their rights to other, possibly colluding, devices. Devices may, however, have received the same rights from other devices and those rights will remain intact.

We rejected an alternative design in which blocks from misbehaving devices are automatically removed from the blockdag. The reason for our choice is that correct devices may already have acted upon transactions from those devices, and those actions may be difficult or even impossible to undo.

Tamperproving

Each device maintains an instance of a blockdag and opportunistically gossips with other devices to disseminate blocks. A device will not try to reconcile with another device if it has a PoM for the other device. A correct device never drops blocks, but Byzantine devices may drop blocks from their blockdags. Also, even devices that do not drop blocks may crash or break in some unrecoverable fashion. This poses a problem, as

applications may rely on blocks on the Vegvisir block graph to be *tamperproof*. Digital signatures prevent blocks from being modified, but not from being dropped. A block is tamperproof only if it is stored on a correct device. Note that because correct devices only store blocks if it stores all its ancestors blocks, a tamperproof block implies that its ancestors are tamperproof as well.

The *system blockdag* is defined to be the collection of tamperproof blocks. It is a strictly growing, but virtual blockdag. Applications typically will take actions only on the basis of what they know about the state of the system blockdag, not the blockdag instances in the devices themselves. But an application cannot know exactly which devices are correct, and thus cannot directly determine which blocks belong to the system blockdag.

To this end, Vegvisir defines a set of so-called *Survivor Sets* [73], which are subsets of devices that are assumed to have at least one correct device. An additional assumption is that at least one survivor set consist of only correct devices. For example, if there is a fixed set of devices and there is an assumption that at most t devices are faulty and the majority of devices is correct (i.e., there are at least $2t + 1$ devices total), then the survivor sets could be exactly those sets that have more than t devices. However, we do not require that survivor sets intersect. Such a requirement is unnecessary and makes progress harder when network partitions occur.

A "witness" is a cryptographic hash of a block signed by a device, certifying that that device stores a copy of that block and all its ancestor blocks. A witness therefore acts not only on the identified block but also on all its ancestor blocks. A "Proof of Witness" (PoWi) for a block is a set of witnesses for the block, one from each device in a survivor set. (Note that PoMs for any of those devices do not invalidate the PoWi because at least one device in the survivor set is correct by assumption.) A PoWi guarantees, by

the assumptions on survivor sets, that the block and its ancestors are tamperproof and therefore in the system blockdag. Also, by the assumption that at least one survivor set consists of only correct devices and further assuming that blocks and witnesses eventually disseminate to all correct devices (Section 2.2.4), it is always possible to construct a PoWi for a tamperproof block.

Each correct device maintains a monotonically growing set of witnesses for its blocks. Note that a device needs to maintain only one witness for each other correct device as there are no concurrent blocks for correct devices. There can be multiple "concurrent witnesses" for blocks from the same Byzantine device. A correct device only stores witnesses for blocks that it stores. When two devices communicate they reconcile not only their local blockdags but also their witness sets (Section 2.2.4). Note that both the blockdag and the witness set of a correct device grow monotonically.

Checkpointing

So far we have assumed that all devices have unlimited storage. Besides the problem that this might be an unrealistic assumption, it also leads to potentially excessive amounts of communication needed to reconcile blockdags.

To solve this problem, a WCRDT can support *checkpoint* transactions that summarizes its state in a way that still allows partitionable operation. In particular, checkpointing preserves the ability to compute a unique state from the partially ordered transactions on the WCRDT. Most WCRDTs already natively support this as they do not need to maintain the history of transactions in order to compute the state. For example, in 2P+ sets, it is only necessary to store the add set and the remove set—not their histories. The problem is how to ensure that a Byzantine device does not create invalid checkpoints.

PoWis can also solve this problem. A correct device can verify checkpoint transactions locally and will not accept a block that contains an invalid checkpoint transaction (although such blocks can be used as PoM). A block containing a checkpoint transaction with a PoWi can be trusted. This allows devices to potentially garbage collect blocks to reduce storage and communication. In particular, blocks that are no longer needed to compute the state of any WCRDT can be safely dropped. Should provenance be required for accountability, then a summary of the provenance can be included in the checkpoint.

Rate Limiting

A Byzantine device can try to create WCRDTs and/or transactions at an unlimited rate. Even if we limited the rate at which a device can add transactions to the blockdag, it could authorize a never ending stream of new devices to add more transactions (aka a *Sybil attack*).

Taking advantage of the fact that applications usually create transactions at a known maximum rate, we employ the following solution currently: each WCRDT specifies at what rate each device is allowed to add transactions to the blockdag. When a device authorizes another device, it must split its rate with the new device. Therefore, the two devices together cannot increase the rate at which the blockdag grows.

To support this, we assume that devices have clocks loosely synchronized with real time. Each block has a timestamp. If b_1 is an ancestor block of b_2, then the timestamp on b_1 must be before that of b_2. Also, correct devices only consider blocks that have timestamps before the time on their clocks. Each device can compute the rate at which other devices are adding transactions to a WCRDT and can use this to detect devices

exceeding their rates. Such detections can be used as PoMs as well because they can be verified by any device.

Summary

Byzantine devices can try to mount three types of attacks: equivocation by submitting concurrent transactions, Denial-of-Service by overwhelming the system with new blocks, and removing blocks that correct devices have already accepted by acting as witnesses but then dropping the blocks. Conventional blockchains prevent these either by making blockchains permissioned—controlling who can participate—or using incentive-based mechanisms such as Proof-of-Work and Proof-of-Stake.

Vegvisir is a permissionless blockchain and allows any device to join, but places restrictions on who can do certain transactions and at which rates on certain WCRDTs. Vegvisir cannot prevent Byzantine devices from *equivocating* by creating concurrent blocks because of our requirement to maintain availability in partitioned networks. However, such concurrent updates from the same device are detectable and result in such devices losing their access rights, providing incentive not to mount such attacks. Sybil attacks by delegating rights to other existing devices or even fake devices can prevent detection. However, WCRDTs remain consistent even in the presence of concurrent transactions. Also, taken together, a Byzantine device and its Sybils cannot exceed the rate of block creation assigned to that device. Proofs-of-Witness and the properties of Survivor Sets prevent Byzantine devices from deleting blocks that correct devices have already accepted.

2.2.4 Reconciliation Tier

Vegvisir's Reconciliation Tier is responsible for reconciling blockdags and witness sets between devices. In a mobile ad hoc network, reconciliation is initiated every time two devices are within each other's communication radius. This is different from a static network infrastructure where devices gossip, or periodically synchronize updates, with one another. As the period when two devices are within communication range may be limited, reconciliation must be fast. In the following section, we discuss approaches we took to address this challenge.

Reconciling blockdags

One approach to reconcile the blockdags of two devices is to exchange all blocks. This *Send All Blocks Protocol* (SABP) can be improved by acknowledging blocks and devices remembering, for each other device, the blocks for which they have received acknowledgments. While simple and reliable, it is inefficient as devices are generally expected to have many of the same blocks already even if they have not communicated before, and thus the protocol may end up sending blocks that the peer device already stores. The challenge, then, is for communicating devices to determine which blocks they hold that their peers do not already have.

Vegvisir could use an existing *set reconciliation protocol* [89] to reconcile blocks. However, there are two reasons why we have rejected this for blockdag reconciliation. First, set reconciliation protocols do not exploit existing structure in blockdags that can simplify and optimize reconciliation. Second, we exchange blocks in oldest-first order to ensure that the communicating nodes can make progress. Otherwise, some transferred blocks would not be able to be applied to a device's blockdag upon receipt and would

incur delays as devices waited until all of the block's ancestors have been received as well. This second issue could easily be exploited by Byzantine devices to waste precious communication bandwidth.

One approach that exploits the structure of blockdags is what we call the *Frontier Set Reconciliation Protocol* (FSRP). In this protocol, the devices start out with sending only the source blocks in their blockdags, that is, those blocks that no other blocks depend on. On receipt of such a block *b*, there are three cases:

1. the device has *b* in its blockdag. In this case, it sends to the peer device all blocks that depend on *b*.
2. the device does not have *b* but it has all the blocks that *b* depends on. In this case, the device adds *b* to its blockdag.
3. the device is missing blocks that *b* depends on. In this case it buffers *b* and requests from the peer the missing blocks.

The FSRP protocol avoids most of the duplication that occurs with the SABP protocol. The protocol can be further optimized by, in the first round, only sending the cryptographic hashes of the source blocks, thus completely eliminating sending blocks to a device that already stores them.

A problem with FSRP is that missing blocks are sent in most-recent-first order. On receipt those blocks generally cannot be accepted until it receives a block for which the device already stores the ancestors. At that point all buffered blocks can be added to the blockdag. But should the communication link break before then, the buffered blocks are useless and communication was potentially wasted. While perhaps unlikely in practice, this effect can be exploited by Byzantine devices to send endless strings of useless blocks to a device. We reduce the potential harm by exchanging *block manifests*

of that only contain hashes of blocks and dependencies. On receipt of a block manifest for which the device stores the dependent blocks, the device requests the content of the missing blocks from its peer in order.

FSRP is slow because it goes through phases rather than streaming blocks one after another. The final protocol that we will describe is the *Hashed Vector Timestamp Protocol* (HVTP). In this protocol, we exploit that correct devices do not generate concurrent blocks, and therefore blocks from a correct device can be identified by a simple counter. In particular, if there were no Byzantine devices, devices could simply exchange *Vector Timestamps* consisting of one such counter per device. On receipt, a device can easily determine which blocks its peer is missing and send it the missing blocks in oldest-first order, allowing the peer to accept each block on arrival without the need for buffering.

With Byzantine behavior we have two problems to deal with. One is that if two correct devices have identical Vector Timestamps, this does not imply that they have the same blockdag. The second problem is that, upon receipt of a block, it is no longer guaranteed that the device stores the ancestors of the block.

The first problem is resolved by enhancing a vector timestamp with a cryptographic hash in each entry. The hash for a device is computed by collecting the most recent blocks for that device (there may be multiple if the device is Byzantine), sorting the blocks into a list, and computing a hash over the list of blocks where both the list and hash can be computed and maintained incrementally. Now, if two correct devices have the same *Hashed Vector Timestamps* they are guaranteed to have the same blockdag.

If devices detect that they do not have the same blockdag, either because they have different HVTs or because receiving a block for which they have no ancestor, then they could revert to one of the other protocols. The current implementation of Vegvisir uses

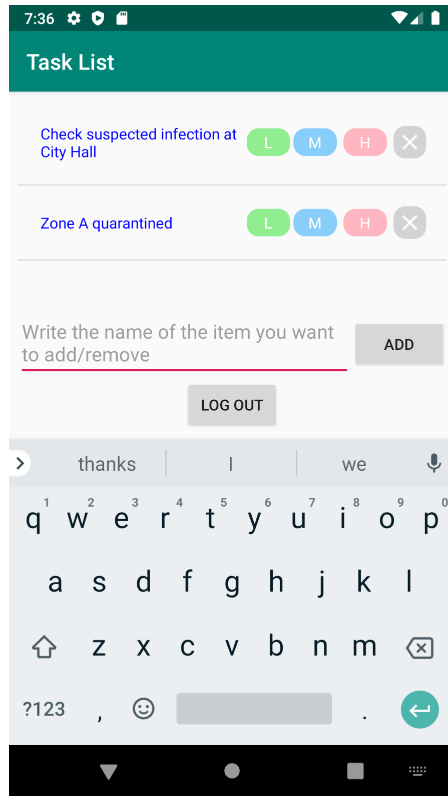
Hashed Vector Timestamp but reverts to FSRP if the protocol fails to reconcile on its own.

Reconciling Witness Sets

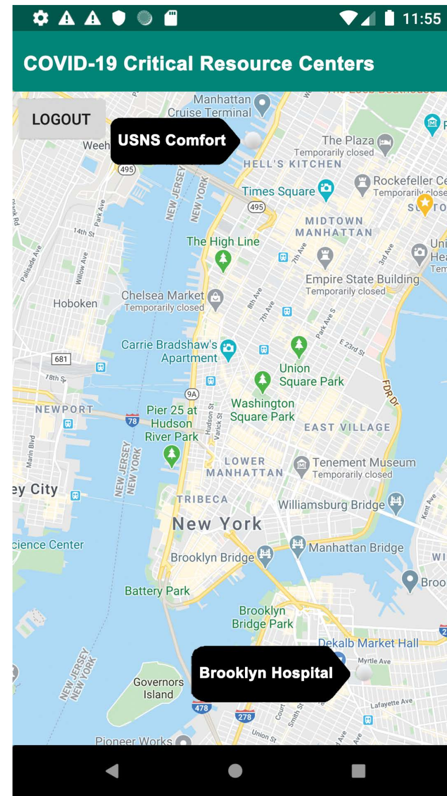
If a device maintained all witnesses, witness sets could be efficiently reconciled with existing set reconciliation protocols. However, a witness for a block also counts as a witness for all ancestor blocks, greatly reducing storage that is needed for witnesses and making set reconciliation protocols redundant. Today, we simply exchange the witness sets. This means that some devices may end up storing witnesses for blocks that they do not store. If space is tight, the device can drop such witnesses. Correct devices never send witnesses for blocks they do not store.

2.3 Vegvisir Prototype

In order to implement Vegvisir into a framework that could be installed on MANET devices as well as on servers, we chose Java as the primary programming language. To demonstrate that two applications can work on the same blockdag, we developed two Android applications for emergency response use: a *Task List* for command and coordination and an *Annotative Map* (see Figure 2.3) for the location sharing. The *Task List* allows users to post and mark completed prioritized jobs on a shared list. The *Annotative Map* allows users to place prioritized annotations on a shared map. Both applications use 2P+ sets (Section 2.2.1) and operate on a mobile ad hoc network. Note that, both applications was running on the same blockdag even though they are using two different WCRDTs. This is possible because both WCRDTs were registered in the genesis block beforehand.



(a) Task List



(b) Annotative Map

Figure 2.3: Screenshots of sample applications

In this section, we discuss the Vegvisir layered structure that supports these applications while also minimizing power consumption. First, we will describe the interfaces on which applications depend in Section 2.3.1, followed by the summary of the Publish-Subscribe layer that enable communication between application instances in Section 2.3.2. Next, we present the Block layer and conclude with Vegvisir's network layer in Section 2.3.4 which talks about the techniques Vegvisir uses to exchange data in MANET.

2.3.1 Application Layer

To interact with Vegvisir, we expose four methods:

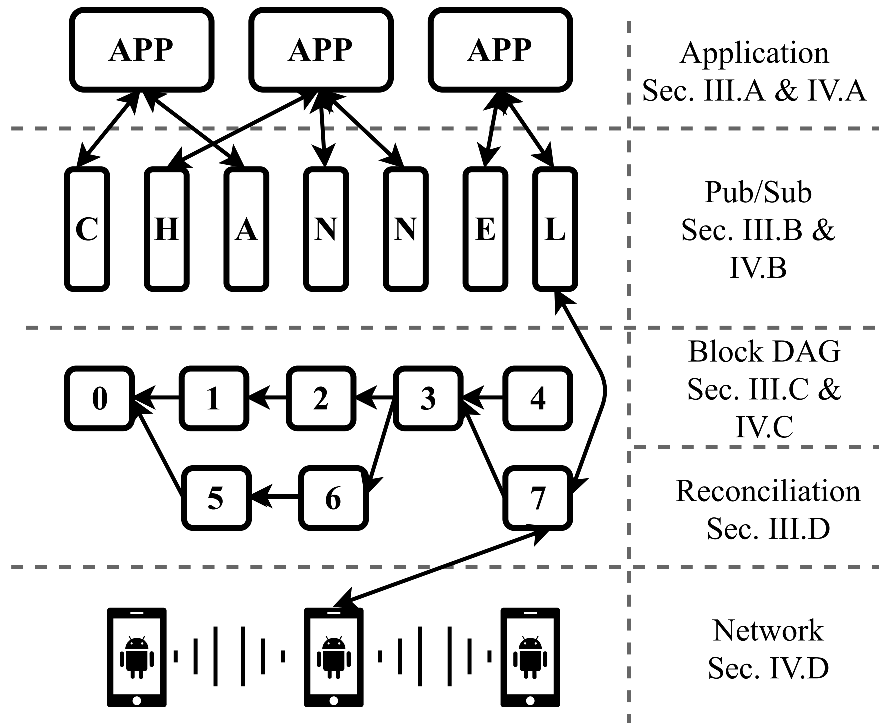


Figure 2.4: Layer structure for Vegvisir

- `registerApplicationDelegator`: lets an application join a subscription service. This function takes two arguments. The first one is a Vegvisir application "Context" object, which contains information about channels to which this application wants to listen; the other one is a subscription handler that will be called for processing transactions received on those channels.
- `addTransaction`: publishes a transaction to a subscription channel. The method takes a Vegvisir application context, the set of targeted channels, the transaction payload in binary format, and a list of dependent transactions. The information is encoded into a Vegvisir transaction with a newly generated transaction id and a timestamp. Finally, the method serializes and sends the transaction to the underlying block layer. Figure 2.5b shows the structure of a Vegvisir transaction.

- `getWitnessesForTransaction`: given a transaction identifier, this function returns a set of identifiers of devices that claim to have stored a copy of the transaction.

An application uses this interface as follows. First, the application invokes `registerApplicationDelegator()` to register a listener for channels of interest and a handler for processing transactions. As in the conventional publish-subscribe model, clients are notified immediately when a transaction from a subscribed channel arrives.

An application uses `addTransaction()` to publish transactions. The new transaction is sent to the specified channels with a list of dependent transactions ids. This function also triggers an event on the local device so that the registered handler for the target channels can be invoked. This improves the user experience because users can immediately see the effects of their actions. However, before collecting enough witnesses, the application should not act on this transaction.

To get PoWis for a particular transaction, an application needs to use `getWitnessForTransaction()`. By calling this method, the application can leverage the notion of survivor sets to devise what witnesses are required before a transaction can be acted upon. Moreover, this method can also help determine how far sensitive information has traveled within a particular network.

Each application determines the survivor sets it uses for the WCRDTs that it uses. Two different applications operating on the same WCRDT can have different survivor sets. This is flexible, but applications have to check PoWis themselves.

Additionally, the application layer is responsible for correctly interpreting the data put inside a transaction. We provide a helper library that presents a set of convenient data structures and supports a transaction format that can be interpreted across devices.

Transactions maintained by the helper library guarantee a correct causal relationship between local transactions.

2.3.2 Pub-Sub Layer

The Pub-Sub layer communicates transactions between applications. The publisher creates blocks that contain transactions, while subscribers monitor the blockdag for such blocks.

A Transaction Message is a data structure defined using Google's Protocol Buffers [26], which allows applications to be written in any programming language. A transaction that contains 5 fields: *Channel*, *Transaction ID*, *Dependencies*, *Timestamp*, and *Payload*. All application-specific information is placed in the *Payload* section of the transaction message as shown in Figure 2.5b. Encryption, if needed, can be performed by the application prior to the data being placed in the payload. The dependencies among transactions must be visible to the Pub-Sub layer. This is necessary in order to eliminate invalid, dangling transactions. Pub-Sub layer is responsible for ensuring only transactions with fulfilled dependencies are published on their associated channels.

We have implemented Pub-Sub as an event loop. Whenever a new block arrives, the following actions are taken:

1. All transactions are extracted from the block.
2. Transactions are validated by confirming the satisfaction of its dependence set.
3. Valid transactions are hashed and placed in a mapping from transaction hash to block hash.
4. Subscribed applications are notified of the presence of new transactions.

