# Dynamic Service Management in Infrastructure-Based Mobile Networks

Edward Bortnikov

# Dynamic Service Management in Infrastructure-Based Mobile Networks

Research Thesis
Submitted in Partial Fulfillment of the
Requirements for the
Degree of Doctor of Philosophy

**Edward Bortnikov**

Submitted to the Senate of
the Technion — Israel Institute of Technology
Tishrey, 5769 Haifa September 2008

The Research Thesis was done under the supervision of
Prof. Israel Cidon and Prof. Idit Keidar
in the Department of Electrical Engineering.

The generous financial help of the Israeli Ministry of Science (Eshkol Scholarship), the Prof. Jacobs/Quallcom Foundation, the Prof. Andrew and Erna Finci Viterbi Foundation, the Intel Corporation and the Technion is gratefully acknowledged.

# Acknowledgements

I am profoundly thankful to Israel and Idit for so many things I learned from them in the past four years. Each of them, in a very special way, was a perfect mentor and friend. Working with them was fun, every single moment, despite the ups and the downs. Thank you for your patience, casualty, and confidence in success.

Many thanks to the staff of the Networking Lab and the Software Lab, especially Yoram and Ilana, for their support all the way long. I also thank all the undergraduate students whose work contributed to my research.

I thank my supporting family, and especially my grandfather Benjamin, for whom this degree was so important. Thanks to my loving wife Vita for helping me realize that I need to earn it. Last but not least, thanks to our three sons – Guy, Amit and Yuval – just for being there, day by day, and making us happy.

# Contents

# List of Figures

# List of Tables

# Abstract

The paradigm of delivering real-time applications through clouds of geographically distributed service points is becoming increasingly attractive for mobile operators. Our research explores distributed management of quality of service (QoS) requirements through this infrastructure. The anticipated system scale and the need to adapt to changing behavior of mobile users raise novel problems and call for *local* and *adaptive* distributed optimization algorithms behind the cloud framework.

We focus on problems of dynamic assignment of mobile users and groups thereof to *application-level* service points. In contrast with link-layer associations, which are primarily driven by physical proximity to the infrastructure, application session assignment must jointly consider network distances, congestion, and handoff costs to optimize the QoS in the long run. We combine theoretical approaches with simulation and prototype system implementations.

We first consider a single-user case, namely, the problem of dynamic balancing between the desire to always assign the user to the closest server, and the need to reduce the number of handoffs. We propose an optimal offline solution, and a tightly competitive and efficient online algorithm, DTrack. We also demonstrate motion-aware algorithms, which achieve a near-optimal result using a very limited and noisy movement prediction.

Next, we address the problem of assigning multiple users to servers. This assignment must jointly consider loads and distances, which we call load-distance balancing, or LDB. We analyze multiple flavors of this optimization problem in the centralized setting, and provide efficient polynomial algorithms for them. Following this, we present a scalable distributed solution, Ripple, which can use any sequential algorithm as a local building block. Ripple adapts its overhead to network congestion, and constructs a local assignment whenever possible.

Finally, we propose a comprehensive handoff management framework, QMesh, which addresses all assignment factors in the context of a large-scale wireless mesh network (WMN). We perform a comprehensive simulation study of QMesh based on real location and mobility data, and demonstrate its significant advantage over traditional approaches. We also present a QMesh software prototype implemented within the Win32 kernel, which harnesses multiple desktops as a QoS-aware WMN infrastructure.

# Notations and Abbreviations

| | | |
|---|---|---|
| AP | - | Access Point |
| ARM | - | Application Resource Monitoring |
| ARP | - | Address Resolution Protocol |
| AWWP | - | Alternating Weighted WayPoint |
| CDN | - | Content Delivery Network |
| DHCP | - | Dynamic Host Configuration Protocol |
| GW | - | Gateway |
| ICMP | - | Internet Control Message Protocol |
| IP | - | Internet Protocol |
| IRDP | - | ICMP Router Discovery Protocol |
| LAN | - | Local Area Network |
| LDB | - | Load-Distance Balancing |
| LQSR | - | Link Quality Source Routing |
| MAC | - | Media Access Control |
| MAP | - | Mobile Anchor Point |
| MCL | - | Mesh Connectivity Layer |
| MMOG | - | Massively Multiplayer Online Game |
| MOS | - | Mean Opinion Score |
| MPLS | - | MultiProtocol Label Switching |
| MTS | - | Metrical Task System |
| NAT | - | Network Address Translation |
| NDIS | - | Network Driver Interface Specification |
| QoS | - | Quality of Service |
| RTP | - | Real-Time Protocol |
| RTCP | - | Real-Time Control Protocol |
| RWP | - | Random Waypoint |
| SIP | - | Session Initiation Protocol |
| SNR | - | Signal to Noise Ratio |
| TCP | - | Transmission Control Protocol |
| TE | - | Traffic Engineering |
| UDP | - | User Datagram Protocol |
| VoIP | - | Voice over IP |
| WAN | - | Wide Area Network |
| WLAN | - | Wireless LAN |
| WMN | - | Wireless Mesh Network |

# Chapter 1

# Introduction and Background

The perception of networked service delivery to mobile users has been rapidly evolving over the last decade. There is strong evidence that future wireless network infrastructures will conform to the TCP/IP architecture and its related supporting mechanisms for quality of service (QoS), and mobility. TCP/IP is being adopted by emerging standards for beyond-3G cellular networks [91]. This trend enables the convergence of the traditionally voice-oriented cellular networks with data access services over the global Internet. At the same time, low-cost and high-speed wireless access to IP networks is becoming widely available via 802.11x (WiFi) and 802.16x (WiMax) [14]. The latest generations of these standards offer increased mobility and QoS support, allowing the proliferation of real-time multimedia services over traditional data networks.

Following the recent trend in wireline networks [9], mobile operators are expected to deploy QoS-sensitive services at the network edge. This approach promises a revolutionary improvement of user experience and scale-up of service capacities compared to traditional data-center architectures. Consider, for example, wireless mesh networks, or WMNs [16] – a growing promise for broadband Internet delivery to the areas of limited wired connectivity. WMN users access the Internet through a multi-hop backbone of fixed wireless routers. The current perception of a router connected to the wired network, called gateway, is sharing its access link among multiple users. We envision extending this functionality to a wide variety of stateful session-oriented applications, e.g., a media cache [67], a VoIP relay [47], a groupware (e.g., push-to-talk) hub [73], an online gaming server [42], or a mix-and-match thereof (see Figure 1.1 for illustration).

This dissertation, which embodies most of the contribution of the Mobility and Group Management Architecture (MAGMA) research project [11], addresses the challenge of managing the complexity of mobile service provisioning through a large-scale distributed infrastructure. In this context, dynamic assignment of mobile users or groups thereof to application service points raises many novel algorithmic aspects. For example, it must jointly consider a variety of factors that affect the QoS, namely, network distances between users and servers, congestion, and handoff costs. Real-time assignment decisions must handle an inherent lack of complete information dictated by dynamic phenomena (user mobility, churn, flash crowds etc.) as well as by the anticipated system scale (thousands of service points and millions of users). These harsh constraints call for *local* and *adaptive* distributed algorithms that approximate global optimization in presence of uncertainty. We propose rigorous solutions, and complement them with extensive simulations and prototype

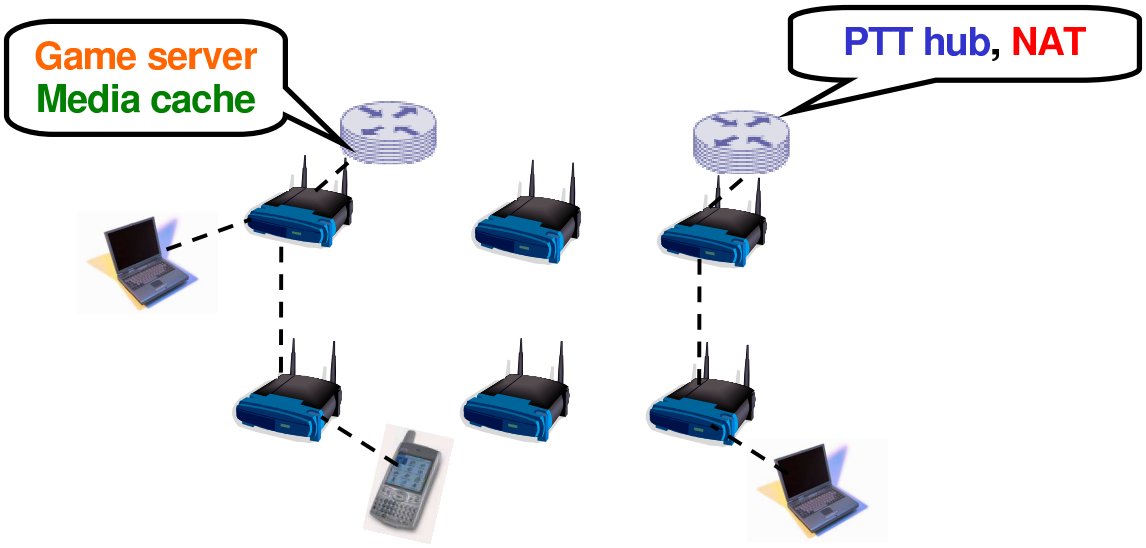



Figure 1.1: **Application services deployment at the edge of a WMN.**

implementations in real systems.