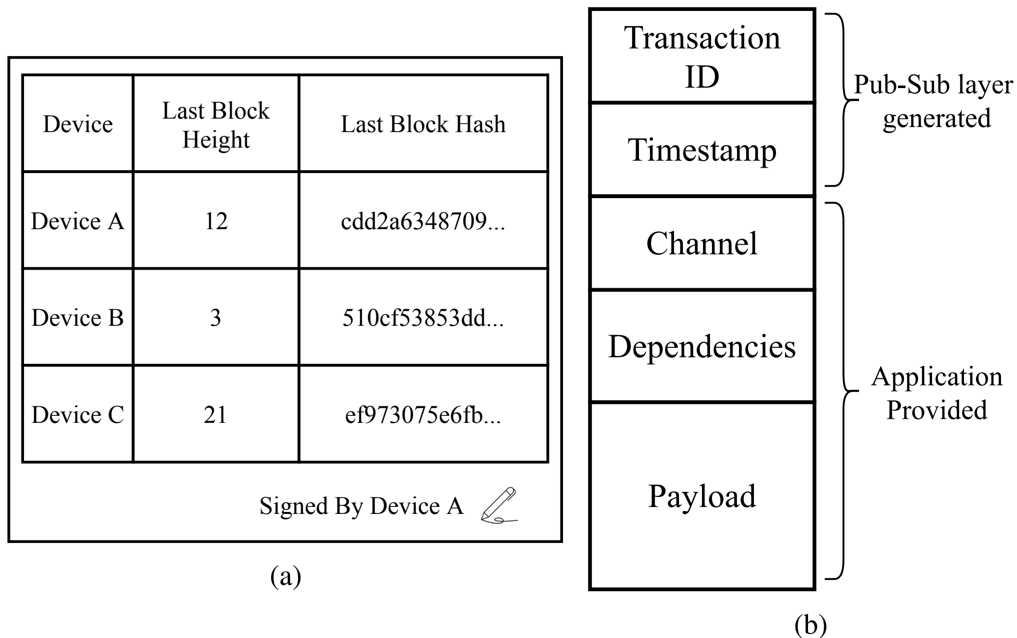


Figure 2.5: (a) The structure of a Hashed Vector Timestamp. The example shows a HVT from device A which has witnessed blocks from device B up to height 3 and device C up to height 21. (b) The structure of a transaction.

2.3.3 Block Layer

A Vegvisir block contains three segments:

- The *header*, which includes the *owner* (currently the public key of the device that created the block), the *timestamp*, the *height* of the block (this is the number of blocks that the *owner* device created thus far), and the *hash* of each of the parent blocks.
- The list of WCRDT *transactions*.
- The cryptographic *signature* of the block header and transactions generated using the private key of the owner.

A device stores each block in a hash map using the hash of the block as key. Additionally, each device maintains, for each other peer device, a *chain object*. A chain

object for a device is a doubly linked list of blocks generated by the same device, used for efficient reconciliation. In order to enable efficient search of witnesses, a device also keeps a set of the last Hashed Vector Timestamps (HVT) for all devices. Figure 2.5a shows the structure of a HVT. To find witnesses for a transaction, Vegvisir first looks up which block contains this transaction. Then, it records the owner of the block o_t and the block height h_t. Next, using the set of the last HVTs, Vegvisir compares the last block height h_d that each device has seen for the block owner o_t and adds devices for which $h_d \geq h_t$ holds to the transaction witness set.

2.3.4 Network Layer

In order to increase code portability and to enhance system modularity, we use Google's Protocol Buffers [26] as our serialization framework. Currently, Vegvisir has two options for network communication: a regular TCP-based solution and Google Nearby [18]. When a WiFi connection to the Internet is available, devices use randomized gossip with known devices over TCP. In the absence of a Wi-Fi base station, Google Nearby leverages Bluetooth and WiFi Direct to establish an ad-hoc network and allow opportunistic peer-to-peer connections among devices.

Using Google Nearby, a device can be in one of two modes: advertiser or discoverer. An advertiser makes its presence known by broadcasting its connection information that discoverers can use to send connection requests. While in theory a device can be in both modes at the same time, in practice this does not work well, and it is better to have a device toggle between one or the other. However, discoverers cannot find other discoverers while advertisers cannot similarly find other advertisers. To solve this issue,

a device switches modes if they have not communicated with another device after a randomized timeout.

2.4 CBDC Application

In this section, we will outline a protocol that utilizes Vegvisir to address the double-spending issue in offline payment systems. Central Bank Digital Currency (CBDC) represents a notable leap in the digital transformation of money. Recently, several countries have announced plans to transform and deploy digital currencies [9, 14, 52]. One persistent issue that hinders deployment of offline payment systems is the prevention of double-spending. Current systems depend on the Trusted Execution Environment (TEE) available on mobile devices such as smartphones and wearable technology. For example, Visa proposed a TEE-Based offline payment system [53]. China Telecom and Conflux Network developed a blockchain-embedded SIM card, Bsim [8], and deployed it in Hong Kong.

However, three primary problems exist with the current TEE-dependent model. First, not all devices have TEE capabilities, and even for those that do, the safety levels and implementations are not uniform. Second, TEE devices are prone to crashes, potentially leading to data loss if transaction data relies exclusively on the TEE on both devices involved in the transaction. Finally, different regions or countries may enforce different hardware requirements for the TEE, complicating international travel. Users might need to replace their device's chip, akin to a currency exchange at an airport, which is an inconvenient solution for frequent international travelers.

To address these challenges, we exploit the locality property of offline payments and propose a proof-of-witness from a quorum of infrastructure nodes (with TEE) to vali-

date transactions. Offline transactions typically occur within a localized range, such as within a district or a town. Infrastructure nodes, like 5G base stations, are generally more reliable than user devices, due to the involvement of governments or large corporations in installation and maintenance, which obviates the need for trust from user devices. Infrastructure can also be upgraded uniformly within a region, reducing inconsistency issues among TEE devices. Furthermore, critical infrastructure is designed to be crash-proof due to its significance. Therefore, we propose a protocol involving offline infrastructure devices and user devices for offline transactions.

2.4.1 Problem Definition

We adopt the problem definition from [53] and define the problem as follows:

A Central Bank Digital Currency (CBDC) payment system consists of two components: the Online Payment System (OnPS) and the Offline Payment System (OffPS). The OnPS facilitates transactions between two clients, given that both are connected to the online server S, while the OffPS ensures transactions can be committed without contacting S.

Specifically, each client, say client A, is associated with an online balance (onBalA) and an offline balance (offBalA). If a payment of amount x is to be made from client A to client B, an online transaction ensures that OnBalA' = OnBalA $-$ x and OnBalB' = OnBalB $+$ x after the transaction. Similarly, for offline transactions, the protocol ensures the balances are updated accordingly: OffBalA becomes OffBalA $-$ x, and OffBalB becomes OffBalB + x.

2.4.2 Offline Payment Protocol

The offline payment protocol, involving two clients A and B, is divided into three phases: PREPARE, WITNESS, and FINALIZE. The setting includes $3f+1$ infrastructure devices situated around A and B of which we assume that at most f can exhibit arbitrary failures. Every dollar in a user's account is either *locked* or *unlocked*. A locked dollar is bound to be spent within a specific geographic region identified by `geoId`. The term `locked(n, geoId)` is used to specify that `n` dollars are locked to the region with id `geoId`. On the other hand, an unlocked dollar is available for use in the cloud or can be locked to another region. *A* **Balance Lock Attestation** (BLA) locks a specific amount of money, while a **Balance Unlock Attestation** (BUA) moves the balance from the attestation back to the user account. The structure of these attestations is defined as follows:

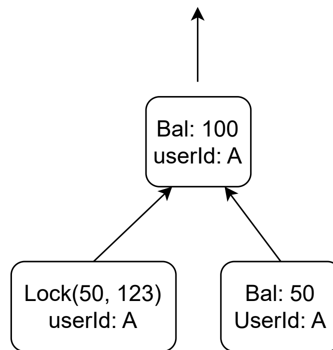


Figure 2.6: PREPARE: Generating Balance Lock Attestation

```
BalanceLockAttestation {
    parentId : String
    userId : String
    geoId : String
    balance : Int
    expiration : Date
    sign : Signature
}


BalanceUnlockAttestation {
    parentId : String
    userId : String
    geoId : String
    balance : Int
    signs : [Signature] // 2f+1 signatures
}
```

Now, let's assume client A intends to transfer x dollars to client B.

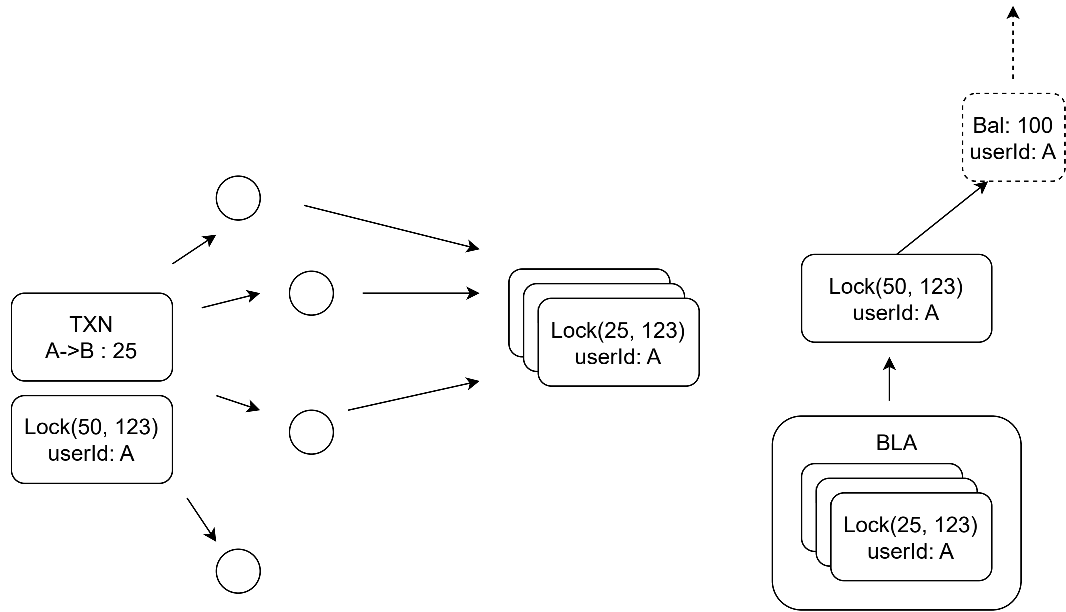


Figure 2.7: WITNESS: Client A talks with peer infrastructure devices to transfer 25 dollars to B.

PREPARE

In order to utilize money from their account, clients must lock a certain amount by obtaining a BLA from server S. This leads to a fork in their balance chain as shown in Figure 2.6, resulting in an offline locked BLA and an online unlocked BLA. In the PREPARE phase, both client A and B must acquire a BLA from the online server S, or possess a BLA witnessed by $2f+1$ different infrastructure devices. Subsequently, clients A and B exchange their BLAs. They then co-sign a transaction stating that client A is transferring x dollars to B, following the alphabetical order of the user ids. Each client keeps a local copy of this transaction on their device.

Witness

Following the PREPARE phase, as shown in Figure 2.7, both clients broadcast their respective BLAs alongside the co-signed transaction to the surrounding infrastructure devices. They then await the receipt of new BLAs from these devices.

When an infrastructure device receives the transaction and the BLA, it first validates the BLA. If the BLA is issued by server S, the device verifies the signature using the public key. If the BLA is from nearby peers, it counts whether the number of valid signatures is $2f + 1$. The device also checks if the BLA has not expired by comparing the local time with the expiration date in the BLA. If the BLA is valid, the device checks whether the balance is sufficient for the transaction. If all these checks pass, the device issues a new BLA with its own signature. This new BLA retains the expiration date from the previous BLA but features an updated balance. The device stores this new BLA for the client. Finally, the device broadcasts the new BLA to clients A, B, and all its peers. If any check fails, the device broadcasts a failure message with the transaction.

Upon receipt of new BLAs from $2f+1$ infrastructure devices, clients A and B consolidate these into a single BLA on their local balance chain, creating a tail block pointing to the last BLA. This concludes the transaction.

Finalize

If the client wishes to use the new balance in a different region, the offline balance needs to be unlocked. This process resembles a normal transaction. To initiate this, a client sends a finalize message with the last BLA to nearby infrastructure devices and collects $2f + 1$ BUAs, which are generated by the nearby infrastructure devices after verifying the last BLA is valid. The infrastructure devices store this BUA until the expiration

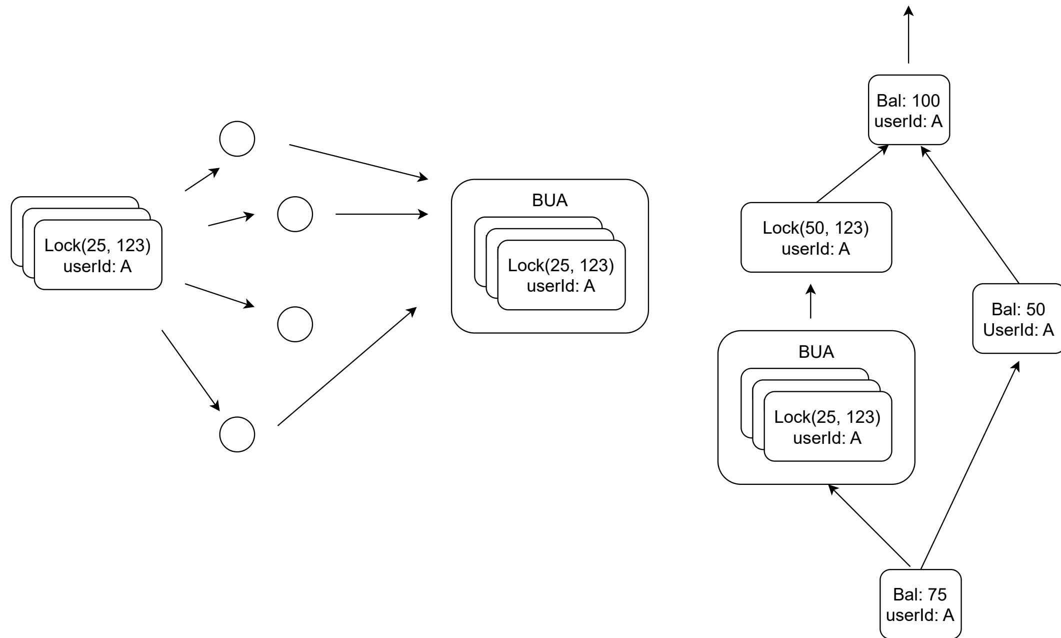


Figure 2.8: FINALIZE: Client A gathers a BUA and unlock the offline balance.

date of the BLA to ensure that the same BLA and its ancestors cannot be reused. The client can then upload this BUA to the server to unlock its offline balance. A diagram representing the finalize phase is presented in Figure 2.8.

2.4.3 Correctness

In the protocol outlined above, we make an assumption that there are $3f+1$ infrastructure devices present in the region, and at most f nodes can experience arbitrary failures. A client gathers $2f + 1$ BLAs as a quorum to confirm a transaction. This ensures that any two $2f+1$ quorums of infrastructure devices will have at least $f+1$ overlapping devices. Consequently, at least one correct device is always available, preventing a client from spending the same BLA twice. If the infrastructure devices are reliable, and only crash failures can occur, the quorum can consist of $2f + 1$ devices. In this case, a client would only require $f + 1$ BLAs to finalize a transaction.

Since a BLA is tied to a specific region, it becomes non-spendable if a client leaves that region. In order to spend the balance at a different location, the client must acquire a new BLA for the new region. A BLA becomes invalid either when it expires or when it is terminated by a BUA.

2.5 Evaluation

To measure the scalability and bandwidth consumption of Vegvisir, we use a simulation based on traces obtained from a taxi company. However, to calibrate the simulation, we also made measurements using a prototype of Vegvisir deployed in a much smaller setting.

The Vegvisir prototype devices used for experiments were six Samsung SM-T510 Galaxy Tab A (Android 9) devices with Li-Ion 6150 mAh batteries. These devices have eight cores, 2GB RAM, and 128GB of internal storage and are equipped with VHT80 antennae that can operate on 802.11 a/b/g/n/ac 2.4G+5GHz. The devices were not connected to a Wi-Fi base station during any of the experiments.

2.5.1 Network Measurements

In order to calibrate our simulation experiments, we performed network measurements using our prototype on devices described above. The experiments used Google Nearby for peer-to-peer communication.

We measured out distances of 5, 10, 20, and 40 meters. In the experiments we ran the Vegvisir reconciliation protocol between devices for transferring 32 MB data. The ef-

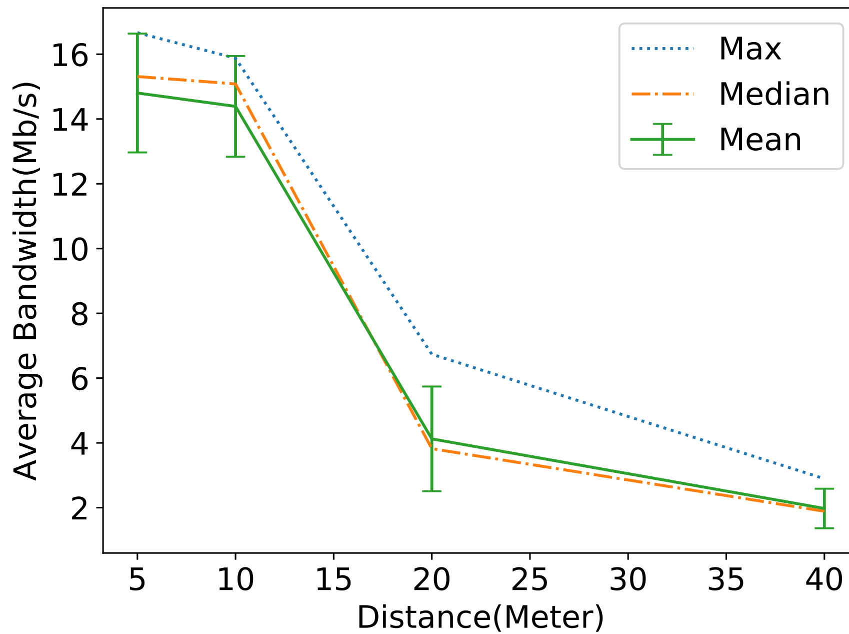


Figure 2.9: Maximum, median, and mean bandwidth with standard deviation experienced by two Android devices running Vegvisir reconciliation as a function of distance.

fective bandwidth at each distance was measured 5 times with different pairs of devices. Figure 2.9 shows the average maximum bandwidth, the overall average bandwidth, and the overall median bandwidth that we measured at various distances. Bandwidth decreases as distance increases because the strength of the signal decreases logarithmically with distance. Measured bandwidth had a significant variance, particularly at small distances. The individual maximum and minimum bandwidths recorded in the experiments were 42.37 Mb/s and 0.01 Mb/s at 5 meters, respectively. We only measured the bandwidth between two devices because, in both prototype and simulation, devices running Vegvisir only communicate with one of its peers at a time.

2.5.2 Scalability

Performing at-scale experiments in realistic scenarios is difficult, and we do not have communication traces at our disposal. Instead, we used a full month of taxicab location traces in the city of San Francisco [100] to run discrete-time event simulation experiments. The dataset is comprised of 490 cabs dispersed throughout the city. The simulation goal is to measure amount of time for a newly generated block to propagate through the peer-to-peer network and confirm the ability of an MANET device to make progress in a partitioned network. The speed at which blocks spread between devices has implications for the tamperproofness of blocks.