In Chapter 3, we consider the problem of dynamically assigning application sessions of mobile users or user groups to service points. Such assignments must balance the tradeoff between two conflicting goals. On the one hand, we would like to connect a user to the closest server, in order to reduce network costs and service latencies (the impact of congestion is considered negligible). On the other hand, we would like to minimize the number of costly session migrations, or handoffs, between service points. We tackle this problem using two approaches. First, we employ algorithmic online optimization to obtain competitive algorithms whose worst-case performance is within a factor of the optimal. Next, we present scalable opportunistic variations of these algorithms, which asymptotically improve the average-case performance. Finally, we demonstrate motion-aware algorithms, which achieve a near-optimal result using a very limited and noisy movement prediction. We conduct case studies of two settings where such algorithms are required: wireless mesh networks with mobile users, and wide-area groupware applications with or without mobility. The results of Chapter 3 appear in [37] (preliminary version in [36]).

In highly utilized networks, QoS-sensitive service assignment should jointly handle network distances and congestion. Chapter 5 introduces a *load-distance balancing* (LDB) problem in assigning users of a delay-sensitive networked application to servers. We model the service delay experienced by a user as a sum of a network-incurred delay, which depends on its network distance from the server, and a server-incurred delay, stemming from the load on the server. The problem is to minimize the maximum (alternatively, average) service delay among all users. We study this problem in the centralized setting (Chapter 4), and further focus on its min-max variation in a distributed setting (Chapter 5).

We prove the NP-hardness and hardness of approximation of the min-max LDB problem in Chapter 4. Following this, the best possible 2-approximation algorithm for general network distance functions is presented. We also demonstrate polynomial algorithms for the min-average LDB

problem, and the special case of min-max LDB in which network distances are Euclidean distances on a line.

Chapter 5 addresses the challenge of finding a near-optimal min-max LDB assignment in a scalable distributed manner. The key to achieving scalability is using *local* solutions, whereby each server only communicates with a few close servers. Note, however, that the attainable locality of a solution depends on the *workload* – when some area in the network is congested, obtaining a near-optimal cost may require offloading users to remote servers, whereas when the network load is uniform, a purely local assignment may suffice. We present algorithms that exploit the opportunity to provide a local solution when possible, and thus have communication costs and stabilization times that vary according to the network congestion. We evaluate our algorithms with a detailed simulation case study of their application in assigning static hosts to Internet gateways in an urban WMN. Preliminary results of this research appear in [38].

Chapter 6 jointly considers load peaks, user mobility, and handoff penalties, thus providing a unified approach to cost-driven handoff management (Chapter 3) and load-distance balancing (Chapters 4 and 5). We focus on very large mobile WMN environments, and propose QMesh, a fully distributed framework for user-gateway assignment. QMesh runs inside the WMN, and is oblivious to underlying routing protocols (a traffic engineering approach). It employs a scalable and adaptive probing policy for LD-balanced gateway selection, which is fundamentally different from the approach taken in Chapter 5. We evaluate QMesh using realistic delay models through an extensive simulation (mostly of VoIP) in two settings: (1) a university campus network, with user mobility traces from the public CRAWDAD dataset [2], and (2) a large-scale urban WMN. Simulations demonstrate that QMesh achieves significant QoS improvements and network capacity increases compared to traditional handoff policies, and illustrate the need for intelligent gateway assignment within the mesh. The results of Chapter 6 appear in [39].

Finally, we present a partial prototype implementation of QMesh (Chapter 7) – a software package that allows utilizing multiple geographically scattered Windows desktops as a wireless mesh network infrastructure with seamless user mobility support. This infrastructure supports its users through standard protocols, and does not require any client software installation. QMesh is implemented as a Windows XP driver, on top of the Mesh Connectivity Layer (MCL) toolkit from Microsoft Research that provides basic routing capabilities. To the best of our knowledge, this is the first mobile mesh solution implemented within the Win32 kernel space. The results of this chapter appear in [40].

All in all, our work makes multiple contributions to the study of managed mobile services. Chapter 8 discusses the conclusions from our research and outlines future directions.

## 1.1 Novelty and Related Work

The high-level goal of this work is contributing to the theory, networking, and systems communities in studying the user problems related to assigning mobile users to service points in a large-scale environment. We contrast the high-level approach taken throughout the thesis to previous approaches. Specific comparisons of particular results with their related techniques appear in appropriate chapters.

Service placement and selection problems have been widely addressed by the CS-theoretic community (see, e.g., [78] for an extensive survey). More recently, the networking community addressed them as well, e.g., in the context of placing mirror caches in content delivery networks [67, 87]. The distributed computing community started addressing the service location and association problems in the context of local computation, i.e., studying the tradeoff between the running time and the achieveable approximation of the optimal cost [70, 79, 71]. In contrast with the traditional approach, which considered static users and facilities, computational geometry researchers addressed dynamic clustering and center maintainance in mobile networks [30, 59]. All of the above focused on optimizing the user-server network distances while ignoring congestion arising among the sessions/flows. A recent research (e.g., [46, 47]) addressed this shortcoming but focused on systems with fairly static users and predictable traffic requirements, e.g., Internet backbone. We take this study one step further, by addressing mobile users which create dynamic workloads.

Handoff optimizations in mobile systems have been studied mostly in the context of cellular networks (e.g., [86]). These studies primarily focused on optimizing network capacity. In contrast with application-level assignments, the link-level associations between cellular users and base stations are primarily driven by physical metrics (e.g., signal strength, signal-to-noise ration (SNR), etc), which leave little room for freedom. In addition, voice session handoffs have negligible performance impact, whereas stateful application transitions may incur high costs.

In the theory plane, the interplay between distances, loads, and handoff costs opens an opportunity to explore new algorithmic problems. We address traditional problems, in which the whole workload is known a-priori, as well as online optimization problems, in which new inputs arrive in the course of the execution [33]. For example, optimizing the handoff sequence in the presence of transition penalties is a variation of the seminal metrical task system (MTS) problem [34, 26]. In this context, a few realistic assumptions about the mobility model allow specific optimizations that significantly outperform the general-purpose solutions. As a second example, considering loads and distances together (the load-distance balancing problem) is a novel approach to service assignment, which resembles the facility location [78] but has a different model, and subsequently, a different solution. We extend this direction by exploring load-distance balancing in a large-scale distributed setting, and presented scalable local algorithms.

In the system/experimental plane, we pursued practical policies (adapted from the theoretical algorithms) which solve the problems that are not addressed by the currently deployed solutions. For example, the prototype implementations of mobile WMNs that recently emerged (e.g., [18]), adopt the nearest-gateway handoff policy, i.e., each useris automatically assigned to the closest gateway. This simple local approach is acceptable for small-scale installations, however it fails to handle hotspots and handover costs in wide-area WMNs correctly. Our work proposes alternative QoS-sensitive assignment policies which, on one hand, scale well with network size, and on the other hand, are simple enough to implement in a real-time environment (e.g., city-wide WMN). These policies are verified through extensive simulations and, partially, through a working mobile WMN prototype.

# Chapter 2

# Methodology

We evaluate our research through closed-form analysis, extensive simulations, and prototype implementations. We precisely define the system models and optimization goals, and rigorously analyze the proposed algorithms wherever possible. Simulations primarily cover the studies in which the worst-case theoretical evaluation does not capture the reality correctly. In this context, we employ credible workloads based on public datasets [2] or widely accepted synthetic models [65, 96]. Finally, a prototype software implementation provides a proof of feasibility of our approach in a real-time environment, and demonstrates interoperability with technologies on the ground.

We employ a variety of theoretical analysis methods, in accordance with the problems studied. Since the research deals with optimization problems, we compare our solutions to optimal ones, and express their quality in terms of approximation factors. We explore the tightness of analysis by demonstrating lower bounds within a constant factor from the algorithms' worst-case performance guarantees. For example, Chapter 3 addresses service assignment as online optimization, in which no forecast about the user's future location exists. We apply competitive analysis, which limits the worst-case ratio between the cost produced by our online algorithms and the optimal cost, for all possible input sequences (mobility patterns). We employ intelligent heuristics to improve over theoretical competitive solutions with worst-case guarantees when side information (e.g., motion prediction) can be exploited. In a one-shot setting, we employ reductions to prove computational hardness, and present a constant-approximation algorithm for an NP-complete centralized load-distance balancing problem (Chapter 4). In some cases, when an optimal solution of an online or offline optimization problem is well-structured, dynamic programming is applicable to achieve a time- and space-efficient algorithm (e.g., Chapters 3 and 4).

Our algorithms are designed for a distributed environment, hence they address challenges specific for this area, e.g., the protocols' progress and termination properties. For example, in Chapter 5, we ensure convergence through randomized tie-breaking. At the protocol evaluation side, specific attention is paid to scalability of solution cost and protocol overhead with network size. Furthermore, we extend the concept of *workload-sensitive locality* introduced by [72, 31] – namely, the algorithms' convergence time and communication overhead depend on the distribution of workload. Hence, whenever the input is not extremely peaky (which happens in most cases), scalable local computation is enough to provide the desired cost. We trade protocol scalability versus the required approximation (Chapter 3 and Chapter 5). Finally, we demonstrate that the quality of

assignment produced by scalable randomized probing is almost as good as that of the best-match assignment with perfect instantaneous information (Chapter 6).