The data transfer rates between nearby cabs is determined by linear regression functions deduced from our real-world bandwidth experiments. Specifically, we sampled an appropriate bandwidth from a Gaussian distribution built from the mean and standard deviation functions for the bandwidth data in Figure 2.9. As a result, the functions, where X is the distance in meters, are defined as:

$$\textit{mean-bandwidth}(MB) = -0.39 * X + 16.24$$

$$\textit{standard-deviation}(MB) = -0.29 * X + 13.13$$

As a baseline, we used an ideal scenario where every block exchange between nearby devices occurs instantaneously and successfully. We denote these in the results as *Oracle*.

When two devices are within a sufficiently small range of one another they are able to communicate. While larger blocks lead to better communication efficiency in general, with mobile devices the exchange of a block is more likely to fail with a large block

size because devices may move out of range during the communication. Thus, large blocks could lead to bandwidth being wasted. Using the taxicab traces, we determined the reconciliation success ratios for various block sizes averaged over 20 experiments (Figure 2.10).

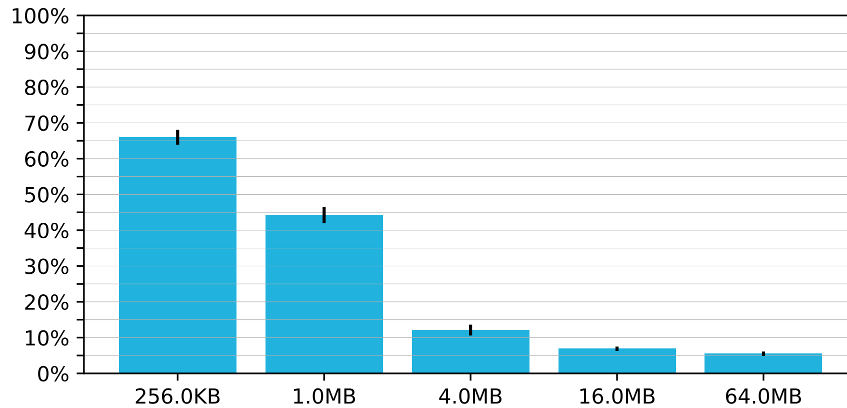


Figure 2.10: Average reconciliation success ratios for various block sizes.

Next, we measured the duration for a single block (generated by a random cab) to reach all cabs in the system using HVTP. As shown in Figure 2.11, the baseline (Oracle) delivers a block to 90% of nodes within 1 hour. For the smallest block size (256KB), Vegvisir delivers similar infection rates as the baseline. For the largest block size (64MB), it takes 2.5 hours to propagate a block to 90% of devices.

Finally, with each taxicab generating a single block at the onset of simulation, we measured the duration for each block to reach all other taxicabs. The number of blocks that a taxicab can disseminate increases over time. Figure 2.12 shows the average spread of 490 blocks circulating in the system. A 64MB block reaches 90% of nodes within 3 hours.

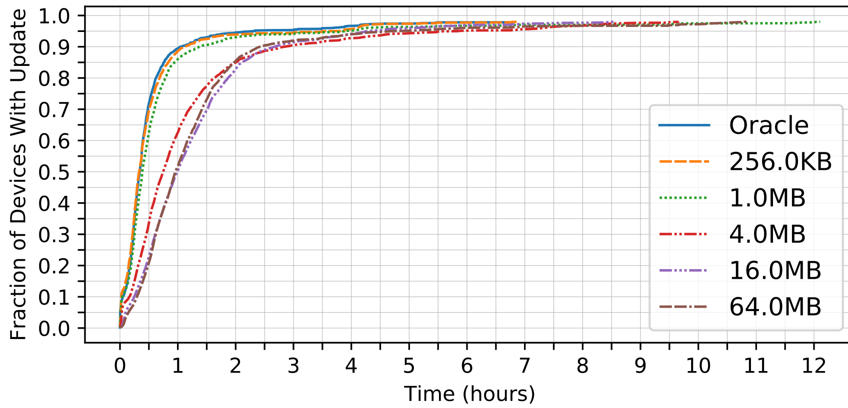


Figure 2.11: CDF of block spread for various block sizes.

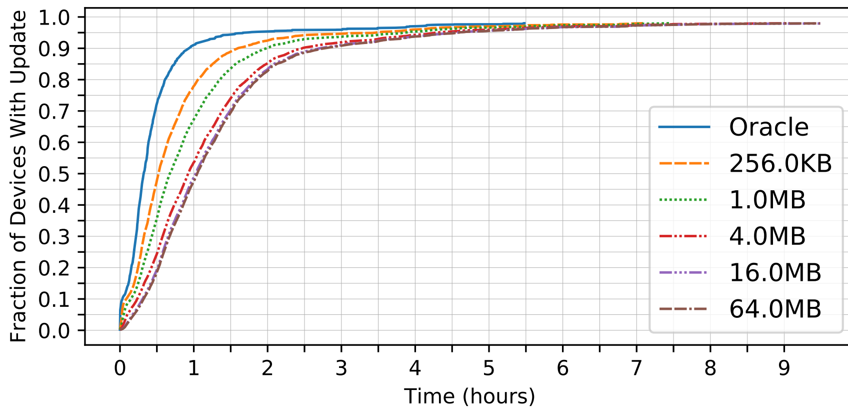


Figure 2.12: CDF of the dissemination duration from a random block's generation to its ultimate receipt at the most remote participants in the system.

2.6 Related work

Gossip protocols originated in the field of distributed databases [57] and has seen a resurgence coinciding with the proliferation of cloud computing [37, 70, 86, 125] and more recently within the blockchain community [38, 74, 88, 121]. Gossip protocols assume full network connectivity and can therefore not be directly applied in MANET environments.

While Vegvisir uses gossip if internet connectivity is available, Vegvisir relies only on opportunistic communication between devices when not. Such protocols have been shown to work well for reliable multicast in unreliable networks [51]. Bayou [99] is a peer-to-peer storage and communication system based on opportunistic communication. Unlike Vegvisir, Bayou does not have provisions to deal with Byzantine behavior. Also, it does not use CRDTs but relies instead on ad-hoc application-dependent merging protocols that requires applications to actively detect and resolve conflicts.

CRDTs [111] provide theoretical consistency guarantees. These data structures include versions of registers, counters, sets, graphs, and maps [77, 110], and can be combined and composed to create more sophisticated data structures such as key-value stores [108]. Applications include collaborative editing [84] and distributed databases [55].

COPS [82] is a causally consistent storage system intended for wide-area networks with high latencies. Like Bayou, it uses application-dependent merging strategies and has no defenses to Byzantine behavior.

To deal with Byzantine behavior, blockchains were first introduced in 2008 as part of the Bitcoin cryptocurrency system [92]. Since then, the blockchain field has seen explosive growth with many variants. However, most blockchain designs have a linear structure and rely on a proof-of-work consensus mechanism that requires solving a computationally expensive crypto puzzle. These characteristics make them poorly suited to MANET environment where computational power is limited and continuous network connectivity is not guaranteed.

Some blockchain variants use a DAG structure similar to Vegvisir. The GHOST protocol is a modification to the Bitcoin blockchain that uses a DAG structure to im-

prove security [115]. The intuition behind this modification is to enable a more robust method for decision making in fork selection. By keeping track of all forks, a node can choose a fork based on the heaviest-subtree-wins rule (the subtree with the largest number of blocks) as opposed to the longest-chain-wins rule, wasting less work and thus eliminating certain forms of attacks. While this approach reduces the computational requirements, it is still much too high for MANET devices. Also, the approach still requires continuous network connectivity.

The recently proposed SPECTRE [114] and MeshCash [44] blockchains also use a DAG structure along with a protocol to reach consensus in the case of conflicts. Both blockchains use Proof-of-Work and require continuous network connectivity, which eliminates them from consideration for our use cases.

Several blockchains have proposed using DAGs and *entanglement* to reduce the reliance on Proof-of-Work. Like Vegvisir, IOTA [23] is a cryptocurrency established to operate within the IoT space. In order to create a new transaction, devices are required to verify a certain number of prior transactions. Consequently, IOTA uses witnesses to determine when a transaction is valid within a system. IOTA uses Markov Chain Monte Carlo sampling to verify unapproved transactions [23]. Additionally, double spends are resolved through a consensus algorithm that determines which transactions to keep based on the number of descendant transactions. In order to accomplish this, IOTA requires continuous strong network connectivity. IOTA also uses a weak form of Proof-of-Work to reduce Sybil attacks.

Other DAG-based protocols such as HashGraph [38] Byteball [54], and Avalanche [127] do not rely on Proof-of-Work. The DAG structure in the aforementioned blockchains is not designed to provide partition tolerance like our case, but rather to exploit available parallelism for increased throughput of transactions by only order-

ing transactions that are dependent. As such, these blockchains expect strong network connectivity and are therefore unsuitable in our use cases.

2.7 Conclusion

In this chapter, we present Vegvisir, a middleware for a Byzantine mobile ad hoc network (MANETs). We designed a new distributed tamperproof data structure, the Vegvisir Block DAG, with opportunistic update propagation. The data structure leverages entanglement techniques from blockchains combined with keeping track of *witnesses* to make blocks tamperproof. Vegvisir uses provable detection of misbehavior to incentivize good behavior. Use of CRDTs combined with happens-before ordering allows applications concurrent and continuous access to consistent data structures. Vegvisir supports applications running under intermittent network connectivity with low power consumption and scales to at least hundreds of devices.

CHAPTER 3

BLOOMBOX: IMPROVING AVAILABILITY AND EFFICIENCY IN GEOGRAPHIC HASH TABLES

While shared cloud infrastructure has steadily increased in accessibility and reliability, there are still important scenarios in which such infrastructure is unavailable. For example, in hurricane-ravaged islands, it can take weeks for networking infrastructure to be restored, with no access to cloud-based messaging or storage. Rural country-side often lack reliable network access to start with. Yet in other scenarios, cloud infrastructure exists but is not usable for certain applications, either because access to it is too slow, too expensive, or subject to censure or unwanted monitoring.

An alternative is to store and share data on opportunistically formed infrastructure. Modern devices are capable of direct peer-to-peer communication. Prior work [80, 103] has proposed a way to store, or at least buffer, data reliably in mobile ad hoc networks. Using a hash function, a *Geographic Hashing Table* (GHT) maps each data object to a geographical location, and devices near this location store the object. Because of churn due to devices being mobile or failing, it is important that data be stored redundantly, most simply by storing each object in multiple locations. In order to maximize availability, it is important that the replicas are failure independent. This would suggest that replicas be placed far away from one another, perhaps using multiple hash functions that map each replica to a different pseudo-random location.

The replicas need to monitor one another through heartbeat messages so that if a replica gets lost, the other replicas can regenerate a new one. When heartbeat messages get lost, there may be significant bandwidth and energy waste because of unnecessary replica regeneration. Because mobile ad hoc networks (MANET) do not provide efficient reliable long-distance network connections, the prior work places all replicas on

devices near the location determined by the hash function to avoid such waste. But placing replicas close to one another makes them vulnerable to dependent failures where all devices in a small region fail because of a local event such as a fire.

To fix this undesirable trade-off, we introduce *BloomBox*. Instead of using heartbeats—a local mechanism to detect lost replicas—we propose a global solution based on *Bloom filters* [47]. Bloom filters provide space-efficient probabilistic membership testing. A single Bloom filter, opportunistically flooded between the devices, could track all object replicas. Information that is flooded through the MANET is more likely to arrive than information that is routed point-to-point, and thus this scheme has the potential to avoid unnecessary replica regeneration.

However, there are various challenges with this approach that need to be overcome. Bloom filters are append-only, so one challenge is how to eventually remove faulty replicas from the Bloom filter. Even though there exist variants of Bloom filters that support deleting elements [105], we would need a way to detect failures first, resulting in a chicken-and-egg problem. Another problem is that the parameters of a Bloom filter must be carefully chosen to minimize false positives. Moreover, these parameters are based on the number of objects in the system, which is variable. But if they are changing, how can devices agree on those parameters?

BloomBox addresses these challenges. First, BloomBox uses *epochs* during which the number of replicas of each object are re-counted. Second, BloomBox introduces a new variant of Bloom filters that we call a *Mergeable Bloom Filters*, allowing devices to temporarily disagree on parameters. Epochs should neither be too short nor too long, and we discuss and evaluate a way to come up with suitable epoch lengths. During data collection, we use techniques to approximate how many objects there are in the system so we can dynamically fine-tune the size of a Mergeable Bloom Filter.

Our simulations show that BloomBox can obtain the same level of data availability while reducing bandwidth consumed by regenerated blocks by an order of magnitude compared to a mechanism based on heartbeats. Furthermore, BloomBox can provide significantly better availability because replicas can be placed in geographically diverse locations.

The contributions of this chapter are summarized as follows:

- We propose using Bloom filters for detecting data loss in Geographic Hash Tables.
- We present a new variant of Bloom filter, a Mergeable Bloom Filter, that can be used for maintaining membership in scenarios where devices do not agree on system parameters.
- Our evaluation shows the advantage of using BloomBox over heartbeats in terms of both bandwidth usage and data availability.

We start with presenting the background and system model in Section 3.1. Section 3.2 presents the design of BloomBox and Mergeable Bloom Filters, as well as an analytical model to determine the appropriate configuration. We discuss implementation details of the protocol in a discrete time event simulation in Section 3.3. Section 3.4 first determines if our analytical model for configuration works and then compares BloomBox with heartbeat-based protocols from the perspective of both availability and bandwidth usage. We also discuss limitations of our work. Related work is presented in Section 3.5. Finally, Section 3.6 concludes.

3.1 Background and System Model

3.1.1 Geographic Routing

Messages can travel between devices that are not within range using store-and-forward routing. Under our assumptions, a given route between two devices in the network is not stable and can break at any given time due to mobility and failures. Therefore, a robust routing protocol is needed so that messages can be eventually delivered with high probability. There exist many approaches. One extreme approach is to flood each message throughout the MANET, optimizing the probability of delivery and minimizing latency, but at the expense of high network overhead. Other approaches try to balance network overhead, probability of delivery, and minimizing latency [48, 72, 75, 79].

We adopt *Greedy Perimeter Stateless Routing* (GPSR) [75] in our system. GPSR has two modes, *greedy forwarding* and *face routing*. In the greedy forwarding mode, data are forwarded to devices closer to the destination. When there is an obstacle in the middle, GPSR changes to face routing. Face routing routes data clockwise around the perimeter of the obstacle until it reaches a device closer to the destination and then changes back to greedy forwarding. Note that this protocol only works for 2D areas. If 3D routing is required, we could instead use a protocol such as MDT [79].

3.1.2 Geographic Hash Tables

A Geographic Hash Table (GHT) [103] is inspired by the Distributed Hash Table (DHT) [91]. In both cases, there is a dynamic collection of storage nodes, devices in the former and servers in the latter. For simplicity, we assume that data is stored in

units called *blocks*. A DHT hashes a block to a key and stores the block at the storage node whose identifier is numerically closest to the key. In a GHT, a block is hashed onto a geographic location and the block is stored on the device closest to that location. In both cases storage nodes are ephemeral, and for GHT there is also the issue that a device may move to a different geographic location.

Because of churn, a block cannot simply be stored on a single device—it has to be replicated. When a device disappears, the replicas of blocks stored on that device need to be regenerated on other devices. In DHTs, a block is hashed onto a server, and replicas of the block are placed at neighboring servers in a logical ring of servers. Prior work on GHTs [80, 103] takes a similar approach: each block is hashed to a *primary location*, and replicas are stored on the $f + 1$ devices physically nearest to the primary location, where f is the maximum number of failures that must be tolerated.

3.1.3 Failure Detection and Recovery

When a device leaves the MANET or fails, the objects on it must be re-replicated elsewhere. This requires that failures must be detected. Once detected, surviving replicas can regenerate the lost copies.

One approach is *heartbeat failure detection*. Let us call the devices that store replicas of the same block *peers*. With heartbeat failure detection, each device containing a replica of a block periodically sends heartbeat messages to its peers. If a device has not heard from some peers after some preset timeout value, it assumes that the replica stored on that device has failed. Optimizations of this approach include organizing the peers into a ring, so that devices only need to detect failures of neighbors on the ring.

An issue with heartbeat failure detection is that heartbeat messages have to be routed through the MANET. Because routing is not stable, a heartbeat message may get lost and lead to an erroneous failure detection, leading in turn to unnecessary replica regeneration. This is an important reason for why GHTs place replicas physically near to one another: heartbeats are easier to route and more likely to reach their destination. Unfortunately, the approach is not good for failure independence, because a single physical event can cause all devices in the same neighborhood to fail simultaneously.

3.2 System Design

In this section, we introduce BloomBox and describe the design of its failure detection and recovery mechanism. We start with a simplifying assumption that the number of objects is fixed so we can use ordinary Bloom Filters. We then drop this assumption and introduce the design of Mergeable Bloom Filters that allows the set of objects to change dynamically. We also present an analytical model to determine how best to set the various configuration parameters of BloomBox.

3.2.1 Failure Detection

We define a *failure event* to be the disappearance of the replica of a block. We need a way to detect failure events. As we saw in Section 3.1.3, heartbeat-based mechanisms require that replicas be placed close to one another, leading to an undesirable dependence among failure event probabilities. Instead, we propose using *Bloom filters* for failure detection.

A Bloom filter is a space-efficient data structure for probabilistic membership testing of a collection. A Bloom filter consists of a bitmap and a finite number of hash func-

tions. For each element in the collection, the bits corresponding to the output of each hash function in the bitmap are set. Bloom filters of the same size and using the same hash functions can be merged together by OR-ing them, representing the union of the collections. To test if a particular element is in the set, the hash functions for the element are computed and the corresponding bits in the Bloom filter are checked. Note that a Bloom filter is append-only: one can only add elements to a Bloom filter, not remove them. Clearing the bits corresponding to some element could inadvertently also clear bits of other elements, which in turn would result in false negatives.

Bloom filters can lead to false positives, but not false negatives. This is an important distinction from using heartbeat-based approaches, where false negatives can lead to significant bandwidth waste.

For simplicity, we will initially assume that the number of blocks in the MANET is static so all devices can agree on the size of the Bloom filter and the number of hashes. We can then use the Bloom filter to represent the replicas of all blocks stored in the MANET. We assign to each replica a unique identifier consisting of a pair `(DataID, CopyID)`. `DataID` is a 256-bit collision-resistant hash of the content of the block and `CopyID` is a small integer, different for each replica of the block. We assume a small fixed number of replicas per block. For example, to tolerate f failures, the number of replicas should be set to $f + 1$. In Section 3.2.3 we will discuss how to determine the number of replicas based on the desired maximum probability of data loss.

Because Bloom filters are append-only and suffer from false positives, we divide time into epochs (using the synchronized clock assumption) and start a new Bloom filter each epoch. By choosing different hash functions each epoch, we avoid the situation in which false positives persist across epochs—if the probability of false positives is low, the faulty replica will be detected and repaired with high likelihood in the next epoch.

The list of k hash functions $H_1^e, .., H_k^e$ for a Bloom filter in epoch e is defined by:

$$H_i^e(r) = H(i \| e \| r) \mod m$$

where r is a replica identifier, m is the bitmap size, $H(.)$ is a collision-resistant hash function, and || represents concatenation.

At the beginning of an epoch, each device creates a Bloom filter containing the replicas that it stores. When two devices are within range within the epoch, they exchange their Bloom filters and merge them by OR-ing them together. (The OR operation on Bloom filters corresponds to union.) This causes Bloom filters to disseminate throughout the MANET within an epoch in a gossip-style fashion, converging to a Bloom filter representing all replicas in the MANET.

3.2.2 Failure Recovery

At the end of an epoch, each device examines its Bloom filter to determine the set of replicas it needs to regenerate. To regenerate replicas, we consider two approaches.

In the first approach, each replica that determines the loss of a peer replica creates a copy of the replica and sends it using GPSR to the location determined by the hash function used by the GHT. This approach is simple, but can lead to multiple replicas trying to regenerate the same lost peer replica.

A better approach, inspired by DHTs, is to organize the peers that store the replicas of a block into a ring in order to prevent all peers from regenerating a failed replica, which would be inefficient. Assuming that there are r replicas of a data block with identifier `DataID`, then the successor of replica (`DataID`, i) is (`DataID`, $i+1 \mod r$).

If the successor replica is missing from the Bloom filter, then the device generates a new replica and sends it to its destination through the MANET.

We have to assume that it may take two epochs to detect a lost replica. After all, if a copy is lost during epoch, it may not be recorded in the Bloom filter of that epoch. Moreover, after detection it may take another epoch to regenerate the replica, so we will assume that it takes approximately three epochs to recover a faulty replica.

3.2.3 Configuration

The efficacy of our approach depends on the length of the epoch and the number of replicas. If the epoch length is chosen too short, Bloom filters do not have sufficient time to disseminate to all nodes and blocks would be unnecessarily regenerated, creating overhead. A long epoch increases the probability that all replicas of a block fail, causing the block to be unrecoverable.

To determine a good epoch value, we assume that the probability of a replica failing in one time unit is p and that this probability is independent of other replica failures. We use $P(t)$ to represent the probability of a replica failing within t time units:

$$P(t) = 1 - (1 - p)^t \approx tp \text{ (when p is small).}$$

Let r be the number of replicas. Let $Q(t)$ be the probability of all r replicas failing in t time units:

$$Q(t) = P(t)^r.$$

If we want the probability of all replicas failing in t time units be smaller than some small probability ϵ, then

$$Q(t) < \epsilon$$
$$\Rightarrow P(t)^r < \epsilon$$
$$\Rightarrow (tp)^r < \epsilon$$
$$\Rightarrow t < \frac{\epsilon^{\frac{1}{r}}}{p}$$

Because it takes three epochs to regenerate a lost replica, the length of an epoch should be less than $\epsilon^{\frac{1}{r}}/3p$. For the lower bound, observe that the epoch should be long enough for a Bloom filter to be disseminated across the MANET. Let $\mathrm{d}_{\mathrm{space}}$ denote the diameter of the network. Then the length of epoch should be at least $\mathrm{d}_{\mathrm{space}}/\mathrm{d}_{\mathrm{radio}}$ where the $\mathrm{d}_{\mathrm{radio}}$ is the diameter of the radio range (assuming the radio signal is omnidirectional).

Given these two expressions, we bound the length of an epoch using the following formula:

$$\frac{\mathrm{d}_{\mathrm{space}}}{\mathrm{d}_{\mathrm{radio}}} < e < \frac{\epsilon^{\frac{1}{r}}}{3p}$$

To obtain a valid epoch value, the upper bound should be larger than the lower bound. This gives us the formula to compute the minimum number of replicas needed in the system:

$$\frac{\epsilon^{\frac{1}{r}}}{3p} > \frac{\mathrm{d}_{\mathrm{space}}}{\mathrm{d}_{\mathrm{radio}}}$$
$$\Rightarrow \frac{1}{r}\ln\epsilon > \ln(3\frac{\mathrm{d}_{\mathrm{space}}}{\mathrm{d}_{\mathrm{radio}}}p)$$

Because $\ln\epsilon < 0$, we obtain the minimum number of replicas:

$$r > \frac{\ln\epsilon}{\ln(3\frac{\mathrm{d}_{\mathrm{space}}}{\mathrm{d}_{\mathrm{radio}}}p)}$$

3.2.4 Mergeable Bloom Filter

So far we have tacitly assumed that all devices use the same Bloom filter size. However, the size of a Bloom filter should depend on the number of objects stored in it and the desired maximum probability of false positives. As devices cannot exactly determine the number of objects stored in the MANET, they may end up using Bloom filters of different sizes.

At the start of each epoch e, each device d estimates the number of blocks in the MANET (see below) and generates a Bloom filter representing the blocks that it stores. When two devices are within range, they reconcile the Bloom filters that they store, OR-ing them together only if they have the same size. So each device ends up storing a set of Bloom filters of different sizes. We call this set a *Mergeable Bloom Filter* (MBF). A device can check if a replica is stored in the network by checking for membership in each of the Bloom filters in the Mergeable Bloom Filter that it stores.

Formally, we first define an aggregated Bloom filter of a set of devices D using the same Bloom filter size in an epoch e as follows:

$$BF_D^e = \bigvee_{d \in D} BF_d^e$$

where Bloom filter BF_d^e represents the set of blocks stored on device d at the beginning of epoch e.

Let $|BF|$ denote the bitmap size of Bloom filter BF. A Mergeable Bloom Filter $MBF = \{BF_1, ..BF_n\}$ is a set of one or more Bloom filters of different sizes (and therefore representing disjoint sets of devices). Two mergeable Bloom filters MBF_1 and MBF_2 can be merged by

$$
\begin{aligned}
MBF_1 \uplus MBF_2 \triangleq \\
&\{BF_1 \vee BF_2 \mid BF_1 \in MBF_1 \\
&\quad \wedge BF_2 \in MBF_2 \wedge |BF_1| = |BF_2|\} \\
&\cup \{BF_1 \mid BF_1 \in MBF_1 \\
&\quad \wedge \forall BF_2 \in MBF_2 : |BF_1| \neq |BF_2|\} \\
&\cup \{BF_2 \mid BF_2 \in MBF_2 \\
&\quad \wedge \forall BF_1 \in MBF_1 : |BF_2| \neq |BF_1|\}
\end{aligned}
$$

3.2.5 MBF Size Estimation

Each device estimates the number of replicas stored in the MANET at the beginning of each epoch to determine the optimal Bloom filter size. If the fraction of clear bits in a Bloom filter with m bits and k hash functions is P_{clear}, then the number of elements n in the collection that the Bloom filter represents can be estimated [47] using the formula

$$n = \frac{-m \times \log P_{clear}}{k}.$$

Because devices only merge Bloom filters of the same size, Bloom filters in a Mergeable Bloom Filter represents disjoint sets of replicas. Thus the total number of replicas in a Mergeable Bloom Filter can be estimated by taking the sum of the estimates for the Bloom filters stored in the Mergeable Bloom Filter.

In the worst case, each device could estimate a different size, and then none of the Bloom filters in the Mergeable Bloom Filter can be OR-ed together. This could result in a very large Mergeable Bloom Filter: if there are d devices, each using a size of ap-

proximately m bits, then the size of the Mergeable Bloom Filter would be approximately $d \cdot m$.

Ideally we would like the size of the Mergeable Bloom Filter to be independent of the number of devices. To accomplish this, each device rounds up its estimate of the number of replicas to the nearest power of 2. This is conservative, as increasing the size of a Bloom filter results in a reduced probability of false positives. If M is the largest Bloom filter generated by the devices, then the size of the resulting Mergeable Bloom Filter is bounded by $2M$.

3.3 Implementation

We have implemented a GHT in a discrete time event simulator. There are two versions. The first uses BloomBox for replica failure detection. The second, Heartbeat, uses the original failure detection mechanism proposed for GHTs. In this section, we describe various implementation details.

3.3.1 Reconciliation

Each simulation consists of a sequence of rounds. In a round, two devices that are within radio range reconcile their Mergeable Bloom Filters as described below. First, devices broadcast their Mergeable Bloom Filters. On receipt, Mergeable Bloom Filters can be OR-ed together as described in Section 3.2. Should a device be able to reconcile with multiple peer devices in the same round, then these reconciliations happen in parallel rather than in series. This means that if device X is in range of device Y and device Y is in range of device Z but devices X and Z are not within range, then no data can travel

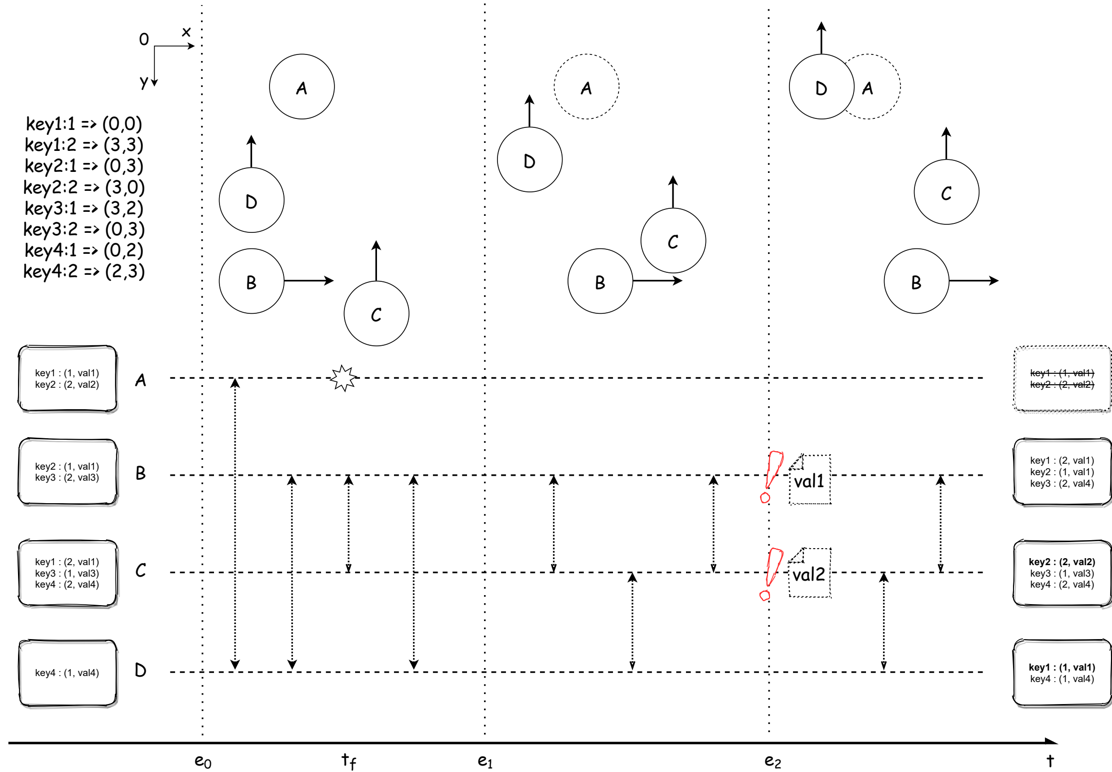


Figure 3.1: A time-space diagram of BloomBox recovery, with time going left to right. There are four devices in the diagram, named A through D. The dotted double-headed arrows represent reconciliation between two devices. The top left corner contains a list of hash mappings from `key:replicaNumber` to a location `(x,y)` on the map. The origin of the map is at the top left corner and the y-axis grows down vertically. The locations and moving directions of the devices in each epoch are shown above the time-space diagram. The rounded boxes next to the device names display data that each device has stored. For example, for device A at time e_0, `key1: (1, val1)` means that device A stores replica 1 of the object with key `key1`.

from X to Z in this round. However, if the same situation persists for two rounds, then X and Z will end up having their data fully reconciled.

For the Heartbeat protocol, a device sends a heartbeat message for each object that it holds to each of the other replicas in each round. Each of these heartbeat messages needs to be routed to its final destination using GPSR.

At the end of an epoch, devices determine which replicas need to be regenerated. In the case of BloomBox, this is based on checking for membership, while in the case of

Heartbeat, it is based on not having received a heartbeat message in the last epoch. To regenerate a replica, a device sends a copy of the local replica to the intended location of the regenerated replica based on the geographic hash function. These regenerated blocks are also exchanged when devices are communicating as part of the GHT protocol—using GPSR for routing. A device only reconciles blocks with at most one neighbor in a round.

3.3.2 Example of Detection & Recovery

Figure 3.1 contains a space-time diagram that illustrates how the system recovers lost block replicas after a device failure. There are four devices in the region. Starting from the left (epoch e_0), device A sends its Mergeable Bloom Filter to its neighboring device D. Then, device A crashes at time t_f. In the same epoch, devices B, C, and D also exchange their Mergeable Bloom Filters. B and D exchange their Mergeable Bloom Filters twice because B got updates from C after reconciling with D. Within the first epoch e_0, no device detects that A has crashed. In epoch e_1, devices B, C, and D generate their Mergeable Bloom Filters and reconcile normally. However, since device A had crashed, no Mergeable Bloom Filter from A for `key1`'s replica 1 and `key2`'s replica 2 has been created.

Failure detection occurs at the beginning of epoch e_2 when devices B and C, which had replicas for `key1` and `key2` respectively, found that `key1-1` (replica 1 for the object with `key1`) and `key2-2` (replica 2 for the object with `key2`) were not present in the Mergeable Bloom Filter reconciled in the prior epoch e_1. In response, they generate new replicas and forward them to their assigned locations determined by the GHT's

hash function (using GPSR). In particular, replica `key1-1` is recreated on device D and replica `key2-2` is recreated on device C.

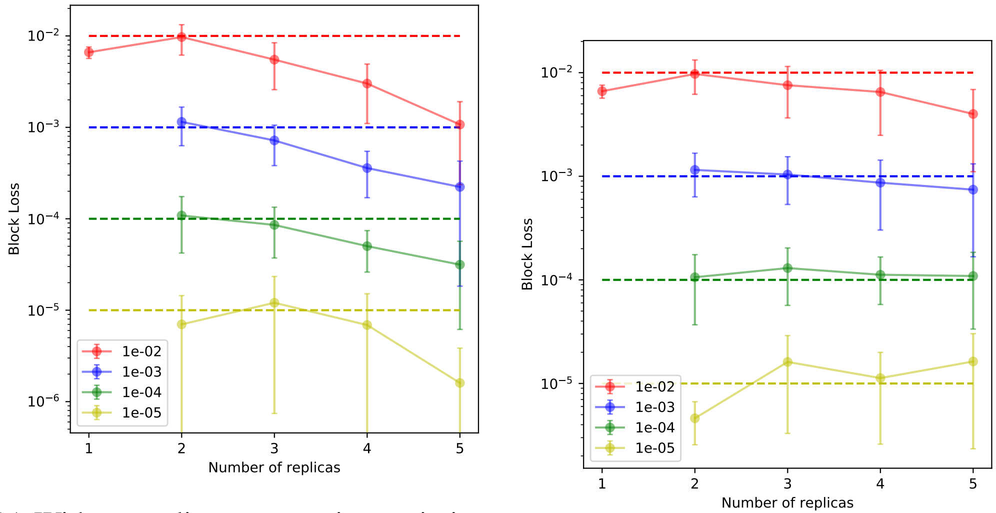
3.3.3 Optimizations

As hashing is pseudo-random, it is possible in theory that two replicas of the same object hash to approximately the same location. In order to optimize failure tolerance, we avoid devices storing multiple replicas of the same object. In particular, we use a tie-breaking rule that causes replicas that would otherwise end up on the same device to be stored on different devices. In addition, at the beginning of each reconciliation, devices exchange the hashes of their Mergeable Bloom Filters first and only exchange Mergeable Bloom Filters if the hashes do not match.

An optimization that we have not yet implemented is the following. Instead of sending an entire Mergeable Bloom Filter from one device to another during reconciliation, we would like to send only those bits that have changed. This would require a mechanism to determine which bits have changed. One option is that devices can keep track of hashes of old Mergeable Bloom Filter versions. After the initial handshake, where devices exchange their hashes, each device can determine if it knows the version of the Mergeable Bloom Filter that the other device has, and if so only send the bits that have changed instead of the entire Mergeable Bloom Filter.

3.4 Evaluation

To evaluate the BloomBox protocol, we built a prototype and evaluated it in a discrete time event simulator. The evaluation has two goals:



(a) Without replica regeneration optimization

(b) With replica regeneration optimization

Figure 3.2: Block availability for computed epoch lengths varying numbers of replicas and ϵ.

Parameter	**value**
Width	100
Height	100
# of Blocks	30,000
Duration(rounds)	30,000
#Replicas	3
Initial #Devices	100
Max Speed	5
Min Speed	0
Max Pausing Duration	10

Table 3.1: Parameter settings for simulations

- Does the analytical epoch model derived in Section 3.2.3 result in good epoch lengths?
- How does BloomBox compare to Heartbeat with respect to block availability and bandwidth usage?

First, we introduce the simulation model. Next, we demonstrate that our analytical epoch model is able to provide us with suitable epoch lengths. Then we show that

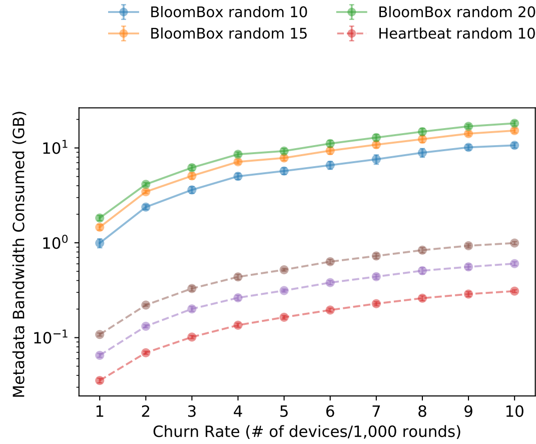
BloomBox can significantly outperform Heartbeat in terms of bandwidth efficiency while provides better availability. Finally, we discuss limitations of BloomBox.

3.4.1 Simulation Model and Metrics

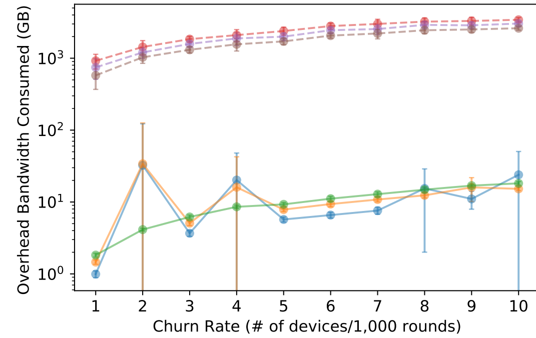
A simulation runs in a series of rounds. One round is one unit of simulation time. Each simulation runs for 30,000 rounds. The simulation area is a square map, the width and height of which are 100 units of length. We used radio ranges to specify how far devices can communicate directly if they are within range. The amount of data that devices within radio range can exchange within a round is unbounded. There are no obstacles on the map—in practice the protocols that we use are able to route around obstacles. In particular, Mergeable Bloom Filters are flooded.

We use a variant of the random waypoint mobility model [72]. Initially, we place 100 devices on the map uniformly at random. Each device selects a random destination uniformly and a speed of v length units per round, where $0 \leq v \leq 5$. When a device reaches its destination, it pauses for a random duration t_{stop} where $0 \leq t_{stop} \leq 10$, then it repeats this procedure until the end of the simulation.