Simulation-based evaluation focuses on average-case performance metrics (either absolute or relative to the optimum), for realistic workloads. Every data point is averaged over 20 to 50 runs. Our mobility simulations employ the widely accepted random waypoint (RWP) model [96] for nearly-random motion. We introduce the Alternating Weighted Waypoint (AWWP) model (Chapter 6, extends [65]), which captures the time-varying non-uniform distribution of mobile users in urban environments more realistically. We use the CRAWDAD set of public traces [2] to model the real campus network topology, as well as mobility patterns of office users. In order to allow for large-scale simulations with thousands of users and access points (Chapter 6), we developed a flow-level WMN simulator, MeshSim [12]. Packet-level simulation tools [4, 13] cannot handle such a scale. MeshSim models the delays incurred to VoIP flows, using a realistic link model [94] and VoIP-specific traffic optimizations in a WMN [58]. We use the common Mean Opinion Score (MOS) metric to evaluate the QoS of VoIP flows.

A prototype mobile WMN implementation (Chapter 7) involved NDIS driver development in the Win32 OS kernel, and multiple design decisions to achieve interoperability with existing technologies. Our implementation is based on the Mesh Connectivity Layer (MCL) software package previously developed at Microsoft Research [5].

# Chapter 3

# Nomadic Service Assignment

Recent advances in network technology, along with the increasing demand for real-time networked applications, are bringing application service providers to deploy multiple geographically dispersed service points, or servers. This trend is expected to further expand with the explosion of new applications and the expansion of services to larger domains. In such settings, a given application session is typically associated with some server. In real-time applications, the association selection is driven by quality of service (QoS) considerations, which may depend, e.g., on the network distance of the user from the server. As many of these applications are becoming increasingly available to mobile users and dynamic user groups, the factors that dictate the server selection can vary with time. For example, due to a user's movement, a server providing optimal QoS at some point may later provide poor QoS, rendering it desirable to migrate the application session from one physical server to another. We therefore believe that many future distributed service infrastructures will employ *nomadic service points*, and will transparently manage such dynamic session assignments.

One important domain where nomadic service points can be exploited to serve mobile users is wireless mesh networks (WMNs) [16, 51, 69]. WMNs provide an increasingly popular solution for Internet access from residential areas with a limited wired infrastructure. These networks are built around multiple stationary wireless routers. Some of them, called access gateways, are wired to the Internet. The mesh access protocol typically routes the traffic of each mobile node through a single access gateway. As the node travels away from its original location, the network delay between it and the gateway grows, and the protocol can re-route the traffic through a different gateway to improve the QoS. For example, a greedy protocol would always route the traffic via the closest gateway. However, this optimization is not always adequate for highly mobile users, which suffer from QoS degradation caused by frequent handoffs. Intelligent nomadic service assignment algorithms can mitigate the tradeoff between access delay and session interruptions.

Server assignment quality also has special importance in collaborative groupware applications like instant messaging, push-to-talk, and massively multiplayer online games, where the impact of a bad association can be magnified with the group's scale. The infrastructure for these applications is typically based on servers that both maintain the application state and act as forwarding proxies. Intuitively, the server should reside close to the group's centroid in order to serve the group best. In groups with a highly dynamic membership, the optimal server selection changes as users join

or leave the group. Thus, there is a tradeoff between the cost of assignment to a suboptimal server (e.g., increased delay) and the cost of state transfer incurred upon the re-assignment.

We study the problem of optimizing the dynamic assignment of sessions to service points. Such a service assignment should balance the tradeoff between connecting sessions to the closest servers at all times, and minimizing the number of session migrations. We capture this tradeoff by assuming two types of service costs: a *setup* cost, incurred whenever the session is assigned to a new server, and a *hold* cost, incurred every unit of time the server is being used. The former reflects one-time expenses like signaling overhead and application state transfer, whereas the latter captures continuous expenses like buffer space, processing power, network latency, and bandwidth. For simplicity, we focus on the case where the setup costs do not vary over time, and are identical for all servers. The hold costs may vary in both aspects. For example, in a mobile WMN, connection transfers are done through wired infrastructure of predictable performance. In this context, the setup cost is fixed, since it does not depend on the location of the source and target gateways. The hold costs, which capture user–gateway distances, are variable.

The *nomadic service assignment* optimization problem is to find a sequence of server assignments that minimizes the total cost. Obviously, we are interested in the *online* version of this problem, in which the service costs are received on the fly. We treat the problem both as a theoretical online optimization problem and as a practical system question. We first handle the generic nomadic service assignment problem, and then examine it more closely in two specific case studies pertaining to specific example domains.

We formally define the problem in Section 3.2. Then, in Section 3.3, we present an *offline* algorithm, `OPT`, which computes the optimal solution assuming that the costs are known in advance. This algorithm's time and space computation complexity is linear in the number of servers $k$ and in the algorithm's duration. While this result has little practical importance, it serves as a baseline for evaluating the online algorithms described in later sections.

In Section 3.4, we study nomadic service assignment as an online optimization problem. A common metric for an online algorithm is its *competitive ratio*, which is the worst-case ratio between the cost produced by the algorithm and the optimal cost. We first prove a lower bound of $k$ on the competitive ratio of any deterministic online assignment algorithm. We then present two simple online algorithms, `DTrack` (deficit tracker) and `CTrack` (cost tracker), parameterized by policies governing *when* transitions happen and *which* server is chosen upon a transition. `DTrack` transitions from its currently assigned server when the session accumulates "significantly more" hold cost than it would have paid had it been assigned to some other server, whereas `CTrack` simply transitions when the session accumulates "enough" hold cost at the currently assigned server. We show that when instantiated with certain policies, these algorithms achieve competitive ratios within a constant factor of the lower bound. Specifically, when using a round-robin (RR) policy to choose the next assignment, `DTrack` achieves a competitive ratio of $2k$, i.e., at most twice as bad as the lower bound, whereas `CTrack` achieves a competitive ratio of $(2 + a)k$, where $a$ is an upper bound on the ratio between the hold and setup costs.

Although, as our lower bound shows, a worst-case cost ratio that is linear in the number of servers is inevitable in the general case, achieving such costs is hardly useful for large-scale services that employ thousands of servers world-wide. From a practical perspective, it is more in-

teresting to examine average costs in common scenarios, and moreover, it is highly desirable for costs not to increase significantly with the number of servers. We address these practical issues in Sections 3.5 and 3.6, via empirical case studies of a WMN with mobile users and an Internet chatroom with dynamic groups, respectively. Interestingly, the competitive versions of `DTrack` and `CTrack`, which achieve the best worst-case costs, are not very promising in practice. However, *opportunistic* versions of these algorithms, which select the next assignment based on current or past offered costs (rather than in a round-robin manner), achieve excellent results. Their costs are at most 50% above the optimum in the average case in the WMN (for a widely accepted random waypoint mobility model, e.g., [96]), and at most 20% above optimal in the groupware service (for uniformly distributed users with a Poisson arrival process). More importantly, this ratio, as well as the total cost, remains almost constant as the problem size scales.

There is a tradeoff between our two algorithms: although `DTrack` achieves better results (lower overall costs), it has a higher computational time complexity, and requires discovering the hold costs of a large number of servers every time unit. In contrast, `CTrack` has a constant per-unit time complexity, and does not need to probe other servers for their costs except when it decides to transition.

In Section 3.5.1, we propose two motion-aware heuristic algorithms, named `TargetAware` and `DirectionAware`. `TargetAware` assumes knowledge of the mobile node's current target and speed, whereas `DirectionAware` only requires the knowledge of the node's current direction, which is used to estimate the target, and speed. These hints can be received either from a higher-level application, or from a positioning system like GPS. Although their lookahead window is quite small (the node's next target), both motion-aware algorithms yield significant cost improvements. Their costs are typically within 10% of the optimal, and exhibit perfect scalability.

## 3.1 Related Work

Handoff optimizations in mobile systems have been extensively studied since the early 1990's, mostly in the context of cellular networks with the advent of the GSM standard [49, 86, 88]. This research targeted increasing network capacity as the primary goal. Handoffs in cellular systems are driven by physical metrics, like signal strength and transmission power, and are handled at the link layer. They cannot be avoided when user location changes significantly, and optimizing their cost is a secondary design goal (e.g, [43]). Our work is fundamentally different, because we consider the network layer and above. In this context, handoffs are optional, they improve the QoS in the long run, but their cost is substantial. For example, migrating a host connection between two WMN gateways can affect packet delivery order, and temporarily degrade the TCP performance.

Initial mesh networking research mostly focused on problems that are specific to fixed wireless, e.g., defining routing metrics [51], exploiting the broadcast nature of the medium [32], and harnessing multiple radio interfaces through smart cross-layer design [17]. More recently, Amir et al. presented a design and implementation of a prototype WMN with mobility support [19]. The algorithms presented in this work can be integrated into such a system for inter-gateway handoff decisions. Lavi et al. [73] proposed employing an overlay service network for supporting groupware in mobile networks. Their architecture suggested associating every mobile user with

the closest server and efficiently maintaining the group membership information between multiple servers. In contrast with this approach, we focus on applications (including possibly groupware) that associate a session with a single server.

The problem of dynamic session management was studied in the context of routing virtual circuits in mobile ATM networks [28], with a similar model of setup and hold costs. However, these costs were defined per link, and the algorithm had to decide whether to retain or to release a redundant link. This model allowed reusing part of the links after the re-routing, thus allowing for lower total costs than in our model where no reuse is possible. Indeed, their algorithms exhibit better competitive behavior than the best possible for nomadic service assignment.