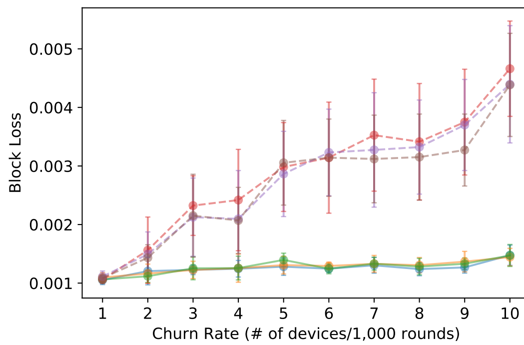
A simulation experiment has three phases of equal length (i.e., about 10,000 time units). In the first phase, each device generates a new block in each round with probability 0.03. Block generation stops in the second phase (after generating approximately 30,000 blocks total), but devices keep moving around, reconciling their Mergeable Bloom Filters and moving block replicas to their destinations. In the third phase we introduce device churn and measure bandwidth usage. At the end of the experiment we can determine what percentage of blocks is still available. A block is available if at least



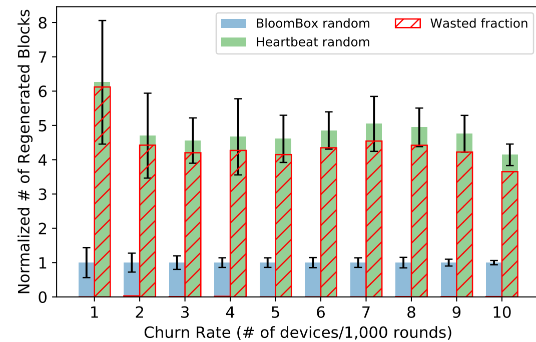
(a) Bandwidth consumed by metadata for each protocol (log scale)



(b) Total bandwidth consumed by overheads of regenerating wasted blocks and protocol metadata (log scale)



(c) Block loss for each protocol

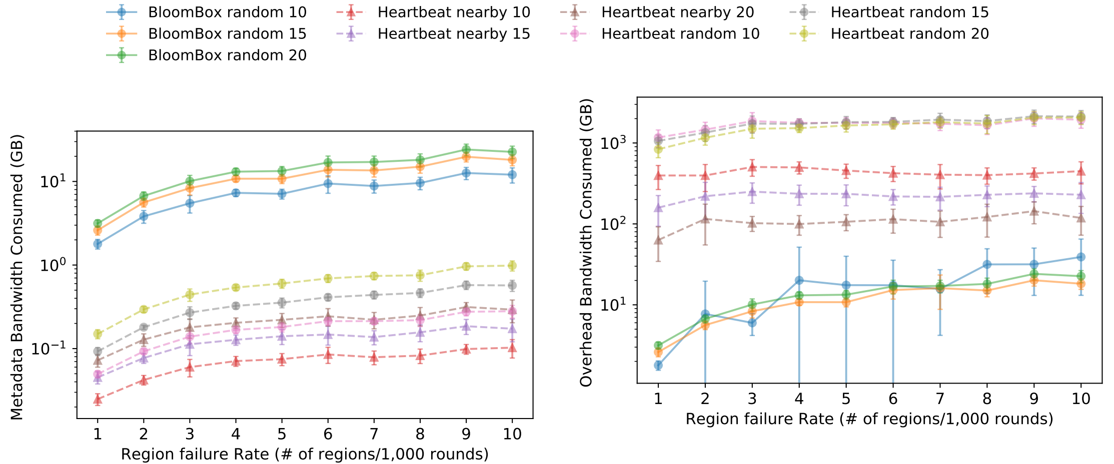


(d) Normalized number of regenerated blocks relative to BloomBox's and the fraction of wasted regenerated blocks

Figure 3.3: Simulation results for independent-failures. The legend for the first three figures is at the top. We ran simulations with three radio ranges, 10, 15, and 20. In all experiments, replicas were placed pseudo-randomly on the map.

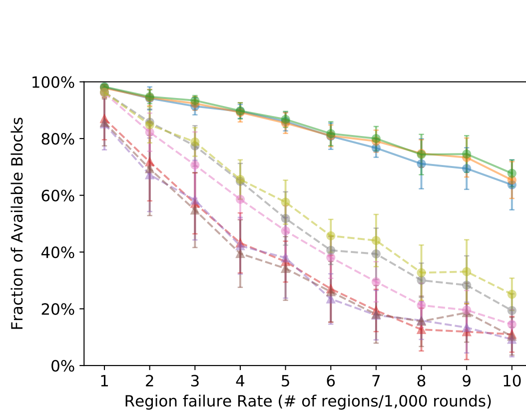
one correct device stores a replica of that block. Table 3.1 contains a summary of the fixed parameters used in the simulations.

In our experiments we vary the *churn rate*, which describes the average number of devices entering or leaving the simulation area per simulation round. We experiment with both independent failures and dependent failures. When using independent failures, devices fail uniformly at random. When using dependent failures, a failure event affects all devices within a 20x20 area, covering 4% of the map, selected uniformly at random.

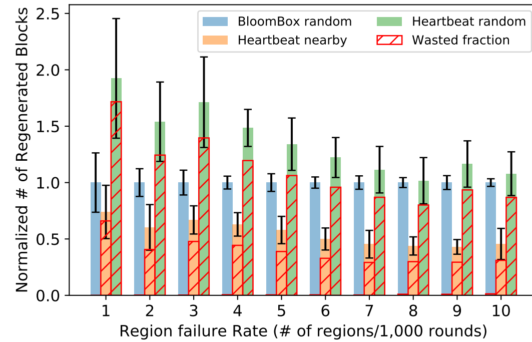


(a) Bandwidth consumed by metadata for each protocol (log scale)

(b) Total bandwidth consumed by overheads of regenerating wasted blocks and protocol metadata (log scale)



(c) Block availability at the end of simulations



(d) Normalized number of regenerated blocks relative to BloomBox's and the fraction of wasted regenerated blocks

Figure 3.4: Simulation results for dependent-failures for both random replica placement and, in the case of Heartbeat, nearby replica placement. Results for BloomBox are in solid lines and results for Heartbeat are in dashed lines. A circle marker indicates random placement and a triangle marker represents nearby placement.

We focus on four metrics for measuring the performance of BloomBox and Heartbeat. The first metric, *block loss*, describes the fraction of unavailable distinct blocks in every three epochs. We use this metric to test our model in Section 3.2.3. A related metric is *block availability*, which tells us the fraction of distinct blocks with at least one replica on a correct device at the end of a simulation. The second metric, *metadata overhead*, is the bandwidth consumed by metadata, which are Heartbeat messages for

Heartbeat and Mergeable Bloom Filters for BloomBox respectively. The third metric, *regenerated blocks*, is the number of blocks regenerated by both protocols to recover lost replicas, including blocks that are regenerated unnecessarily due to false negatives. The last metric, *total overhead*, is the bandwidth consumed by unnecessarily regenerated blocks and metadata.

For each metric, we report the average of our measurements as well as an error bar displaying one standard deviation (unless it is negligible).

3.4.2 Availability versus Epoch Length

Section 3.2.3 presents an analytical model to find the optimal epoch length for a given number of replicas if we want the probability of unavailability be less than a given probability. A longer epoch length increases this probability, but a shorter epoch length leads to a higher Mergeable Bloom Filter refresh rate and more false positives due to Mergeable Bloom Filters not fully reconciling, and thus more unnecessary regeneration of replicas causing higher bandwidth usage. One should therefore choose the longest possible epoch length that leads to a desired maximum failure probability ϵ. Note that it takes approximately 3 epochs from the time that a device holding a replica fails until the failure has been detected and a new replica reaches another device at its intended location. ϵ, then, is the average percentage of the number of blocks that are lost in every 3 epochs.

We ran simulations for different numbers of replicas and different values of ϵ to verify the model. In particular, we want to show that, by using the epoch lengths computed by our model, the system achieves block availability similar to the target level in all experiments.

Instead of using churn rates, we use device failure probabilities to match the model in Section 3.2.3. At each round, the probability that a device can fail is set to 0.0001 and the arrival rate for a new device is 0.01 devices per round so that the average number of active devices throughout the simulation is around 100.

We run two sets of experiments. In the first set, every device that detects a lost replica regenerates it. This corresponds to the model but can be wasteful. In the second set, we use the optimization where the devices that hold replicas are organized into a directed ring, and a device only regenerates a replica if it is its successor replica that has been lost.

Figure 3.2a shows block availability for various experiments without the optimization. The x-axis is the number of replicas from 1 to 5 and the y-axis is the percentage of the lost blocks on a log scale. The colored dashed lines represent the target failure percentages of blocks at every 3 epochs. Different colors represent different values of ϵ. The solid lines represent the simulation results running with the calculated largest epoch length that can support the respective target failure percentages. We ran 10 simulations with different random seeds for each data point.

As can be seen, all solid lines are mostly at or slightly below the corresponding dashed lines. There are, however, some data points where the solid lines are slightly above the dashed lines. This is expected because we use the longest possible epoch and the model considers averages rather than worst case scenarios.

Figure 3.2b presents the results of the experiments with the optimization enabled. It demonstrates that the optimization still allows BloomBox to approach the desired target block availabilities quite accurately.

3.4.3 Comparing BloomBox and Heartbeat

Figure 3.3 and Figure 3.4 show the simulation results for independent failures and dependent failures respectively. In Figure 3.3, the x-axis represents the churn rate of devices under independent failures. To investigate the effect of different communication radio ranges, we ran experiments with radio ranges 10, 15, and 20. The size of a block in all simulations described below is 1 MB. The Time-to-live (TTL) for a Heartbeat message is set to one epoch. The target failure probability is 0.001, and the optimization where replica only regenerates its next replica on a ring is in effect.

In the independent-failure scenario, we investigate two protocols, BloomBox random and Heartbeat random. In both, replicas are placed randomly across the map. Figure 3.3a shows the bandwidth consumed by all Heartbeat messages, 32 bytes each, and by all Mergeable Bloom Filters. The bandwidth of metadata for BloomBox is larger than Heartbeat. However, when we consider the total overhead (Figure 3.3b) including both metadata and blocks that are regenerated by mistake due to false negatives, BloomBox saves orders of magnitude of bandwidth compared to Heartbeat. This can be explained by Figure 3.3d, which shows the normalized number of regenerated blocks for each protocol relative to BloomBox's. Heartbeat suffers considerably from false negatives. The wasted fraction of regenerated blocks is high for Heartbeat but almost zero for BloomBox. At the same time, BloomBox provides better availability than Heartbeat as shown in Figure 3.3c. Figure 3.3b also shows that, with a small range, overhead is less predictable than with a larger range—the reason is that with a small range there are more false negatives and therefore more block regenerations. This is particularly prominent with a low churn rate—with a high churn rate the probability of false negatives goes down.

In Figure 3.4, the x-axis represents the region failure rate under dependent failures. When a region fails, all blocks on devices within the randomly selected 20 by 20 region are lost. Besides random replica placement, we added experiments that show the performance of Heartbeat when replicas are placed around a single mapped location, which is the usual mode of Heartbeat protocol deployment. When a Heartbeat message reaches its destination, it is face-routed around the destination to increase the probability of getting to all replicas.

Similar to the experiments with independent failures, in the dependent failure case, BloomBox also consumes more bandwidth for its metadata. But again this is compensated by only a small fraction of wasted regenerated blocks (Figure 3.4a). As a result, BloomBox saves orders of magnitude of bandwidth over both Heartbeat random and Heartbeat nearby (Figure 3.4b). Since the model in Section 3.2.3 only considers random placement for replicas, instead of showing the *block loss*, we plot block availability here for dependent failures. As shown in Figure 3.4c, BloomBox maintains the highest availability of blocks. Heartbeat nearby is the worst because a region failure can erase all replicas of a block. Figure 3.4d shows the normalized number of regenerated blocks for each protocol relative to BloomBox's.

3.4.4 Limitations

In order to achieve good scalability, our simulations made various simplifying assumptions. For example, we used a round model in which two devices within radio range can completely reconcile. In practice, devices are within range at arbitrary times and may or may not be able to fully reconcile their data. Another assumption is that each device has an unlimited amount of storage. In practice, devices may run out of storage, in which

case they may not be able to store all the replicas mapped to their location. We have used a fixed block size. It is important that the block size be large enough so that the overhead of BloomBox metadata is not overwhelming. Also, our dependent failure model is simplistic. In our simulations, we assumed that device clocks were fully synchronized. In practice, it is fairly easy to synchronize the clocks of devices from ambiant sources such as cell phone carriers and GPS. Even without those, our protocol is not very sensitive to small skews between clocks. To route messages in our evaluation, we have used GPSR. Different routing protocols may lead to different efficacies.

In spite of these limitations, we believe that the simulations are accurate enough to demonstrate that the Mergeable Bloom Filter approach will be superior to Heartbeat in that it can significantly reduce bandwidth usage by reducing false negatives while providing better availability in the face of both independent and dependent failure events.

3.5 Related Work

3.5.1 Geographic Hash Tables

Distributed Hash Tables [87, 91, 102, 107, 131] work well if devices have network addresses that can be efficiently accessed from anywhere. This is not the case in mobile ad hoc networks. As devices move around, it becomes difficult to maintain the ring structure and keep track of where data is located. [103] introduced Geographic Hash Tables (GHT) to address this problem. GHTs are inspired by DHTs, but instead of mapping a key of an object onto a virtual ring space, in a GHT a key is mapped onto a physical geographic location. In the last two decades, GHTs have been well studied

by multiple research groups in sensor networks, delay tolerant networks, and mobile ad hoc networks [61, 80, 103].

An important difference between DHTs and GHTs is that with GHTs the location of data storage is no longer related to the identifier of a device. As a device moves around, changing its geographic location, it ends up storing different data items as a result. Similarly, data is not replicated on devices that are nearby in the logical ring space but replicated on devices that are geographically nearby.

In a GHT, the *home node* of a data object is defined to be the device that is geographically closest to the location where the data object is mapped. A home node periodically sends *refresh packets* to nearby devices that replica the data. If such a device does not receive any refresh packets before a preset timeout, or if the location of the sending device is now farther from the home location, then the device considers itself a candidate for the home node and starts generating refresh packets itself. GHT also proposes to use Structured Replication (SR), which divides the map into 4^d regions containing one primary and $4^d - 1$ mirrors where d is the depth of the hierarchical structure. SR can help with load balancing but the paper does not discuss a failure recovery mechanism for mirrors.

Replicating data on physically close devices suffers from dependent failures. MHT [80] is a GHT variant that replicates objects onto both physically local peers and remote peers. To maintain availability, replicas exchange heartbeat messages to detect data loss. CHT [61] divides the map of a geographic region into multiple levels of layers of cells. Data items in a CHT are mapped into a particular cell. Devices within a cell are responsible to store all data items mapped to that cell. To detect a loss of a replica, CHT uses a similar protocol as Tapestry [131].

3.5.2 Mobile Ad Hoc Routing Protocols

The BloomBox design relies on prior work in Mobile Ad Hoc Networks (MANET), in particular those where packets are routed to a particular geographic destination rather than a logical network address. GPSR [75] routes packets in two modes, *greedy forwarding* and *face routing*. In greedy forwarding mode, a packet is routed to a device that is closer to the destination than the device that holds the packet. If no such device is found because of obstacles, GPSR switches to face routing mode. In face routing mode, packets are routed to nearby devices to the right of the moving direction of the current device until a peer device that is closer to the destination than the device that started face routing appears. Then GPSR switches back to greedy forwarding mode. MDT [79] uses Delaunay Triangulation to triangulate the local view of the network graph. Then, a routing table is built based on the triangulated graph and packets are routed accordingly.

Our work is also related to routing protocols for Delay Tolerant Networks. For example, epidemic routing [120] floods packets into the network to optimize the probability of reaching the destination. Spray and Wait [117] floods a predefined number of packets into the network. A device that receives the packets forwards half of the available packets when it reconciles with other devices. A device holds a packet until either it communicates directly with the destination device of the packet or until a timeout is reached.

3.5.3 Bloom Filters

A Bloom Filter [47] is a probabilistic data structure for membership testing. It uses on average a small number of bits per element, but this comes at the cost of having a probability of false positives when checking for membership. Bloom filters have been

used in many applications, such as web caching [62], network routing [58, 116] and storage [78].

Counting Bloom Filters [62] (also known as *count-min sketches*) generalizes Bloom filters. Whereas in a standard Bloom filter each element is mapped onto k bits using k different hash functions, in Counting Bloom Filters elements are mapped onto k counters that are incremented. As a result, Counting Bloom Filters allow deletions. However, if a counter overflows or an element is removed that was not added to the Counting Bloom Filter, it can lead to false negatives.

A *Deletable Bloom Filter* [105, 118] achieves safe element deletion by recording which bits conflict and which bits do not. A Deletable Bloom filter must reserve extra space to record whether a region of the bitmap has conflicting bits. The size of thise region is adjustable and affects the success probability of an element being deleted.

Dynamic Bloom Filters [66] aim to remove the a priori chosen maximum size of a plain Bloom filter. A Dynamic Bloom filter maintains an ordered list of plain or counting Bloom filters as well as a counter for each Bloom filter that contains the number of elements in that Bloom filter. An element is inserted in the first Bloom filter that has an element counter smaller than a preset value. To test membership, all Bloom filters are tested and the query returns true if the element is present in any of the Bloom filters.

The goal of Scalable Bloom Filters [33] is to scale up to an unbounded number of elements by dynamically adding new Bloom filters when the existing ones fill up. The size of the newly added plain Bloom filters are carefully chosen to match a desired false positive rate over the entire series of Bloom filters. Other variants on Bloom filters include [101, 106].

Note that Mergeable Bloom Filter, Dynamic Bloom Filters, and Scalable Bloom Filters all contain more than one Bloom filter. There are important differences, however. In a Mergeable Bloom Filter, the different Bloom filters are guaranteed to be disjoint. Two different Mergeable Bloom Filters can be merged by OR-ing together Bloom filters of the same size. Finally, with Mergeable Bloom Filters, each device selects a Bloom filter size based on measurements in the prior epoch.

3.6 Conclusion

In this chapter, we present BloomBox, a new protocol for detecting and recovering lost data replicas in Geographic Hash Tables deployed in a mobile ad hoc network. The protocol uses a novel data structure called Mergeable Bloom Filters. Our simulations demonstrate that BloomBox has much less bandwidth overhead compared to existing protocols based on heartbeats while maintaining better data availability under an independent failure model. Moreover, in a dependent failure model, BloomBox provides superior availability by being able to spread replicas widely across the deployment area.

CHAPTER 4

DISAGGREGATING APPLICATIONS USING UNISERVICES

The pursuit of better hardware resource utilization is driving the shift towards disaggregated data centers (see Figure 4.1). Instead of relying on one-size-fits-all servers containing a mix of CPUs, memory, storage, NICs, GPUs, FPGAs, etc., disaggregated data centers employ racks filled with specialized servers connected through fast interconnects like RDMA [104], Gen-Z [16], or CXL [10]. These racks include separate pools for CPUs, memories, NICs, disks, GPUs, and more, effectively creating a massive computer with components connected via high-speed networks. This architecture offers several key advantages over the traditional server-centric cloud infrastructure. First, it enhances overall resource efficiency by reducing physical boundaries that cause segmentation in resource allocation. Second, it simplifies hardware maintenance and upgrades, as each component can now be managed individually. Lastly, it enables the seamless integration of new hardware types by simply connecting server blades to the network. As a result, many industry leaders, including Amazon, Facebook, and Snowflake, have adopted resource disaggregation.

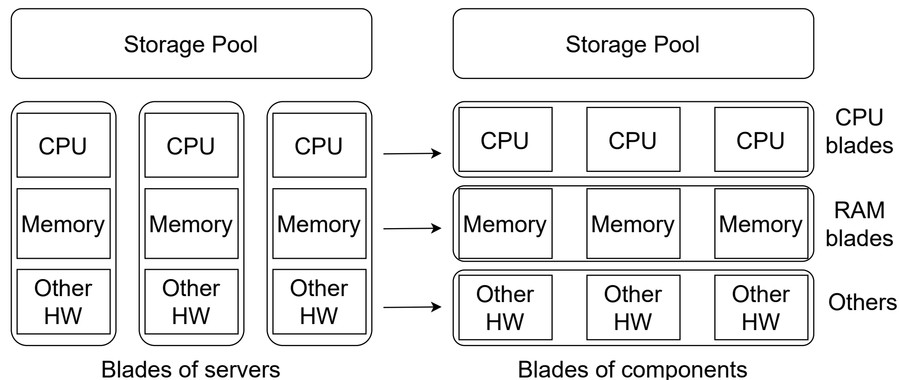


Figure 4.1: Server-centric v.s. disaggregated architecture.

However, adopting this new architecture for all its benefits also presents challenges. First, there is a need for managing disaggregated hardware and providing convenient abstractions to applications. Existing kernels are designed for monolithic servers and assume local access to other hardware, such as memory and storage. This assumption can lead to significant performance degradation due to high network overhead across servers. LegoOS [109] addresses this issue by using a *Split Kernel* design where each resource server runs part of the kernel. However, it provides a logical server interface for backward compatibility to hide the underlying disaggregation. Recent studies [130] show that concealing disaggregation can cause considerable performance degradation when porting unmodified applications to a disaggregated architecture, even with LegoOS.

Second, the current programming paradigms for building cloud applications, such as microservices [50] and serverless [42], are not easily scalable on disaggregated architectures. Their components are divided by logical rather than physical boundaries. For example, AWS Lambda [5] requires a fixed ratio between the number of cores and the size of allocated memory for a function [6], 1769 MB/core. As a result, scaling out requires obtaining resources together instead of just scaling the resource that has become the bottleneck, as disaggregated architecture is designed for.

Thirdly, in addition to the fixed ratio among different resources, there are also fixed combinations of the types of resources one can utilize. For instance, to take advantage of a powerful GPU like the A100, a customer must also allocate high-end CPUs and expensive memory. This is because the current server-centric datacenters only offer a limited selection of resource combinations [17].

Therefore, we need a generic programming paradigm for this new architecture, allowing developers to fully leverage the advantages of hardware disaggregation.

In this chapter, we argue that creating logical servers, as seen in works like LegoOS and GiantVM [129], may not be the most effective way to utilize disaggregated resources. Similar to previous research [35], we believe that hardware resource disaggregation should be exposed to applications rather than hidden from them. However, we go further by proposing that the applications themselves should also be disaggregated.

Drawing inspiration from the actor model [69], we propose building applications as a collection of ***uniservices***. A uniservice primarily uses a single type of resource and consists of numerous actors. An actor, a unit of computation in the actor model, can only communicate through messages, and its private state can only be modified by itself. This design aligns well with the disaggregated architecture, as different hardware components are also connected by a fast network and can only communicate through messages. Moreover, an actor can be treated as a specialized computation tailored for a specific resource type, such as a CPU, GPU, memory, storage, and networking servers. This allows actors to be grouped by the resource they most demand.

Disaggregation comes with its own challenges. The operational and communication overhead introduced by incorporating network-attached small components increases the system's complexity. To reduce this complexity, we use a shared log as a central bus for different application components. Applications can utilize this log to store the code and state of actors as well as to facilitate message publishing and subscribing. Consequently, the log preserves the application's state while providing sequential semantics for state changes and enabling parallel computation across various actors.

In the rest of the chapter, we will first discuss the motivation behind uniservices; then, we will present the uniservice design and its current implementation in detail; finally, a brief evaluation of a video processing example will be provided.

4.1 Motivation

Many applications currently deployed on the cloud are designed for a server-centric architecture that is several decades old. This approach has some inherent limitations.

First, because of the physical boundaries between monolithic servers, it is difficult to achieve precise bin-packing for different resources. As a result, many applications and servers are over-provisioned to ensure an acceptable level of performance, which leads to a low utilization of certain resources and energy waste.

Second, the server-centric architecture can limit performance, particularly on memory-intensive workloads such as data processing. Since the memory size on a single motherboard is often insufficient to hold the entire dataset, the application is forced to swap data between disks and memory, resulting in performance degradation.

Third, the server-centric architecture has poor support for hardware reconfiguration. Many datacenter applications, at the time of writing this thesis, rely on multiple specialized hardware resources, such as TPUs, FPGAs, and NVMe, in addition to CPUs. However, in a monolithic server architecture, integrating these new hardware resources is challenging because the motherboard tightly couples resources together. Moreover, adding or removing resources from a monolithic server can be difficult and may even require replacing the entire motherboard.

Finally, fate-sharing of all components can result in wasted resources on a monolithic server. If a hardware component on a server fails, all of the other functional components become unusable until the malfunctioning part is fixed. This can lead to unnecessary downtime and resource wastage.

To address these issues, data centers are increasingly adopting a disaggregated datacenter (DDC) design [21, 49, 76, 81, 93], where the same type of hardware resources are grouped together on a single server. However, this new architecture also brings new challenges for both operating systems and application developers in three dimensions.

4.1.1 Disaggregated Hardware Management

Existing operating system kernels are designed for monolithic servers and assume local access to most hardware resources, such as memory, disks, and GPUs. However, when components are disaggregated and connected only by the network, normal operations, such as reading from a memory address, can trigger high overhead due to the data being stored remotely instead of in the local cache on the CPU server.

4.1.2 Programmability

The two current trending approaches for building cloud applications are the microservice paradigm and the serverless paradigm. However, neither of these paradigms is well-suited for the disaggregated architecture.

Microservices break an application down into small logical components, which can span multiple types of resources. While it may be possible to deploy applications in this paradigm in a disaggregated environment, it cannot fully leverage the advantages of hardware disaggregation. Microservices assume that they are running on a monolithic server, so the underlying hypervisors or operating systems have to provide a ”logical server” abstraction that hides the underlying truth of disaggregation. This abstraction prevents developers from explicitly optimizing their code for better performance, such

as easing network overhead between memory servers and CPU servers by explicitly prefetching data in batch instead of relying on memory management in the OS. Additionally, microservices do not scale well along physical boundaries. Unlike monolithic servers, individual resources can now be scaled and managed independently in a disaggregated architecture, but to scale a microservice, all resources used by that microservice need to be scaled together.

Serverless takes application decomposition even further by breaking down applications into functions. However, current serverless functions have many restrictions. First, they can only be deployed on CPUs, so they cannot leverage special hardware like GPUs or FPGAs. Second, serverless functions like AWS Lambda and Azure Function only provide a fixed ratio between the number of CPU cores and the size of memory for a single function, which can lead to low resource utilization and high costs [45], particularly when the real bottleneck is determined by real-time inputs that are difficult to predict. Finally, similar to microservices, serverless functions assume a monolithic server architecture, which means they cannot be optimized for better performance with the underlying architecture. Even with far memory extension, page faults that trigger remote fetching can hurt performance badly.

4.1.3 Application Performance

The performance of applications can either benefit from or harmed by resource disaggregation. Applications whose working set cannot fit in local memory but can fit in remote memory, such as Redis [27] and Spark [128], may have better performance. However, prior studies [64, 65, 109] have shown that the performance of some unmodified applications on a disaggregated architecture can be significant reduced due to the additional

network traffic overhead. Even on a disaggregated operating system such as LegoOS, if one hides the underlying disaggregation for backward compatibility, unnecessary data movement through the underlying network could make application performance 1.7x to 2.5x slower on average for different applications [130].

To manage the hardware, existing research in resource disaggregation has mainly focused on virtual machine monitors [81, 129]. LegoOS [109] presents the disaggregated hardware as a collection of virtual servers, each appearing as a standard physical server. This is a useful approach, as it provides backward compatibility for existing applications, yet allows each virtual server to be specialized for a particular application process that runs on the virtual server. While useful, such virtual servers require many mechanisms to hide the underlying distribution of resources that hurt performance of applications [130].

To get better performance for existing applications, some kernel extensions such as Infiniswap [65], Leap [85], and Fastswap [34] provide efficient remote paging to hide memory disaggregation. This complicates the operating systems and/or virtual machine monitors while the additional level of indirection makes them less efficient than if the applications use underlying physical resources directly. pDPM [119] focuses on storage disaggregation by separating the control and data paths and by a pointer chaining technique for building a efficient disaggregated key value store. All are focusing on improving the performance on one type of resource instead of the performance on disaggregated hardware in general.

Unfortunately, there are only a few industrial applications that have adopted disaggregated architecture. These include PolarDB [49] and Snowflake [123]. Both are databases on disaggregated memory and storage. We believe that applications on disaggregated hardware are not widely deployed due to (1) the complexity of building new

	Microservice	Uniservice
Deployment unit	A logical server, such as a VM or a container	A physical hardware platform with a specialized resource
Modularity level	Coarse-grained, combining various functionalities and using multiple resource types	Fine-grained, focusing on managing a single hardware resource
Network stack	Usually TCP/IP	RDMA or optical switching network

Table 4.1: Comparison between microservices and uniservices

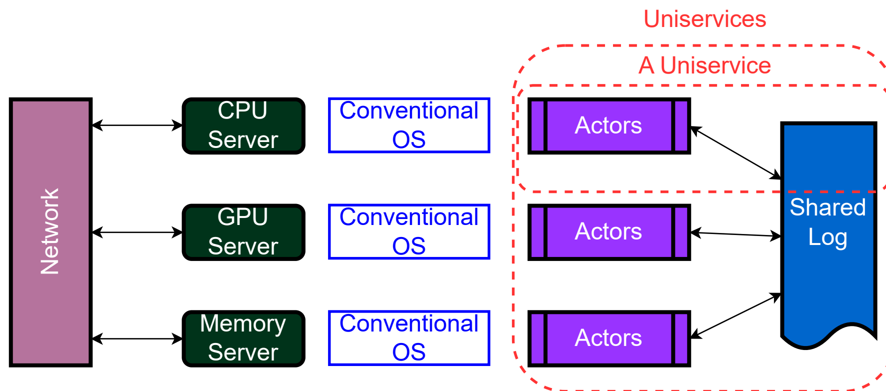


Figure 4.2: Uniservices Overview

applications, (2) the potential for performance degradation, such as unnecessary data movements across the network, and (3) mismatched assumptions between the software and the underlying physical hardware layout.

To promote widespread adoption of resource disaggregation, it is crucial to identify appropriate abstractions and develop a programming paradigm that enables the creation of efficient and high-performance applications.

4.2 Uniservice Design

To a large extent, at the time of writing this thesis, cloud applications are already highly modularized. Many new applications are built using paradigms such as serverless com-

puting and microservices, which offer various advantages. First, customers deploying the software only pay for the resources they use. Second, the applications become easily scalable, as more resources can be added as needed (elasticity). Finally, because state and function are often separated, it is relatively easy to recover from failures.

However, these applications are modularized and divided into components along ***logical*** boundaries, not ***physical*** boundaries based on resources. As a result, each component, such as a microservice or a lambda function, may still require a full complement of resources to scale out. Fortunately, we can leverage that people are already building modularized applications–but we need to shift the boundaries to align with the physical reality. To better program on disaggregated architectures, we propose building applications from uniservices that are specialized for a particular hardware resource and scale out with only that resource (Figure 4.2) with modularized actors on conventional operating systems such as Linux. To mitigate the complexity increase of decomposing applications into finer-grained components, each function within uniservices communicates through a shared log, except for real data movement, which will be going through direct links connecting servers.

A uniservice has the following features:

1. Each function is designed for scaling along a single resource type and highly modularized. Other resource consumption will be restricted.
2. Functions do not wait for responses from another function after sending requests; rather, they are running asynchronously and do not invoke any RPC except for fetching streaming data of a known size. So, the running time is predictable with a given input.
3. Functions are running on a commodity Linux kernel instead of a customized virtual machine monitor.

The uniservice design offers three benefits. First, decomposing application functions into different resource components can significantly improve resource utilization and reduce cost and energy waste for both cloud providers and application developers. In the cloud at the time of writing this thesis, particularly in serverless settings, a function is scaled out by creating more instances that have a fixed ratio between CPU and memory. Applications such as video processing and computation-intensive jobs pay the cost of extra memory to benefit from the cloud. Carefully decoupling memory and CPU allocation can potentially save cloud providers 40-50% of execution costs [45]. With the uniservice design, cloud providers could scale functions by adding more resources of the required type, knowing that the function will focus on that particular resource and not require reserving other resource types that will not be used. This approach allows an applications' resource ratio usage to converge towards an empirically sensible ratio, rather than adhering to a fixed provisioned one, thus providing more efficient and targeted scaling, better resource utilization and, consequently, significant cost and energy savings.

Second, the uniservice design allows functions to optimize their jobs with underlying hardware support. Existing research on leveraging far-memory often focuses on designing effective prefetching algorithms to avoid page faults as much as possible based on historical traces. However, if we consider that the CPUs are also far from the memory, we can view it as having far CPUs instead of far memory. From this perspective, each computational uniservice function should know its capabilities and only process stream data or data that can fit in its local memory to eliminate remote page faults. This introduces a new memory-centric application design that offloads computations to far CPUs. The most crucial job for a memory uniservice is to prepare and manipulate in-memory data so that they are ready when computation uniservices begin.

Third, a non-stop running design that does not involve any RPCs except for loading streaming data objects of known sizes makes the running time of uniservices more predictable. As a result, optimization such as using predicted running time to prewarm containers can be employed to reduce the response time of functions.

Table 4.1 shows a comparison between microservices and uniservices. Both decompose applications into different components that serve a specific purpose. However, there are a few key differences between them. First, unlike microservices, which do not have a clear boundary between different logical components, uniservices know where the boundary is by dividing upon physical boundaries among different components. This is particularly crucial because in a disaggregation environment, decomposing applications into only logical components can hurt performance due to network overhead and data movement. Second, a uniservice manages only a single type of resource, whereas a microservice can span across multiple resource types. This allows uniservices to scale independently without being capped by the resources they do not need.

A uniservice application is composed of *functions* targeting different resources, *data objects* with references and an *execution graph* describing relations among functions and objects. All three components are connected by a shared log. Such applications can be written from scratch or can be derived from existing applications by carefully separating code for different resources, such as separating CPU intensive code from memory intensive code by leveraging a profiling tool [19] or by empirical experience. For small-scale applications, using uniservices might not seem like an optimal choice as it requires developers to decompose the application to utilize different resources. However, for large-scale applications that are challenging to manage as a single program, modularization is a common strategy.

As shown in Figure 4.3, once an application is disaggregated into resource specific functions, the functions will be registered in a function hub and return an identifier from the shared log. Below, we will discuss what a function is in the uniservice design, how an *application execution graph* describes the workflow of functions, and how the shared log connects functions and data objects all together to form a uniservice application.

4.2.1 Functions

In the uniservice design, functions are the basic units of an application. A function is similar to an actor in the actor model. It is registered by appending a registration entry to the log; the sequence number of the entry is used as the unique identifier of the function. A function is invoked by appending a log entry including the function identifier, the input arguments, and the location of the server on which the function should be running. There are two types of functions, computation functions (C-Functions) and state manipulation functions (M-Functions). A C-Function is a stateless, *one-shot* function optimized for various computational processing units. This includes not only CPU code but also GPUs such as CUDA code [25], FPGAs, NPUs, and so forth. An M-Function is a stateful function made up of two components: a C-Function, denoted as C_m, and a long-lasting state, S. C_m, compared to other C-Functions, first needs to prepare data objects that will serve as input or output for other C-Functions. Second, C_m will invoke other C functions by sending an invoker function entry to the shared log. The state S is an object store containing objects used by the current application. Unlike a C-Function and C_m, S is typically available until the application concludes.

4.2.2 Data Object Reference and Movement

Data in a uniservice are stored as objects. An object is a chunk of memory that can be retrieved using an associated name. M-Functions utilize a key-value object store to manage these objects during the execution of an application. A C-Function resides on a computation node, which typically does not have as much local memory compared to a memory node. We provide three parameter types for C-Functions. These types are optimized differently to serve objects as input parameters and to minimize unnecessary data movement between memory and computation nodes over the network.

1. *Primitive types* include boolean, integers, floating point numbers, and (short) strings. The content of primitive type parameters will be available at the computation node when the function is invoked.
2. *Streamable types* are objects that are fully available at the memory node but are served incrementally to the function due to the limited amount of local cache on the computation node. The data can only be consumed sequentially and does not support random access by an arbitrary pointer.
3. *Random-access types* are objects completely available at the memory node but allow functions to explicitly request portions of a remote data object (which must fit in local memory) by accessing an address using a remote pointer.

Each data object can operate in one of three modes:

1. Read-only mode: The object cannot be modified by the C-Function.
2. Copy-on-write mode: The C-Function can alter the content by creating a new copy of the original object as output.

3. Mutation mode: The C-Function has full control to change the content of the object and can overwrite the original content on the memory node once the C-Function finishes execution.

At the time a function is invoked on streamable and random-access objects, a tuple (session id, object location, object name) is passed to the log as a data reference instead of the actual content, The session id is used to identify the current running instance of the application. To share the local cache with multiple C-Functions, an M-Function can set an object into copy-on-write mode. This allows C-Functions scheduled on the same computation node with common input objects to share the local cache of the object. Consequently, this reduces the need to load data from the memory node, further diminishing the overhead of data movement.

4.2.3 Application Execution Graph

The execution graph of a uniservice application is composed of a collection of functions. These applications leverage their execution graphs to define function relationships and orchestrate data flows. There are two methodologies for defining an execution graph: explicitly delineating the function workflow, or implicitly building the execution graph dynamically at runtime.

Explicit Execution Graph

An explicit application execution graph shares similarities with workflow graphs found in contemporary serverless platforms, such as AWS Step Functions [7]. This includes the specification of function identifiers, dependencies amongst functions, and data flows

within an application. An external agent publishes an execution graph to the log. The sequence number of the log entry serves as the unique identifier for the application. A uniservice scheduler then uses this execution graph to schedule functions based on *triggers*, such as external events or nested function invocations.

However, generating an explicit execution graph for a large-scale application can prove complex and challenging to manage. Moreover, it's difficult for applications to update dependencies dynamically at runtime. For these reasons, Uniservice also facilitates an alternative approach to implicitly build the execution graph at runtime.

Implicit Execution Graph

In order to construct an implicit execution graph, Uniservice introduces two abstractions: `UniFunction` and `UniResult`.

```
class Uniservice:
  def __init__(self, rtype=mem):
    rtype = rtype
    pass

  def call(self, *args, **kwargs) -> Result:
    pass
```

A `UniFunction` is a subclass of `Uniservice`. It is required to specify its resource type and override the `call()` method, which acts as the trigger entry point.

```
class Result:
  task_id: int
  outputs: UniArgument

  def chain(service: Uniservice, *args, ***kwargs) -> Result:
```

```
    pass
```

A `UniResult` object includes a `chain()` method, which is used to invoke the provided `service`, taking the output of the result as its first argument. The `UniResult` concept takes its inspiration from the Promise-Future asynchronous programming paradigm, with the `UniResult` serving a similar role to a Future object.