Nomadic service assignment is closely related to the classical *metrical task system* (MTS) problem [34]. Since the introduction in the early 1990's, this problem keeps receiving considerable attention in the theory community (e.g., see [26, 55] for some new approaches). In this context, there is a set of $k$ states, and a matrix of inter-state transition costs (the cost of a self-transition is zero). A schedule for a sequence of tasks is a sequence of states in which these tasks are processed. The cost of a schedule is the sum of all task processing (hold) and transition (setup) costs. For symmetric cost matrices subject to the triangle inequality, there is a deterministic online algorithm with a competitive ratio of $2k - 1$, and this bound is tight. Nomadic service assignment closely resembles a special case of this problem with uniform transition costs, except that in our problem, the initial assignment always incurs a cost. However, the online MTS algorithm [34] makes use of the entire history of setup and hold costs until the scheduling decision, which makes it impractical to implement. We use a very different algorithmic technique, which requires $O(k)$ operations per decision, regardless of the history length. In a specific setting of a WMN with mobile users, in which the hold costs are defined as user–gateway distances, the computation overhead of our algorithm can be further reduced by an order of magnitude, through the use of spatial data structures.

Mobile user location, a basic service in wireless networks, is a prerequisite for any network optimization task, including dynamic session management. Multiple papers treated the mobile tracking problem in an online fashion, capitalizing on the tradeoff between the accuracy in location estimation and the number of updates [22, 23, 24, 27]. Part of these works consider stochastic motion [22, 23] while the others make no assumptions on the mobility model. Some algorithmic techniques employed in our work bear some resemblance to these papers, since we address the same competitive analysis framework. However, our problem belongs to a distinct application domain, and pursues different optimization goals.

Optimal center location for a group of users is an instance of the well-studied *facility location problem* [78], which given a set of facility locations and a set of customers in a metric space, chooses which customers should be served from which facilities so as to minimize the total service cost. Facility location was studied as an online problem [77], and was used for various applications, including optimizing the delivery of Web content in CDN's [67, 87], maintenance of mobile centers in ad-hoc networks [30] and adaptive server selection in online games [74]. The problem differs from ours in that multiple facilities are used per group, and the online algorithm is allowed to *add* facilities over time, instead of migrating sessions among existing ones.

## 3.2 System Model

Consider an application session that can be hosted by any one of $k$ servers $\mathcal{S} = \{s_0, ..., s_{k-1}\}$. The session is assigned to some server at the beginning of the session but can be re-assigned to a different server at each discrete time slot.

There are two types of non-negative costs charged for the session: a *setup cost* that is paid when the session is assigned to a new server, including the initial one, and a *hold cost*, paid for each time slot the session is assigned to some server. From a session's perspective, different servers offer different costs at a given time slot, and may also change them at the beginning of each slot. We denote the setup cost offered by server $s$ at time $t$ by $\texttt{setup}(s,t)$ and the hold cost by $\texttt{hold}(s,t)$.

The *assignment schedule* $\sigma(t)$ in a time interval $\mathcal{I}$ is a function, $\sigma : \mathcal{I} \rightarrow \mathcal{S}$, which assigns the session to server $s \in \mathcal{S}$ at each discrete time $t \in \mathcal{I}$. For convenience, we define $\sigma(t) = \perp$ for $t \notin \mathcal{I}$. We define the set of *transitions* on an interval $\mathcal{I}$ as

$$\mathcal{T}(\sigma, \mathcal{I}) = \{t \mid t \in \mathcal{I} \ \wedge \ \sigma(t) \neq \sigma(t-1)\}.$$

In particular, the initial assignment is also considered a transition.

The assignment schedule $\sigma$ on an interval $[t_1, t_2)$ incurs a total hold cost

$$\texttt{hold}(\sigma, [t_1, t_2)) \triangleq \sum_{t=t_1}^{t_2-1} \texttt{hold}(\sigma(t), t),$$

a total setup cost

$$\texttt{setup}(\sigma, [t_1, t_2)) \triangleq \sum_{t \in \mathcal{T}(\sigma, [t_1, t_2))} \texttt{setup}(\sigma(t), t),$$

and a total overall cost

$$\texttt{cost}(\sigma, [t_1, t_2)) \triangleq \texttt{setup}(\sigma, [t_1, t_2)) + \texttt{hold}(\sigma, [t_1, t_2)).$$

The *optimal nomadic service assignment* problem for interval $[0, T)$ is to compute an assignment schedule $\sigma^*$ that minimizes $\texttt{cost}(\sigma^*, [0, T))$.

The presence of positive setup costs is what makes the problem nontrivial. Otherwise, the session would always associate with the server that offers the minimum hold cost. Hence, we always consider positive setup costs.

## 3.3 An Optimal Offline Algorithm

In this section, we describe an optimal *offline* algorithm for the assignment problem, i.e., assuming that the setup and hold cost functions are known in advance. The algorithm is linear-time in the interval length $T$ and the number of servers $k$.

We first identify the structure of the optimal solution $\sigma^*$. Let $\sigma^*_{s,t} : [t, T) \rightarrow \mathcal{S}$ be a lowest cost schedule among those in which $s$ is the initial assignment, that is, $\sigma^*_{s,t}(t) = s$. We observe that if

$\sigma^*_{s,t}(t+1) = s'$, then

$$\texttt{cost}(\sigma^*_{s,t}, [t+1, T)) = \texttt{cost}(\sigma^*_{s',t+1}, [t+1, T)).$$

In other words, the cost of an optimal schedule for $[t+1, T)$ that assigns $s'$ at $t+1$ is identical to the cost of the $[t+1, T)$-suffix of the optimal schedule for $[t, T)$ with the same assignment. Otherwise, the global optimality is violated. If $s' = s$, then $\texttt{setup}(s', t+1)$ does not contribute to $\texttt{cost}(\sigma^*_{s,t}, [t, T))$.

The problem can be represented as a layered directed acyclic graph. Node $i$ in layer $t$ stands for $\sigma^*_{s_i,t}$, for $1 \leq i \leq k,\ 0 \leq t \leq T$. There is an edge between every pair of nodes $(i, t)$ and $(j, t+1)$, which represents a possible transition from $s_i$ to $s_j$ at time $t$. The cost of this edge is $\texttt{hold}(s_j, t+1)$ if $i = j$, and $\texttt{hold}(s_j, t+1) + \texttt{setup}(s_j, t+1)$ otherwise. The optimal solution's cost is the weight of the shortest path in the graph. While this weight can be computed in linear time in the number of edges, i.e., $O(k^2T)$, the time complexity can be optimized to $O(kT)$, by exploiting the optimal solution's structure, as we now explain.

We define the *tail contribution* function for $t < T$ as follows:

$$\texttt{tail}(s, s', [t, T)) \triangleq \begin{cases} \texttt{cost}(\sigma^*_{s,t}, [t, T)) - \texttt{setup}(s, t) & \text{if } s = s' \\ \texttt{cost}(\sigma^*_{s',t}, [t, T)) & \text{otherwise} \end{cases}$$

Then, $\texttt{cost}(\sigma^*_{s,t}, [t, T))$ for $t < T$ can be expressed as

$$\texttt{cost}(\sigma^*_{s,t}, [t, T)) = \texttt{setup}(s, t) + \texttt{hold}(s, t) + \min_{s' \in \mathcal{S}} \texttt{tail}(s, s', [t+1, T))$$

We define $\texttt{tail}(s, s', [T, T)) \triangleq \texttt{cost}(\sigma^*_{s,t}[T, T)) \triangleq 0$. For $t < T$ we get:

$$\begin{aligned} &\texttt{cost}(\sigma^*_{s,t}, [t, T)) = \\ &\quad \texttt{setup}(s, t) + \texttt{hold}(s, t) + \\ &\quad \min(\min_{s' \in \mathcal{S}} \texttt{cost}(\sigma^*_{s',t+1}, [t+1, T)), \texttt{cost}(\sigma^*_{s,t+1}, [t+1, T)) - \texttt{setup}(s, t+1)). \end{aligned}$$

An optimal solution can be computed through dynamic programming [48] using the above recurrence. The algorithm employs a two-dimensional table $\texttt{Table}[1..k, 0..T]$ where an entry $\texttt{Table}[s, t]$ holds the value of $\texttt{cost}(\sigma^*_{s,t}, [t, T))$ and the identity of $s' = \sigma^*_{s,t}(t+1)$. The table is computed column by column from $T-1$ down to 0. Column $T$ is initialized by zeroes. During the processing of column $t$, the value of

$$\min_{s' \in \mathcal{S}} \texttt{cost}(\sigma^*_{s',t}, [t, T)) = \min_{1 \leq s \leq k} \texttt{Table}[s, t]$$

is computed once to be used in computing all entries of column $t-1$. After the whole table is filled, the overall optimal cost is computed as

$$\texttt{cost}(\sigma^*, [0, T)) = \min_{0 \leq s \leq k-1} \texttt{Table}[s, 0],$$

and an optimal schedule is built by tracing the algorithm's choices through the columns $0 \ldots T-1$.

The computation of a single table entry requires a constant number of operations thanks to the pre-computation of the previous column's minimum cost, and therefore, the algorithm's time complexity is $O(kT)$. The space complexity is also $O(kT)$ – the table's size.

## 3.4 Online Server Assignment

In a realistic scenario, the costs are not known in advance. This is especially true for the hold cost, which can reflect dynamic network conditions like user mobility, group membership, etc. In this section, we study server assignment as an *online* optimization problem [33]. The cost for a time slot becomes known at the beginning of that slot, and the algorithm must produce a new scheduling decision. We restrict ourselves to the case where the setup costs are identical and constant, that is, $\texttt{setup}(s,t) = C$ for all $s$ and $t$, whereas the hold costs are dynamic. We denote the schedule produced by the optimal algorithm OPT as $\sigma^*$, and the schedule produced by an online algorithm ALG as $\sigma$.

The *competitive ratio* is the common performance measure for online algorithms. In our problem, an online algorithm ALG is called $r(\texttt{ALG})$-*competitive* if there is a constant $\delta$ such that for *all* finite intervals $\mathcal{I}$ and for *all* setup and hold costs

$$\texttt{cost}(\sigma, \mathcal{I}) \leq r(\texttt{ALG}) \cdot \texttt{cost}(\sigma^*, \mathcal{I}) + \delta.$$

The rest of this section is structured as follows. In Section 3.4.1, we show that no deterministic online algorithm can achieve a competitive ratio better than $k$. In Section 3.4.2, we present a generic online algorithm called DTrack (deficit tracker). A version of this algorithm termed DTrack$-RR$, that is, DTrack with round-robin selection of server assignments, achieves a competitive ratio of $2k$ with a certain parameter choice. DTrack needs to track the cost of up to $k$ servers every time slot, and may thus have a large control message overhead in a distributed implementation. In Section 3.4.3, we present a simple and efficient algorithm called CTrack (cost tracker), which yields a competitive ratio of $(2+a)k$ for a certain parameter choice, assuming that a server's per-slot hold cost never exceeds $aC$. The competitive version of CTrack, called CTrack$-RR$, probes the cost of only one server every slot. In Section 3.4.4, we present opportunistic versions of these algorithms, called CTrack$-F$, DTrack$-F$, and DTrack$-B$, which are not competitive but greatly improve the cost in the *average* case, and achieve good scalability.

### 3.4.1 A Lower Bound of $k$ on the Competitive Ratio

**Theorem 3.1.** *No deterministic server assignment algorithm can achieve a competitive ratio of less than $k$.*

*Proof.* Consider $k$ symmetric servers that offer the same setup cost $C > 0$ and a zero hold cost each at $t = 0$, that is, $\texttt{hold}(s_i, 0) = 0$. Consider the following simple adversary strategy against any deterministic algorithm ALG. When ALG connects to $s_i$ at time $t$, set $\texttt{hold}(s_i, t+1) = 1$. When ALG disconnects from the server at time $t'$, set $\texttt{hold}(s_i, t'+1) = 0$. Regardless of what the online

algorithm is, it will have to transition to a different server at some point if it wishes to remain competitive. This process continues until $k-1$ moves happen. At this point, the adversary stops the run.

If ALG has visited every server exactly once, let $s^*$ be its last assignment. Otherwise, there exists a server $s^*$ that has never been picked by ALG. The best offline algorithm, OPT, assigns the session to server $s^*$ at time 0 and never changes the assignment.

OPT pays only $C$ for the initial setup, whereas ALG pays $kC$ for setup and zero or more for hold. Therefore, $r(\texttt{ALG}) \geq \frac{kC}{C} = k$, and the algorithm's competitive ratio has a lower bound of $k$. □

### 3.4.2 DTrack - a $2k$-Competitive Online Algorithm

We present a simple online algorithm called DTrack (*deficit tracker*). It is parameterized by factor $\alpha \geq 0$, which controls *when* transitions happen, and a subroutine nextchoice(), which controls *which* server is chosen upon transition. In this section, we focus on a $2k$-competitive version of DTrack, called DTrack$-RR$, obtained by a round-robin nextchoice() policy. Its pseudocode appears in Figure 3.1.

We begin with some definitions. The *deficit* between the servers $s$ and $s'$ during the interval $[\tau, t)$ is the greatest total difference between the total hold costs in a suffix $[t', t)$:

$$\texttt{def}(s, s', [\tau, t)) \triangleq \max_{\tau \leq t' \leq t-1} (\texttt{hold}(s, [t', t)) - \texttt{hold}(s', [t', t))).$$

Let us denote the current assignment by $s_c$. A server $s$ for which $\texttt{def}(s_c, s, [\tau, t+1)) > 0$ is called a *leader* at time $t$.

The algorithm's code maintains the following variables: $t$ is the current time, $\tau$ is the last transition's time, $c$ is the current assignment's id, Leaders is the set of the current leaders' ids, and Def is the vector of deficit values between $s_c$ and the other servers. The algorithm maintains that at time $t$, $\texttt{Def}[s] = \texttt{def}(s_c, s, [\tau, t+1))$.

DTrack maintains an invariant that the deficit between $s_c$ and any other server $s$ never exceeds $\alpha C$. Initially, DTrack makes an assignment to the server with the minimal hold cost. It then keeps tracking the deficit versus the other servers. A server becomes a leader when it offers a smaller hold cost than $s_c$, and stops being one when the cumulative deficit value becomes negative. Since the hold costs are published at the beginning of each time slot, DTrack makes its decision using a single-slot lookahead. When some server is about to accumulate significantly less hold cost than the current choice (a deficit of above $\alpha C$), the algorithm changes its assignment. Due to the lookahead mechanism, the update() procedure that updates the deficit values is invoked twice at transition times. First, for the current choice in order to decide whether to transition, and then for the new choice, which does not necessarily offer the best hold cost, hence the new deficit must be computed.

In the instance of DTrack we consider now, termed DTrack$-RR$, nextchoice() selects the next assignment in a round-robin way, among servers whose a-priori deficit versus any other server (that is, the hold cost gap) does not exceed $\alpha C$.

The intuition behind DTrack is that the current server must be provably bad (costing $\alpha C$ more than the best) in order to change the choice, and the next server must also *not* be provably bad (not

```
1:  Initialization:
2:     t ← 0
3:     c ← i  s.t. hold(s_i, 0) = min_{s∈S} hold(s, 0)
4:     reset()

5:  Every time slot do
6:     update()
7:     if (Def[s] > αC) for some s ∈ Leaders then
8:        nextchoice()
9:        reset()
10:    t ← t + 1

11: procedure reset()
12:    τ ← t
13:    Leaders ← ∅
14:    for all s ≠ s_c do
15:       Def[s] ← 0
16:    update()

17: procedure update()
18:    for all s s.t. s ∉ Leaders ∧ hold(s_c, t) > hold(s, t) do
19:       Def[s] ← 0
20:       Leaders ← Leaders ∪ {s}
21:    for all s ∈ Leaders do
22:       Def[s] ← Def[s] + hold(s_c, t) − hold(s, t)
23:       if (Def[s] < 0) then
24:          Leaders ← Leaders \ {s}

25: procedure nextchoice() /*RR version*/
26:    repeat
27:       c ← (c + 1) mod k
28:    until hold(s_c, t) − min_{s∈S} hold(s, t) ≤ αC
```

Figure 3.1: DTrack$-RR$ **- an Online Algorithm for Server Assignment.**

costing $\alpha C$ more than any other server). When instantiated with $\alpha = 0$ (this algorithm is termed `Greedy`), `DTrack` immediately changes the assignment when some other server offers a better hold cost. At the other extreme, when $\alpha = \infty$, it never changes its initial assignment. It is clear that the algorithm is not competitive in either of these extreme cases.

In Section 3.7.1, we provide a detailed competitive analysis of `DTrack`$-RR$, and get the following result:

**Theorem 3.2.** *The competitive ratio of* `DTrack`$-RR$ *is bounded as follows:*

$$\begin{array}{ll} r(\texttt{DTrack}-RR) < k(1+\frac{1}{\alpha}) & 0 < \alpha \leq 1 \\ r(\texttt{DTrack}-RR) < 1 + (k-1)\alpha + k & \alpha \geq 1 \end{array}$$

**Corollary 3.1.** *For* $\alpha = 1$, `DTrack`$-RR$ *achieves a competitive ratio of* $2k$.

The crux of the algorithm's competitiveness lies in the round-robin selection policy, and can be informally explained as follows. If we consider a schedule $\sigma$ by `DTrack`$-RR$ that *overtakes* (that is, either leaves or skips) every server while the optimal schedule $\sigma^*$ does not change its assignment $s^*$, then $\sigma$ overtakes $s^*$ exactly once. This overtake implies that the total hold cost incurred by $\sigma^*$ during the interval exceeds $\alpha C$. The total hold cost incurred by $\sigma$ exceeds the one incurred by $\sigma^*$ by at most $(k-1)\alpha C$. The subtle point in this proof is the deficit bookkeeping, because upon transition the hold cost lookahead affects the assignment but does not contribute to the total hold cost. The total setup cost incurred by $\sigma$ during this period is at most $kC$, whereas $\sigma^*$ pays $C$ upon the assignment to $s^*$. A careful analysis of the worst-case ratio between the total costs concludes the proof.

In order to illustrate this result, consider the adversary strategy from Theorem 3.1, assuming $\alpha = 1$ and an integral $C$. In this scenario, `OPT` pays $C$ for the initial setup, whereas `DTrack`$-RR$ pays $kC$ for setup and $kC$ for hold (accumulating a deficit of $C$ before each transition). Hence, the exact competitive ratio of $2k$ is achieved.

### 3.4.3 CTrack - an Efficient Online Algorithm

At each slot, `DTrack` checks the hold cost of every server, which results in linear time complexity per slot. Since the number of servers can be large, sublinear complexity is desirable to achieve efficiency of communication in a distributed implementation.

We now present a simple online algorithm `CTrack` (*cost tracker*), which achieves constant computation time complexity at the expense of a weaker competitive guarantee, under the assumption of an upper bound on the ratio between the hold and the setup costs. `CTrack` is also parameterized by a factor $\alpha$ and a subroutine `nextchoice`$()$. Initially, it assigns the server with the minimal hold cost. The assignment changes when the total hold cost since the last transition exceeds $\alpha C$ (e.g., for $\alpha = 0$, it transitions every time slot). The rationale behind this policy is controlling the fraction of the setup cost in the total cost. It only requires receiving the hold cost of the *current* assignment every time slot, which leads to constant per-slot time complexity.