A Uniservice can take `UniResults` and generate another `UniResult` from it. The `UniResult` object is a reference to the output of a Uniservice. A Uniservice can either call `Uniservice.call(args: [UniResult])` or use `UniResult.chain(service)` to submit a trigger record to the shared log and wait for an acknowledgment containing the task ID. The dependencies of the Uniservice are determined based on the task IDs of all `UniResult` objects in the arguments. By utilizing the `chain()` and `call()` methods, the dependencies of a `UniFunction` can be tracked, enabling the implicit construction of the execution graph.

Here is an example showcasing the chaining of Uniservices for data parallel training in a large language model:

```
@uCPU
class Tokenizer(Uniservice):
  def call(self, data):
    r = Result()
    r['tokens'] = tokenize(data)
    return r

@uGPU
class Train(Uniservice):
  def call(self, tokens):
    r = Result()
    # do whatever with tokens ..
```

```
13    return r
14
15 @uGPU
16 class SyncModel(Uniservice):
17   def call(self, *grads):
18     r = Result()
19     # update model
20     return r
21
22 @uMEM
23 class Main(Uniservice):
24   def call(self, rounds: int = 0, ...):
25     r = Result()
26     if rounds > 0:
27       # loading data to ‘data‘
28       res_list = []
29       for d in data:
30         res_list.append(Tokenizer()(d).chain(Train, ...))
31       SyncModel()(*res_list).chain(Main, rounds = rounds-1, ...)
32     return r
```

`@uGPU`, `@uCPU`, and `@uMEM` denote the resource node on which a given Uniservice should run. `Tokenizer`, `Train`, `SyncModel`, and `Main` are all examples of Uniservices, with `Main` managing the application and the other Uniservices.

The `Main` Uniservice begins by loading data and dividing it into segments. Subsequently, it calls the `Tokenizer` Uniservice concurrently, chains the output of the Tokenizer to the `Train` Uniservices, and stores the result, such as gradients, from the `Train` Uniservice in the `res_list`.

Once all results are available, it merges all gradients into the model, and chains the updated model returned by the `SyncModel` to another instance of the `Main` Uniservice. This creates a recursive Uniservice pattern which trains the model over several iterations, referred to as `rounds` epochs.

4.2.4 Shared Log

To manage complexity, the uniservice platform uses a shared log as a central place for invoking and communicating among functions. There are five types of log entries used by a uniservice application:

1. **FuncRegistration**. Includes a function name and the URI of the function body. The sequence number of the log entry becomes the identifier for the function.
2. **AppRegistration**. Includes the application name and the application execution graph. The sequence number of the log entry becomes the identifier for the application.
3. **Store.** Includes a metadata for an object but not the content. The sequence number of the log entry becomes the identifier for the object.
4. **Invoke.** Includes a function identifier and parameters.
5. **Finish.** Includes a function identifier.

After an application is registered, its entry function can be invoked by some external event. Functions may trigger other functions by appending an **Invoke** entry to the log. A function can also create a new streamable object or random-access object by appending a **Store** entry with the metadata of the object.

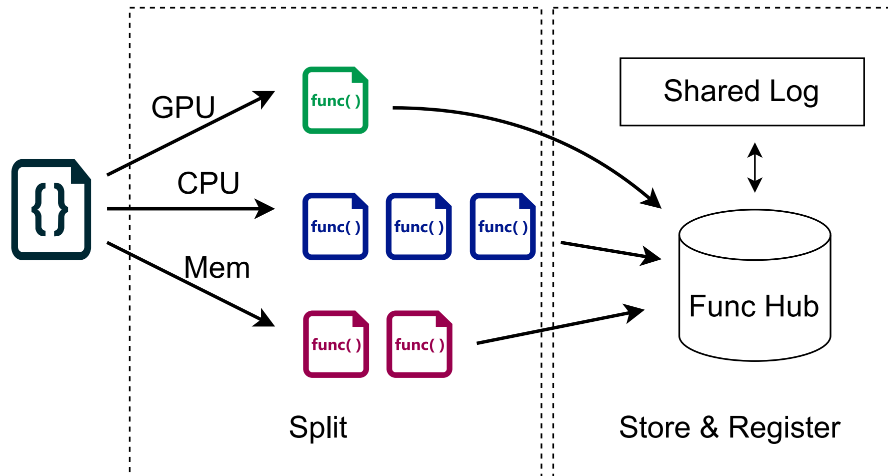


Figure 4.3: Uniservices code decomposition

As the shared log acts as a coordinator among all functions, it is important to keep it running efficiently. Even though the log is not on the critical path of each individual function's computation, overloading it can slow event delivery and potentially degrade overall performance, particularly for short-lived tasks. Therefore, it is necessary to use a fast, scalable, and consistent log (e.g., [40, 59, 71]).

4.2.5 Scheduling

During a function's execution, uniservice functions operate asynchronously in relation to each other, much like actors, but unlike microservices. The uniservice scheduler uses an execution graph to schedule functions on different resource nodes. An application completes when there are no more runnable functions in the execution graph. To achieve predictable running times for more effective scheduling, a uniservice C-Function must adhere to the following principles:

1. A C-Function does not make any external RPC calls to other C-Functions, communicating only with M-Functions to load input parameters.

2. All parameters and input data objects associated with a C-Function are available and have a known size at an M-Function.
3. A C-Function does not contain any unbounded loops within the function body.

These principles make uniservice applications particularly efficient for deterministic workloads such as machine learning training, inference and large dataset processing. For applications that have variable running times not solely determined by the input data size, we provide an interface for developers to calculate their estimated running time score for a C-Function, given the input arguments. This score is an integer value. Consequently, the scheduler can use this user-provided score to predict the running time of a given function. It does this by comparing the score against the historical score and the actual running time of the function.

4.2.6 Node Failures and Data Recovery

Both C-Functions and M-Functions can fail when the underlying hardware malfunctions. To detect physical node failures, the scheduler can utilize the shared log, which connects to all nodes, implementing a heartbeat failure detection protocol.

When a C-Function fails, since all the input objects are backed up by an M-Function running on a different memory node, the scheduler can simply relaunch a new C-Function instance. This is achieved by examining the invocation entry along with all subsequent log events it triggered before the last crash.

For functions requiring exactly-once semantics, because all the events are published on the shared log, the relaunched instance can optionally take an extra input. This input

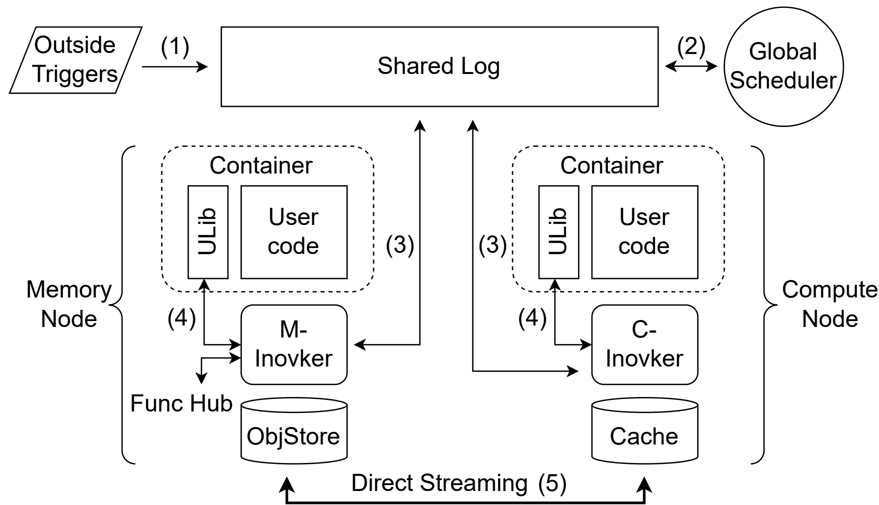


Figure 4.4: Uniservices architecture. (1) A trigger event is inserted into the log. (2) The global scheduler finds the execution graph of the triggered application and assigns functions to invokers. (3) Invokers on different resources receiving either *prewarm* or *invoker* commands, load function bodies from the function hub, and (4) launch the functions. Data movement among functions happens through the underlying RDMA network (5).

would include events the crashed function had sent to the log, allowing the new instance to resume where the previous function left off.

In the event of an M-Function failure, since the data are stored along with the M-Function, it is necessary to relaunch the function from the start on another memory node since all objects are lost. However, one could also mirror the objects or saving a snapshot of the current object store on other memory nodes. Recovery would only involve relaunching the C-Function part of an M-Function, without needing to rebuild the object store.

4.3 Implementation

We implemented a prototype of the uniservice platform in C++ using 5935 LoC. The implementation has four major components, illustrated in Figure 4.4.

4.3.1 Shared Log

The shared log is a server that listens on a particular port for clients to append log entries. Once a log entry is received, it is stored locally in memory and broadcast to all clients based on their interests. A log entry contains a tag field that indicates where it should be delivered. Clients can specify their interests when they register with the log server. All uniservice components act as clients of the shared log. Currently, the log is not sharded, but as the scale increases, a sharding approach can be taken to reduce the load on the log server [40, 59]. Not only do functions use the log, but the infrastructure itself also leverages it. Global schedulers and invokers use the log as a means of discovering and registering themselves with the system.

4.3.2 Function Hub

The Function Hub stores uniservice function binaries and registers functions to the log. In addition to the function body, an execution graph is also uploaded for the scheduler to prewarm containers. Each function can be registered to the log by appending a FuncRegistration entry. Similarly, for an application, it can be registered through an AppRegistration entry that includes an execution graph. The functions that compose the application should be registered first.

4.3.3 Global Scheduler

The Global scheduler maintains a mapping for each invoker to its available resources, as well as a map from each available resource to its invokers. The scheduler uses the shared log to assign and control the tasks. When a new task comes in, the scheduler will try to pick a slot from an invoker that registered with the resource that the job needs. The job is then prewarmed at the selected invoker and invoked when all of its parameters are available.

4.3.4 Invoker

An invoker is a server that specializes in a particular type of resource. For example, in Figure 4.4, there are CPU Invokers (C-Invoker) and Memory Invokers (M-Invoker). An invoker registers its available resource on the log and waits for the scheduler to assign a new job. Once the job arrives, the invoker fetches the code from the function hub and sets up connections to servers that own the streamable objects. When everything is available, it starts a Docker container for the code and invokes the binary inside the container. To serve the streamable object, a uniservice sidecar process runs along with the user code. This process, denoted as ULib in Figure 4.4, connects to the underlying invoker by a Unix socket and shared memory to serve incoming streamable data. Once a function finishes, the invoker cleans the Docker container and puts a Finish record on the log. A Uniservice function requires the following interface for ULib to correctly run a C-Function:

```
run(ULib* handle, Args...) -> int
```

```
bigo(const Args...) -> int
```

The `run` function takes a ULib handle for triggering commands and a list of arguments. The `bigo` function takes a list of const arguments to compute an estimated running time of the function.

4.4 Evaluation

One of the advantages of a disaggregated architecture and a uniservice system is that applications can scale independently for each resource, rather than being confined to a box with a fixed ratio of resources. This means, for example, that CPU-intensive applications can acquire more CPU cores without over-provisioning other types of resources, such as memory.

In order to assess the performance of uniservices in terms of resource utilization, we created a basic video processing application using our prototype. The workload involves simulating the process of resizing a video to various resolutions in order to accommodate users with different bandwidth capacities. This application consists of three functions: M1, C1, and M2. As illustrated in Figure 4.5, M1 (an M-Function) loads a video into memory and divides it into parts in batches, a task that is not computationally intensive as it does not involve encoding or decoding. The video segments are then sent to C1 (C-Function) workers. M1 also triggers M2 before it exits. M2, as an M-Function, shares the state with M1. This shared state is implemented as an in-memory key-value store that persists throughout the application's execution. Once the last function in the execution graph exits, the shared state is destroyed. C1 resizes the video to a different resolution, converting it from 1080p to 720p. Finally, C1 triggers and forwards the processed video clips to M2, which merges them into a new video. Usage of each resource

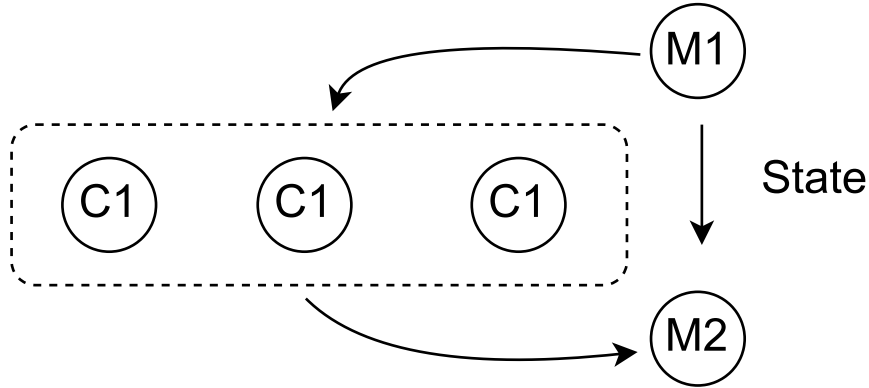


Figure 4.5: Video processing design

type is monitored by a process that invokes the `docker stats` command throughout the experiments. Resource adjustments are made based on the percentage of resource utilization P. This percentage at time t is calculated as a moving average:

$$P_i = kU_i/L + (1 - k)P_{i-1}$$

where k is a tunable parameter in the range [0, 1], L is the current limit, and U_i is the resource usage at time i. For these experiments, we used $k = 0.70$ for memory and $k = 0.67$ for CPU. For instance, for M1, memory is incrementally increased by 128MB when the moving average reaches 70 percent.

In Figure 4.6 and Figure 4.7, the memory and CPU usage of M1 and C1 functions are displayed, with memory represented in red and CPU in blue. Solid lines represent real-time CPU or memory usage, while dashed lines indicate the allocated amount of resources.

Figure 4.6 demonstrates that when processing a large video, the M1 function can scale along the memory boundary, with the number of CPU cores fixed at 1 in this case. In contrast, an AWS Lambda function performing the same task would require around 2GB of allocated memory and 2 cores as a fixed amount. This is because the Lambda function needs to provision the maximum usage before launching, and every 1769 MB

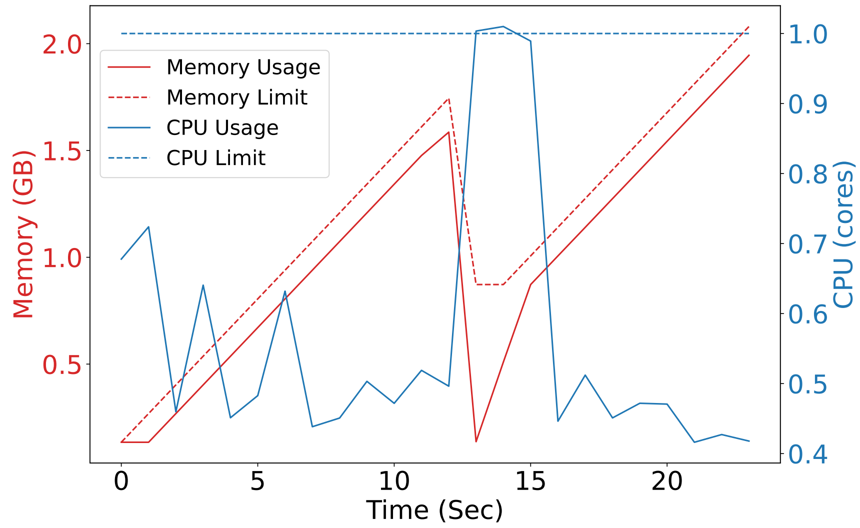


Figure 4.6: M1 memory and CPU usage

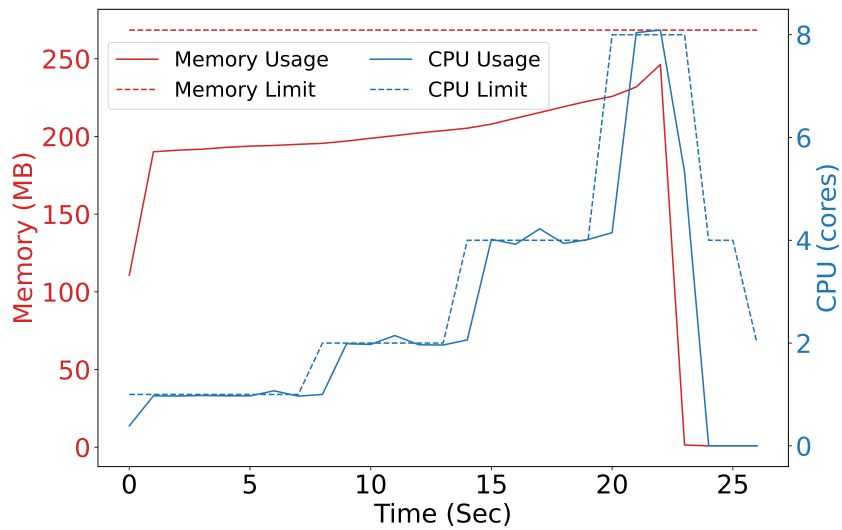


Figure 4.7: C1 memory and CPU usage

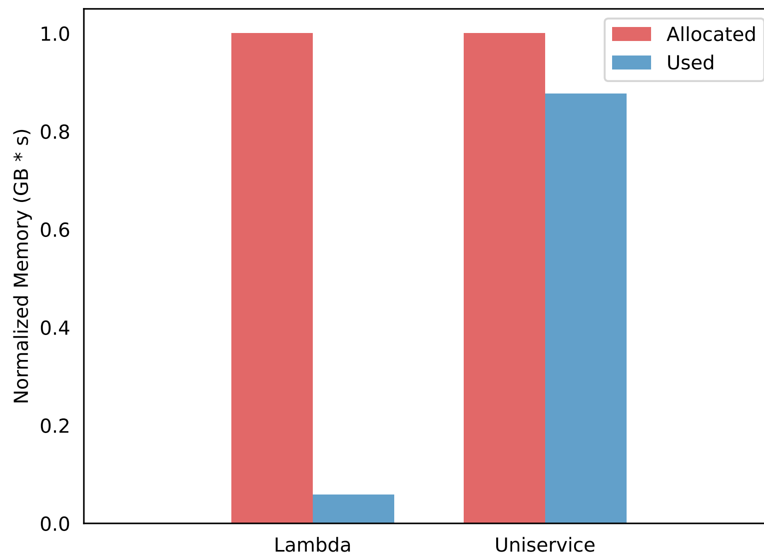


Figure 4.8: Normalized memory utilization of Lambda Function and Uniservice.

of allocated memory results in an additional core. The CPU usage spike observed in the middle, accompanied by a drop in memory usage, corresponds to the moment when a batch is completed and the data is sent to the worker.

Similarly, Figure 4.7 displays the CPU and memory usage for C1 when the input data size can be accommodated within its local memory. Unlike M1, as a C-Function, the amount of memory is fixed as 256MB for C1. As video resizing is a CPU-intensive task, C1 scales out by utilizing more CPU cores while maintaining relatively stable memory usage. In our current implementation, the scaling policy is to double the current core count. However, this policy is not fixed and can be adjusted based on user preferences.