In Section 3.7.2, we provide a detailed competitive analysis of `CTrack`$-RR$, the round-robin version of `CTrack`, and get the following result:

**Theorem 3.3.** *If* $\texttt{hold}(s,t) \leq aC$ *for all* $s$ *and* $t$*, then* $r(\texttt{CTrack}{-}RR) < (2+a)k$ *for* $\alpha = 1$.

### 3.4.4 Opportunistic Heuristics

While the competitive ratio is an accepted metric for measuring the worst-case performance of an online algorithm, the average-case performance is more important in practice. An algorithm that behaves $2k$ times worse than the optimal solution in the average case is impractical in systems accommodating thousands of servers.

In this section, we introduce opportunistic versions of `CTrack` and `DTrack`, in which `nextchoice`() selects an assignment that is locally optimal for some metric, instead of the round-robin traversal. This approach exploits the well-known locality principle to achieve good performance in typical scenarios. Note that although locality is common in practice, it is not a property that holds in all possible runs, and hence, the cost of using opportunistic selection policies is that they yield worse competitive ratios than the round-robin ones.

In the *forward* heuristics $\texttt{DTrack}{-}F$ and $\texttt{CTrack}{-}F$, `nextchoice`() picks the server with the current minimal hold cost. The *backward* heuristic $\texttt{DTrack}{-}B$ augments $\texttt{DTrack}{-}RR$'s selection policy with the following rule: the deficit between the next choice and the previous assignment is greater than $\beta C$ for some $-\infty \leq \beta \leq \alpha$. Using any $\beta > 0$ allows the algorithm to choose the next server from those that presented good behavior since the last transition. For $\beta = -\infty$, the resulting algorithm is $\texttt{DTrack}{-}RR$. For $\beta = 0$, $\texttt{DTrack}{-}B$ chooses the next server from the leader set. For $\beta = \alpha$, it selects a leader that triggered the transition. Theorem 3.2 can be generalized to describe $\texttt{DTrack}{-}B$'s worst-case behavior (the proof appears in Section 3.7.3):

**Theorem 3.4.** *The competitive ratio of* $\texttt{DTrack}{-}B$ *is bounded as follows:*

$$\begin{array}{ll} r(\texttt{DTrack}) < k(1+\frac{1}{\alpha}) & \alpha \leq 1 \text{ and } \beta \leq 0 \\ r(\texttt{DTrack}) < 1 + (k-1)\alpha + k & \alpha \geq 1 \text{ and } \beta \leq \alpha - 1 \\ r(\texttt{DTrack}) < 1 + \frac{(k-1)\alpha + k}{\alpha - \beta} & \max(0, \alpha - 1) \leq \beta \leq \alpha \end{array}$$

**Corollary 3.2.** *For* $\alpha = 1$ *and* $\beta \leq 0$, $\texttt{DTrack}{-}B$ *achieves a competitive ratio of* $2k$.

The worst-case competitive ratio achieved by $\texttt{DTrack}{-}F$ and $\texttt{DTrack}{-}B$ with $\alpha = \beta$ is not limited by the problem size $k$ (see Section 3.7.4 for the proof):

**Theorem 3.5.** *The competitive ratio of* $\texttt{DTrack}{-}F$ *and* $\texttt{DTrack}{-}B$ *with* $\alpha = \beta$ *is* $\Omega(C)$.

## 3.5 Case Study: Mobile Users in a WMN

In this section, we study nomadic service assignment in an urban WMN environment. The results of the optimal algorithm `OPT` are used as a comparison baseline. For each algorithm `ALG`, we measure its cost as well as *performance ratio*, which is the average ratio between the total costs incurred by `ALG` and `OPT` during multiple runs. We average over 20 simulations, each $10{,}000$ slots long. This metric is analogous to the competitive ratio, the theoretical worst-case metric.

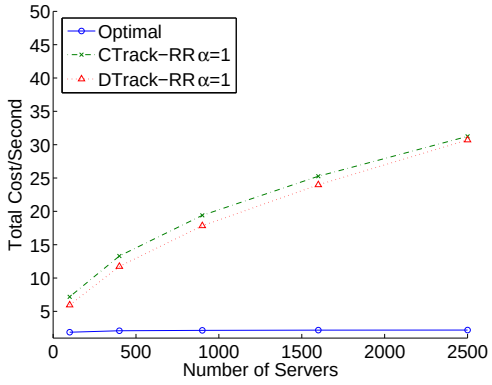



Figure 3.2: $\texttt{CTrack}-RR$ **and** $\texttt{DTrack}-RR$ **with** $\alpha = 1$ **do not scale well with the network size.**

The simulated network spans a square grid with uniformly distributed wireless routers. The number of routers that populate a $1000\text{m} \times 1000\text{m}$ grid is 100, that is, a single router spans an average area of $100\text{m} \times 100\text{m}$. A mobile node moves using the random waypoint mobility model [96]. The node uniformly chooses the destination and moves toward it at a constant urban driving speed of 10 m/sec (36 km/hour). The time slot is one second.

We assume that the wireless infrastructure is the main bottleneck, whereas the gateway resources are abundant, and hence, the end-user QoS is not affected by the congestion among multiple connections. The hold cost between mobile node $n$ and router $r$ is defined as $\frac{d(n,r)}{100}$, i.e., a normalized Euclidean ($L_2$) distance. Under these parameters, the average hold cost offered by the closest router is roughly 0.5. The setup cost is 50.

Our main interest is in the scalability of the online solutions, i.e., how the total cost per second and the performance ratio are affected as the problem size grows. For this purpose, we gradually increase the grid size from $1000\text{m} \times 1000\text{m}$ to $5000\text{m} \times 5000\text{m}$, and correspondingly increase the number of routers from 100 to 2500, keeping the router density fixed. We study the performance of different versions of `CTrack` and `DTrack` with different selections of $\alpha$, $\beta$, and $\texttt{nextchoice}()$.

Our first goal is to study the performance of $\texttt{CTrack}-RR$ and $\texttt{DTrack}-RR$ with $\alpha = 1$, which have the best proven worst-case ratios. Figure 3.2 shows that both algorithms scale poorly with the network size (their costs grow approximately as $\sqrt{k}$, whereas `OPT`'s cost remains nearly constant). This is intuitive, since the round-robin selection policy tends to assign a session to a random server, and the average distance grows as $O(\sqrt{k})$.

$\texttt{DTrack}-B$ requires selecting the $\beta$ parameter for a given $\alpha$. Contrary to the worst-case analysis, our results show that the algorithm's performance improves as $\beta$ becomes closer to $\alpha$. Figure 3.3 depicts the results for $\alpha = 1$. The curves for all $\beta$ values from 0.2 to 1 are barely distinguishable. Hence, a good worst case ratio can be guaranteed by selecting small $\beta$ values without compromising the average performance by much (for example, for $\alpha = 1$ and $\beta = 0.2$, the competitive ratio is bounded by $2.5k - 0.25$).

Figures 3.4(a) and 3.4(b) depict the results of simulating the opportunistic algorithms `Greedy`, $\texttt{CTrack}-F$, $\texttt{DTrack}-F$, and $\texttt{DTrack}-B$ with $\alpha = 1$ and $\beta = 1$. The performance curves of $\texttt{CTrack}-F$ and $\texttt{DTrack}-F$ are almost indistinguishable. The algorithms' performance ratios re-

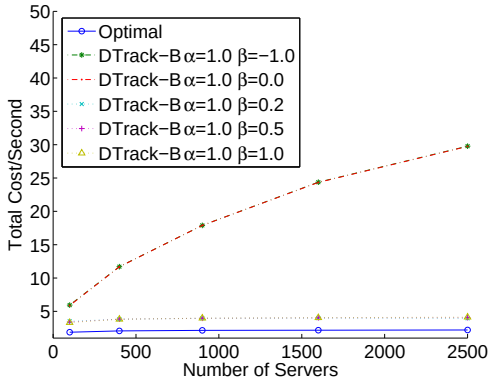


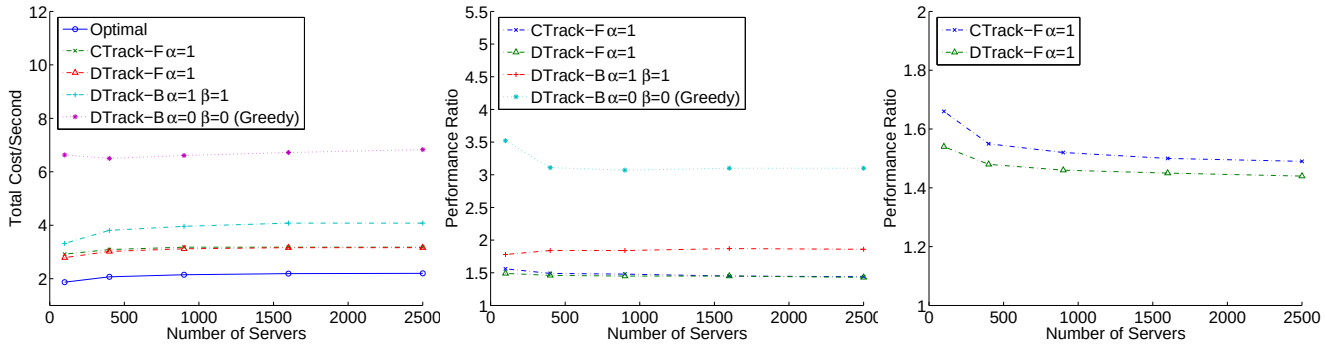

Figure 3.3: **Choosing a $\beta$ value for DTrack$-B$ with $\alpha = 1.0$. The values between 0.2 and 1 exhibit very close behavior and scale well with the network size.**

main constant as the problem scales – around 50% above the optimum. The total cost per second also remains constant, since OPT itself is very scalable. Greedy, which takes the opportunistic heuristic to the extreme, exhibits a weaker performance ratio (more than three times the optimum) although it scales well. In this setting, Greedy's reasonable behavior can be explained by the moderate speed (hence, the hold cost changes are slow), and by the moderate setup cost (hence, the penalty for making a wrong decision is limited). The fact that DTrack$-F$ consistently produces better results than DTrack$-B$ can be explained by the motion's nature. Since the motion is random, the deficit values exhibit poor locality. The result could have been different had the motion happened around a small number of stationary points (home, office, cab station etc).

Figure 3.4(c) depicts the results of the same experiment with an average simulated speed 25 m/sec (90 km/hour). In this setting, DTrack$-F$ starts producing a consistently lower total cost (by 5-6%) than CTrack$-F$. This happens because at higher speeds, the hold cost changes faster, and the total cost becomes a worse transition indicator than the deficit. This phenomenon cannot be further magnified at reasonable driving speeds, but can be clearly demonstrated in a different application (Section 3.6). As expected, Greedy performs worse at higher speeds (above five times the optimum).

Further simulations (Figure 3.5) show that $\alpha$ values between 0.5 and 2.0 exhibit nearly the same average-case performance.

DTrack's computation overhead can be significantly improved in a WMN environment since the hold cost monotonically increases with distance. Therefore, maintaining the deficit values requires accessing the hold costs of the servers that are closer to the user than the current assignment, as well as the servers that already have a positive deficit. This can be achieved by using data structures that support efficient nearest neighbor queries in a multidimensional space like KD-trees or R-trees [78]. Figure 3.6 depicts the percentage of hold costs that need to be accessed by DTrack$-F$ and DTrack$-B$ with $\alpha = \beta = 1$. We can see that the fraction of hold costs that must be accessed to maintain the positive deficit values is very low.

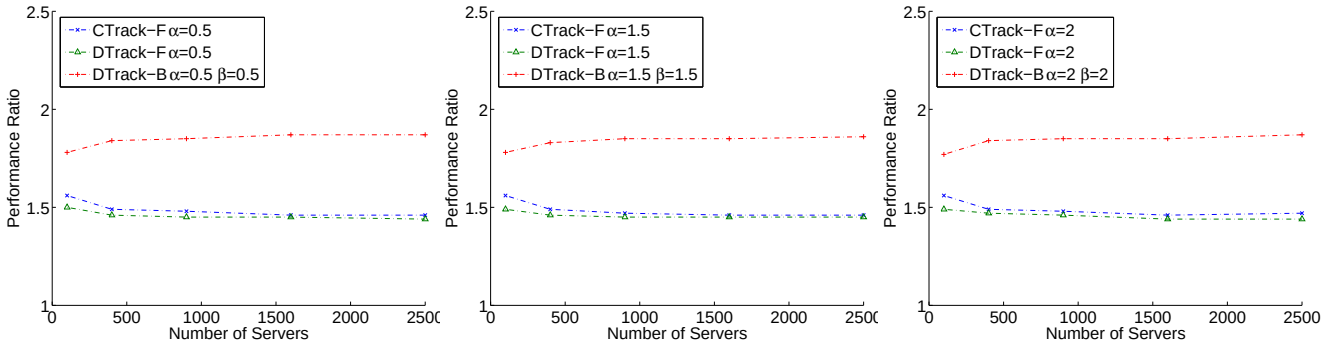



(a) Average total cost/second, speed=10 m/s

(b) Performance ratio, speed=10 m/s

(c) Performance ratio, speed=25 m/s

Figure 3.4: **Scalability of** CTrack$-F$**,** DTrack$-F$**, and** DTrack$-B$ **in a WMN with mobile users,** $\alpha = 1.0$ **and** $\beta = 1.0$**.**




(a) Performance ratio, $\alpha = 0.5$

(b) Performance ratio, $\alpha = 1.5$

(c) Performance ratio, $\alpha = 2.0$

Figure 3.5: **Scalability of** CTrack$-F$**,** DTrack$-F$**, and** DTrack$-B$ **in a WMN with mobile users, speed=10 m/s, with different** $\alpha$ **values.**

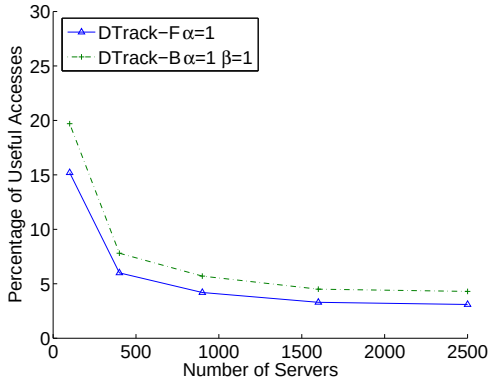



Figure 3.6: **Percentage of useful hold cost accesses per second for** DTrack$-F$ **and** DTrack$-B$ **with** $\alpha = 1$ **and** $\beta = 1$**.**

### 3.5.1 Motion-Aware Heuristics

In order to achieve a better practical performance, we employ two simple online heuristics tailored specifically to the mobile user environment. These heuristics exploit the near-term motion pattern, and therefore can project the hold costs better than DTrack, which has only a single-slot lookahead.

The first heuristic is called TargetAware. It requires information regarding the mobile node's current target and speed. This target information can be provided from a higher-level system, e.g., a car navigation system, where the user can indicate the current status (e.g., "driving home"). TargetAware is informed every time the mobile node changes its target, and applies OPT as a subroutine in order to compute the assignment schedule until the next target is reached. Every time the target changes, TargetAware selects the best of two choices: running OPT with the fixed first assignment that is identical to the current one (i.e., no setup cost is incurred for it), or letting OPT pick an arbitrary first assignment.

If the target information is not available, a mobile node equipped with a positioning system (e.g., GPS) can use the direction information provided by it. In this context, we propose the second heuristic that is called DirectionAware. It receives information about the grid size as well as the mobile node's estimated current direction and speed, which are received upon the node's direction changes. The algorithm projects the next target as the clipping point of its current trajectory and the grid's boundary, and applies TargetAware as a subroutine.

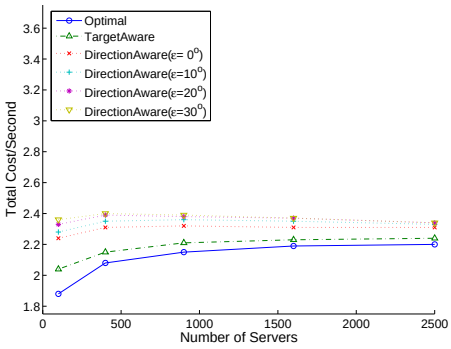
Figure 3.7 depicts the scalability of both motion-aware heuristics, in the same environment as the previous simulation. Both TargetAware and DirectionAware are clearly superior to CTrack$-F$ and DTrack$-F$. Their performance ratios are less than 10% and 18% above the optimum, respectively. As expected, TargetAware performs slightly better than DirectionAware because it uses an accurate motion forecast. The motion-aware heuristics scale even better than OPT because their lookahead window grows as the grid scales up.

We also evaluated DirectionAware's capability to handle inaccurate predictions, by supplying it with direction estimates that are normally distributed around the real direction with variance $\varepsilon$. The values of $\varepsilon$ ranged from $0^o$ (exact prediction) to $30^o$ (Figure 3.7). As expected, the algorithm's performance ratio grows with $\varepsilon$. However, this growth is limited by 25% above the optimum, i.e., only 7% above the algorithm with a perfect direction forecast. Therefore, DirectionAware is quite tolerant to moderately inaccurate direction estimates.

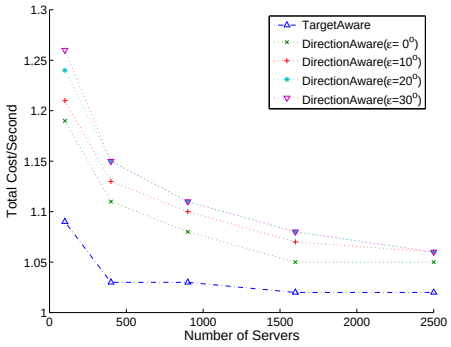
Note that both heuristics perform very well despite their small lookahead window. In the context of the offline assignment problem, this means that a practically good solution can be achieved with constant space complexity, without the need to capture the entire data stream before running the dynamic programming algorithm.