For an AWS Lambda function, the amount of memory allocated determines the number of cores. To obtain 6 cores (the maximum number of cores a Lambda function can have), 10,240 MB of memory would need to be allocated, which is significantly more than what the C1 function with even 8 cores requires. Figure 4.8 illustrates the compar-

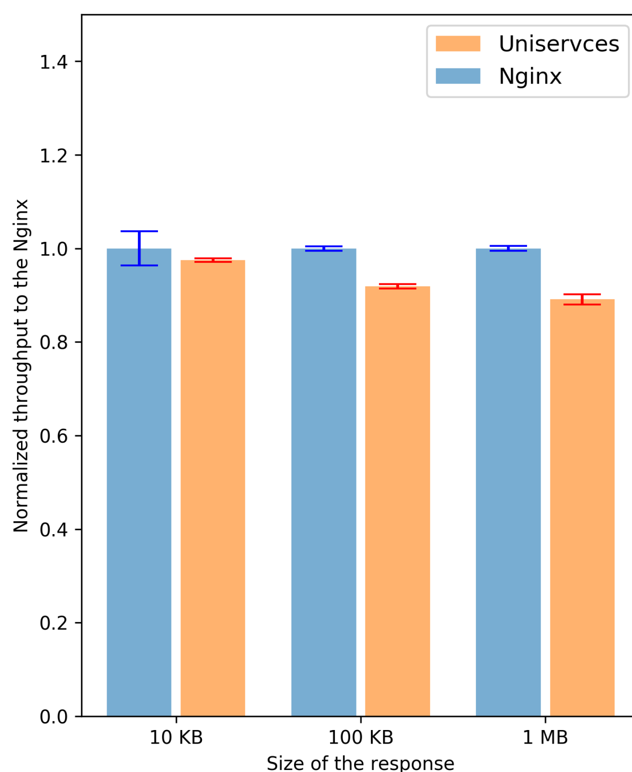


Figure 4.9: Normalized throughput of Nginx and Uniservice on serving different sizes of objects.

ison between the actual memory used versus the allocated memory when running the same video conversion workload in a single Lambda function. In comparison to Uniservice, the Lambda function utilizes only 6% of the allocated memory, while Uniservice effectively uses 87%.

Our experiments support the conclusion that uniservices can improve resource utilization by independently scaling the required resources. This is in contrast to serverless functions, such as AWS Lambda and Azure Functions, which can only scale at fixed ratios among different resources.

We also disaggregated a static web serving application into a series of workers (C-Function) running on CPU nodes and a memcached service (M-Function) operating on

the memory node. Subsequently, we utilized a microbenchmark tool, wrk [29], to simulate 1000 clients requesting static objects from the web server. The workers are responsible for handling these requests, fetching the requested objects from the memcached service via the RDMA network.

Figure 4.9 presents a comparison of normalized throughput between this Uniservice web server and Nginx [24] version 1.18.0 with 8 cores. The size of the requested objects ranges from 10KB to 1MB. The throughput of the Uniservice can achieve up to 97% of that of Nginx when dealing with a 10KB payload, but it drops to 89% with a larger 1MB payload. This decrease can be attributed to the fact that an increase in payload size adversely affects RDMA latency. This demonstrates that, given a fast enough network, Uniservice has the potential to provide performance that is comparable to certain monolithic applications when using a similar amount of resources.

4.5 Related Work

Disaggregation and Serverless. Molecule [60] is a serverless system designed to work with heterogeneous resources. It introduces a new shim layer, providing a set of abstractions that need to be implemented for each resource type, hiding the underlying heterogeneity from serverless functions.

FUSEE [112] is a fully disaggregated key-value store that runs on a disaggregated architecture. They divide the key-value store into a compute part and a memory part. Scad [67] also proposes a design that breaks down a large serverless application into components for each resource. However, Scad relies on caching trace history for prefetching to reduce remote page faults and does not have an active function on the memory node. It uses the memory node as a shared memory region among different

compute nodes. Ray [90] is a distributed computation framework designed for emerging machine learning tasks. However, our design differs from Ray in a few key aspects. First, we designed specifically for a disaggregated environment, leveraging a fast shared log for global scheduling and limiting data movement. Moreover, a node failure does not require a complex recovery protocol like Ray, thanks to the inherent hardware disaggregation.

Actor Model. The *Actor Model*, introduced in 1973 [69], is designed for writing concurrent programs in high-performance networks. In the post-Moore's Law era, the emergence of many-core systems has revitalized this paradigm. Erlang [12] is a functional programming language that incorporates ideas from the Actor Model. In Erlang, each actor is a process with no shared memory between two actors. Erlang's actors use pattern matching to receive and filter messages. Akka [4] is a serverless framework that utilizes the Actor Model and can leverage a shared log, such as Kafka, for message exchange. However, Akka does not support disaggregated architecture. Kappa [98] is a distributed serverless framework designed for deploying serverless functions on Internet of Things (IoT) devices. It is built on the foundation of Calvin, an IoT serverless platform. Similar to our design, Calvin [97] aligns the capabilities of nodes, such as laptops or Raspberry Pi boards, with the requirements of various exection units (actors).

Log-based Applications. In recent years, the study of applications on shared logs has gained traction, driven by the emergence of novel fast logs [39, 40, 59, 83] and blockchains [13, 92]. Tango [41] is a framework that facilitates the creation of distributed data structures over a shared log, simplifying the development of decentralized systems. Boki, a serverless runtime, further extends this concept by employing a shared-log *metalog* [71] to construct stateful serverless functions. Our design resembles Boki in its utilization of shared logs to store application metadata and ensure fault tolerance,

but it also incorporates the actor model and leverages the underlying disaggregated architecture.

Blockchain, first introduced by Nakamoto in 2008, serves as a distributed shared log managed by all network participants. Smart contracts, such as Ethereum smart contracts [124], are code segments stored and executed on a blockchain by maintaining the program's state on-chain. Typically, a smart contract is associated with a universal programming language applicable to all services. Drawing inspiration from smart contracts, Uniservice also places its code metadata and states on the shared log, enabling applications to seamlessly restart upon crashing by merely replaying the log.

4.6 Conclusion

To take full advantage of disaggregated data centers, we propose to forego an approach based on logical servers in favor of an approach based on uniservices, each specialized for a particular type of hardware resource. Applications based on uniservices can be more easily scaled in various dimensions (based on available resources) and take advantage of new resource types, improving overall utilization of resources and reducing waste. Today programmers are already writing code that specializes for specific resource types such as GPUs and network processors—we propose to extend this to all resource types using the actor model and a shared log.

CHAPTER 5

CONCLUSION

In this dissertation, we address the problem of heterogeneous environments in some aspects by introducing and evaluating a *resource-egalitarian* abstraction and programming paradigm. This paradigm leverages shared logs and actors and is tailored for diverse heterogeneous hardware environment. We replace the CPU in the CPU-centric abstraction with a shared log and use actors in place of processes. An IPC call transforms into an invocation record on the shared log from one actor to another. Depending on the specific environment, data movement can occur through local shared memory or across a network.

In an IoT environment, we utilize a blockchain based on proof-of-witness as the shared log. Actors here are smart contracts that respond to blockchain blocks. To resolve the issue of storing every block on every node, we introduce a novel failure detection protocol, BloomBox, which stores blocks on a select few replicas in a geographic hash table, rather than on every node.

In a disaggregated cloud, we use a shared log such as Kafka or Corfu. Applications are also disaggregated into different functional components, aligned with the physical boundaries. In this context, actors are the disaggregated application functional components that can share their data either through remote memory pools or by copying across the network to their local cache.

There are certain limitations in our current design. First, although asynchronous actors are designed to prevent indefinite blocking of application execution, this system may not be suitable for applications with stringent latency requirements. Second, our programming paradigm necessitates that application developers rework their applications

into actors, which can pose a significant challenge for complex, large-scale projects. Last, not all applications can be conveniently divided along physical boundaries.

For future improvements, we propose to expand the Uniservice platform to support runtimes for GPUs and FPGAs. We also aim to optimize communication overhead between different resource nodes by implementing efficient prefetching and caching policies, an area that certainly merits further research. Last but not least, we plan to make use of recent Large Language Models (LLMs) to simplify the process of partitioning existing application code into uniservice units.

BIBLIOGRAPHY

[1] Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption/, 2020.

[2] Cambridge bitcoin electricity consumption index. https://cbeci.org/, 2020.

[3] AI ASICs Will Become Increasingly Application-Specific. https://semiengineering.com/ai-asics-will-become-increasingly-application-specific/, 05 2023.

[4] AKKA. https://akka.io/, 3 2023.

[5] AWS Lambda. https://aws.amazon.com/lambda/, 3 2023.

[6] AWS Lambda Ratio. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, 3 2023.

[7] AWS Step Functions. https://aws.amazon.com/step-functions/, 05 2023.

[8] Bsim. https://crypto.news/blockchain-based-mobile-sim-card-goes-live-in-china/, 2023.

[9] Canada CDBC. https://www.bankofcanada.ca/digitaldollar/, 2023.

[10] Compute Express Link. https://www.computeexpresslink.org/, 3 2023.

[11] DDR5 memory: Everything you need to know. https://www.crucial.com/articles/about-memory/everything-about-ddr5-ram, 2023.

[12] Erlang. https://www.erlang.org/, 3 2023.

[13] Ethereum whitepaper. https://ethereum.org/en/whitepaper/, 3 2023.

[14] Europe CBDC. https://www.ecb.europa.eu/paym/digital_euro/html/index.en.html, 2023.

[15] Evolution of Single-threaded x86 CPU Performance. https://mlech26l.github.io/pages/2020/12/17/cpus.html, 05 2023.

[16] Gen-Z Consortium. https://www.computeexpresslink.org/projects-3, 3 2023.

[17] Google Cloud Platform. https://console.cloud.google.com/compute/instancesAdd, 05 2023.

[18] Google Nearby. https://developers.google.com/nearby, 2023.

[19] GProf. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html, 04 2023.

[20] How DNAnexus and Edico Genome are Powering Precision Medicine on Amazon Web Services (AWS). https://aws.amazon.com/blogs/apn/how-dnanexus-and-edico-genome-are-powering-precision-medicine-on-amazon-web-services-aws/, 05 2023.

[21] HP The Machine. https://www.hpl.hp.com/research/systems-research/themachine/, 05 2023.

[22] Intel Blockscale ASIC. https://www.intel.com/content/www/us/en/products/docs/blockchain/custom-asic-product-brief.html, 05 2023.

[23] IOTA. https://www.iota.org/, 2023.

[24] NGINX. https://www.nginx.com/, 3 2023.

[25] Nvidia CUDA. https://developer.nvidia.com/cuda-toolkit, 05 2023.

[26] Protocol buffers. https://developers.google.com/protocol-buffers, 2023.

[27] Redis. https://redis.io/, 10 2023.

[28] TPU. https://cloud.google.com/tpu, 05 2023.

[29] Wrk2. https://github.com/giltene/wrk2, 2023.

[30] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith

Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Regions: A Simple Abstraction for Remote Memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[31] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 121–127, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 120–126, 2019.

[33] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 101(6):255–261, 2007.

[34] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[35] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[36] Krste Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, 2014. USENIX Association.

[37] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. Design and implementation of a p2p cloud system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 412–417, 2012.

[38] Leemon Baird. The Swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds, Inc. Technical Report SWIRLDS-TR-2016*, 1, 2016.

[39] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632, 2020.

[40] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, 2012.

[41] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340, 2013.

[42] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[43] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Iddo Bentov, Pavel Hubácek, Tal Moran, and Asaf Nadler. Tortoise and Hares Consensus: the Meshcash framework for incentive-compatible, scalable cryptocurrencies.

[45] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions, 2021.

[46] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[47] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[48] Matthew Caesar, Miguel Castro, Edmund B Nightingale, Greg O'Shea, and Antony Rowstron. Virtual Ring Routing: network routing inspired by DHTs. *ACM SIGCOMM Computer Communication Review*, 36(4):351–362, 2006.

[49] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. PolarDB serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2477–2489, 2021.

[50] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. Contextual Understanding of Microservice Architecture: Current and Future Directions. *SIGAPP Appl. Comput. Rev.*, 17(4):29–45, January 2018.

[51] Ranveer Chandra, Venugopalan Ramasubramanian, and Kenneth Birman. Anonymous gossip: Improving multicast reliability in mobile ad-hoc networks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 275–283. IEEE, 2001.

[52] Jonathan Cheng. China rolls out pilot test of digital currency. *Wall Street Journal*, 20, 2020.

[53] Mihai Christodorescu, Wanyun Catherine Gu, Ranjit Kumaresan, Mohsen Minaei, Mustafa Ozdayi, Benjamin Price, Srinivasan Raghuraman, Muhammad Saad, Cuy Sheffield, Minghua Xu, et al. Towards a two-tier hierarchical infrastructure: an offline payment system for central bank digital currencies. *arXiv preprint arXiv:2012.08003*, 2020.

[54] Anton Churyumov. Byteball: A decentralized system for storage and transfer of value. `https://byteball.org/Byteball.pdf`, 2016.

[55] B Cihan. Getting started with active-active geo-distribution for Redis applications with CRDTs. `https://dzone.com/articles/getting-started-with-active-active-geo-distributio`, 2017.

[56] Alex de Vries. Bitcoin's energy consumption is underestimated: A market dynamics approach. *Energy Research and Social Science*, 70, 2020.

[57] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[58] Luca Deri. High-speed dynamic packet filtering. *Journal of Network and Systems Management*, 15(3):401–415, 2007.

[59] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, 2020.

[60] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 797–813, 2022.

[61] Jingzhe Du, Evangelos Kranakis, and Amiya Nayak. Distributed storage in disruption tolerant network. In *2010 IEEE International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 1–6. IEEE, 2010.

[62] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

[63] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[64] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, 2016.

[65] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[66] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. The Dynamic Bloom Filters. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):120–133, 2009.

[67] Zhiyuan Guo, Zachary Blanco, Mohammad Shahrad, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, Harry Xu, and Yiying Zhang. Decomposing and Executing Serverless Applications as Resource Graphs, 2022.

[68] Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[69] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute Menlo Park, CA, 1973.

[70] Márk Jelasity. Gossip. In *Self-organising software*, pages 139–162. Springer, 2011.

[71] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[72] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile computing*, pages 153–181. Springer, 1996.

[73] Flavio Junqueira and Keith Marzullo. Designing algorithms for dependent process failures. In *Future Directions in Distributed Computing*, pages 24–28. Springer, 2003.

[74] Kolbeinn Karlsson, Weitao Jiang, Stephen Wicker, Danny Adams, Edwin Ma, Robbert van Renesse, and Hakim Weatherspoon. Vegvisir: A partition-tolerant blockchain for the internet-of-things. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1150–1158. IEEE, 2018.

[75] Brad Karp and Hsiang-Tsung Kung. GPSR: Greedy Perimeter Stateless Routing for wireless networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 243–254, 2000.

[76] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, Kobi Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 690–695. IEEE, 2016.

[77] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[78] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. ACM New York, NY, USA, November 2000.

[79] Simon S Lam and Chen Qian. Geographic routing in d-dimensional spaces with guaranteed delivery and low stretch. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):217–228, 2011.

[80] Olaf Landsiedel, Stefan Gotz, and Klaus Wehrle. Towards scalable mobility in Distributed Hash Tables. In *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, pages 203–209. IEEE, 2006.

[81] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news*, 37(3):267–278, 2009.

[82] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[83] Joshua Lockerman, Jose M Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A partially ordered shared log. In *13th US ENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 357–372, 2018.

[84] Xiao Lv, Fazhi He, Weiwei Cai, and Yuan Cheng. A string-wise CRDT algorithm for smart and large-scale collaborative editing systems. *Advanced Engineering Informatics*, 33:397–409, 2017.

[85] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020.

[86] Miguel Matos, António Sousa, José Pereira, Rui Oliveira, Eric Deliot, and Paul Murray. Clon: Overlay networks and gossip protocols for cloud environments. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 549–566. Springer, 2009.

[87] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[88] Jon Foss Mikalsen. Firechain: An efficient blockchain protocol using secure gossip. Master's thesis, UiT Norges arktiske universitet, 2018.

[89] Yaron Minsky and Richard Trachtenberg, Ari andZippel. Set reconciliation with nearly optimal communication complexity. In *Transactions on Information Theory*, volume 49.9, pages 2213–2218. IEEE, 2003.

[90] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 561–577, 2018.

[91] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. September 2001.

[92] Satoshi Nakamoto. A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[93] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[94] Nvidia. NDR 400G INFINIBAND ARCHITECTURE. `https://www.nvidia.com/en-us/networking/ndr/`, 3 2023.

[95] Nathan Pemberton. Exploring the Disaggregated Memory Interface Design Space. In *The First Workshop on Resource Disaggregation (WORD)*, 2019.

[96] Nathan Pemberton and Johann Schleier-Smith. The Serverless Data Center: Hardware Disaggregation Meets Serverless Computing. In *The First Workshop on Resource Disaggregation (WORD)*, 2019.

[97] Per Persson and Ola Angelsmark. Calvin–merging cloud and IoT. *Procedia Computer Science*, 52:210–217, 2015.

[98] Per Persson and Ola Angelsmark. Kappa: serverless IoT deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 16–21, 2017.

[99] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: replicated database services for world-wide applications. In *Proceedings of the 7th ACM SIGOPS European workshop*, pages 275–280, 1996.

[100] Michal Piorkowski, Natasa Sarafijanovoc-Djukic, and Matthias Grossglauser. A parsimonious model of mobile partitioned networks with clustering. In *The First International Conference on COMmunication Systems and NETworkS (COMSNETS)*, January 2009.

[101] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient Bloom Filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[102] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-addressable Network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, 2001.

[103] Sylvia Ratnasamy, Brad Karp, Scott Shenker, Deborah Estrin, Ramesh Govindan, Li Yin, and Fang Yu. Data-centric storage in sensornets with GHT, a Geographic Hash Table. *Mobile networks and applications*, 8(4):427–442, 2003.

[104] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. RFC 5040: A Remote Direct Memory Access Protocol Specification, October 2007.

[105] Christian Esteve Rothenberg, Carlos AB Macapuna, Fábio L Verdi, and Mauricio F Magalhaes. The Deletable Bloom Filter: a new member of the Bloom family. *IEEE Communications Letters*, 14(6):557–559, 2010.

[106] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. The Variable-Increment Counting Bloom Filter. *IEEE/ACM Transactions on Networking*, 22(4):1092–1105, 2013.

[107] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[108] Salvatore Sanfilippo. Redis. `http://redis.io`, 2009.

[109] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[110] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[111] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[112] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 81–98, 2023.

[113] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)*, 34(3):1–31, 2016.

[114] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol.

[115] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[116] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using Extended Bloom Filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.

[117] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. Spray and Wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 252–259, 2005.

[118] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of Bloom Filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[119] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020.

[120] Amin Vahdat, David Becker, et al. Epidemic routing for partially connected ad hoc networks, 2000.

[121] Robbert van Renesse. A blockchain based on gossip? – a position paper. In *Distributed Cryptocurrencies and Consensus Ledgers (DCCL 2016)*, July 2016.

[122] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 352–367, 2022.

[123] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[124] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[125] Fetahi Wuhib, Rolf Stadler, and Mike Spreitzer. Gossip-based resource management for cloud environments. In *2010 International Conference on Network and Service Management*, pages 1–8. IEEE, 2010.

[126] Yang Xiao. Accountability for wireless lans, ad hoc networks, and wireless mesh networks. *IEEE Communications Magazine*, 46(4):116–126, 2008.

[127] Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless BFT consensus through metastability. *CoRR*, abs/1906.08936, 2019.

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

[129] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. GiantVM: a type-II hypervisor implementing many-to-one virtualization. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 30–44, 2020.

[130] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow.*, 13(9):1568–1581, May 2020.

[131] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.

[132] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1042–1057, 2022.