## 3.6 Case Study: Wide-area Chatroom Service

The second environment studied is an Internet-scale groupware application service [73, 74], e.g., chat. The service overlay network consists of 100 servers uniformly selected among the nodes of a random network. Groups of users run a chatroom application, where each group is assigned to a single server. The users are stationary, and their locations are uniformly distributed in the




(a) Average total cost/second, speed=10 m/s




(b) Performance ratio, speed=10 m/s

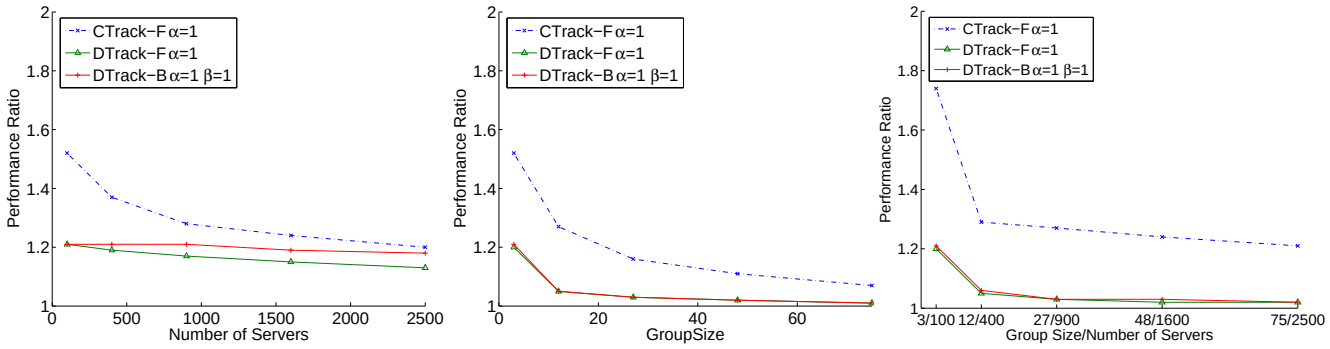
Figure 3.7: **Scalability of the motion-aware algorithms in a WMN with mobile users.**

network. The user arrival to a group is described by a Poisson process with a mean of $\lambda$, and the membership lifetime is distributed exponentially with a mean of $T$ (that is, the average number of users in a group is $\lambda T$). The hold cost between group $G$ and server $s$ is proportional to the maximal network distance between the server and some node in the group, which reflects the application's buffer space requirements affected by the maximal delay. In this context, the server is seen as the group's *center*, and the maximal distance is the group's *radius*. We study the same instances of $\texttt{CTrack}-F$, $\texttt{DTrack}-F$, and $\texttt{DTrack}-B$ as in Section 3.5 (that is, $\alpha = \beta = 1$). We explore the algorithms' scalability with both the number of servers and the average group size.

In the first experiment, we increase the number of servers (in parallel with the network's size) from 100 to 2500, without increasing the number of users. We set $\lambda = 0.1$ users/second and $T = 30$ seconds, yielding three users in the chatroom on average. Figure 3.8(a) depicts the simulation results. Both versions of DTrack are within 15-20% above the optimal cost. $\texttt{DTrack}-F$ consistently outperforms $\texttt{CTrack}-F$ because individual join or leave events in a small group trigger fast changes in the hold costs. This is the same phenomenon that happens in WMNs at high speeds (Figure 3.4(c)), but it is more significant since the hold cost changes are faster.

In the second experiment, depicted in Figure 3.8(b), we scale the average group size up from three to 75 (a large-scale conference) by increasing both $\lambda$ and $T$. The network size is not changed. Both versions of DTrack exhibit a performance ratio of under 5% above the optimum for groups with more than ten members, and converge to the optimal cost as the group scales. This happens because in dense groups, individual join and leave events do not considerably affect the group radius. Therefore, the algorithms perform fewer transitions.

Finally, we study the algorithms' scalability to large groups in large networks. For this purpose, we gradually increase both the number of servers and the group size by the same factor. The results depicted in Figure 3.8(c) show that when the number of servers grows from 400 to 2500 and the number of users grows from 12 to 75, the performance ratios of both versions of DTrack remain constant at less than 5% above the optimum, whereas the performance ratio of $\texttt{CTrack}-F$ also remains constant but exceeds the optimum by 30%.




(a) Scalability with number of servers (b) Scalability with group size (c) Scalability with number of servers/group size

Figure 3.8: **Scalability of** $\texttt{CTrack}-F$**,** $\texttt{DTrack}-F$ **and** $\texttt{DTrack}-B$ **in a wide-area chatroom application service,** $\alpha = 1.0$ **and** $\beta = 1.0$**.**

## 3.7 Analysis

### 3.7.1 A Competitive Analysis of DTrack-RR

In this section, we give a competitive analysis of the worst-case performance of $\texttt{DTrack}-RR$, and derive the parameter value of $\alpha$ for which the best competitive ratio is obtained.

**Claim 3.1.** *Let $t$ be a time and $s$ a server. Let $\tau$ be the time of the latest transition before $t+1$.* $\texttt{DTrack}-RR$ *maintains that* $\texttt{Def}[s] = \texttt{def}(s_c, s, [\tau, t+1))$.

*Proof.* Immediate from the code (Lines 14–15 stands for the initialization upon assignment, and Lines 18–24 stand for the maintenance between assignments). □

**Lemma 3.1.** *Let $t$ be a time and $s$ a server. Let $\tau$ be the time of the latest transition before $t+1$. Then,* $\texttt{def}(\sigma(\tau), s, [\tau, t+1)) \leq \alpha C$.

*Proof.* By induction on $t$. For $t = 0$, the claim holds because the server with the minimal hold cost is selected (Line 3). For $t > 0$, if there is no transition at $t$, then the invariant is maintained by the algorithm's code (Line 7). Assume that a transition occurs at time $t$, i.e., $\tau = t$. By the induction hypothesis, $\texttt{def}(\sigma(\tau'), s, [\tau', t)) \leq \alpha C$, where $\tau'$ is the previous transition time. However, since a transition happened at $t$, then for some $s$, $\texttt{def}(\sigma(\tau'), s, [\tau', t+1)) > \alpha C$. Hence, there exists some server $s$ such that $\texttt{hold}(s, t) < \texttt{hold}(\sigma(\tau'), t)$, that is, $\texttt{hold}(\sigma(\tau'), t)$ is not the minimal hold cost at time $t$. Therefore, some identity $s \neq \sigma(\tau')$ can be found such that $\texttt{def}(s, s', [t, t+1)) \leq \alpha C$, for all $s'$ (Line 28) – e.g., the server with the minimal hold cost at $t$ satisfies this requirement. □

**Corollary 3.3.** *If* $\texttt{nextchoice}()$ *is invoked at time $t$, it returns an identifier that is different from $\sigma(t-1)$.*

We term an interval $[\tau, \tau')$ between two consecutive transitions of algorithm ALG or between ALG's last transition and the end of the run as ALG-*round*. Where ALG is clear from the context, we simply say round. It is convenient to describe the assignment choices made by $\texttt{DTrack}-RR$

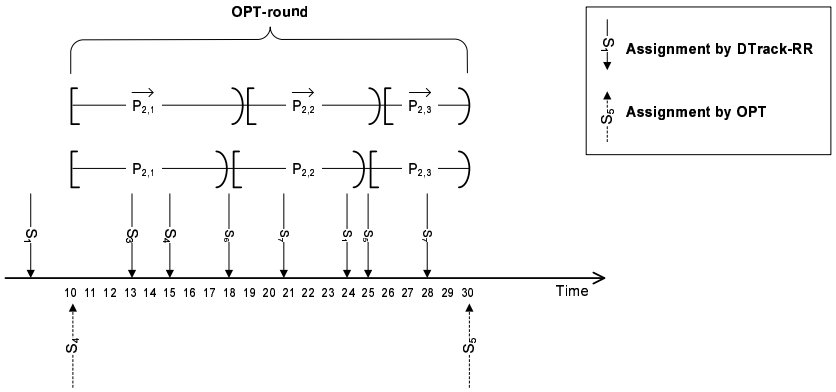



Figure 3.9: **Definition of phases for** DTrack$-RR$**.**

with time as a movement in a circular server identifier space, with a clockwise direction from $s$ to $(s+1)$ mod $k$. We say that $\sigma$ *overtakes* $s$ at time $t$ if $s$ is encountered while moving clockwise from $\sigma(t-1)$ to $\sigma(t)$, and $s \neq \sigma(t)$. In other words, either $\sigma(t-1)$ is $s$, or $s$ is skipped at $t$.

We now consider a DTrack$-RR$-round and an ALG-round of an arbitrary algorithm ALG. We analyze the competitive ratio of DTrack$-RR$ for different values of $\alpha$ by comparing the cost it incurs with the cost incurred by ALG during a single ALG-round $[\tau_i, \tau_{i+1})$ and then generalizing for the whole run. We denote ALG's schedule by $\sigma'$, and ALG's assignment during this ALG-round by $s'$ (if ALG is OPT, the notations are $\sigma^*$ and $s^*$, respectively).

We define two partitions of the interval $[\tau_i, \tau_{i+1})$ into sub-intervals. The first one partitions the interval to *phases* $\{\mathcal{P}_{i,j} = [t_{i,j}, t_{i,j+1})\}$, defined as follows. The first phase starts at $\tau_i$. A phase completes at the earlier between the time when $\sigma$ overtakes $s'$ and $\tau_{i+1}$. The second partition is to *shifted phases* $\{\overrightarrow{\mathcal{P}_{i,j}}\}$, defined as follows. The first shifted phase starts at $\tau_i$. A shifted phase completes at the earlier between one slot after the completion of the corresponding phase and $\tau_{i+1}$.

Figure 3.9 depicts the above definitions for an OPT-round $[10, 30)$, in which $s^* = s_4$. The first phase ends at time 18 when the algorithm chooses $s_6$ and overtakes $s_4$, which was its previous assignment. The second phase ends at time 25 when the algorithm chooses $s_5$ and overtakes $s_4$ for the second time, without choosing $s_4$ in this phase.

**Lemma 3.2.** *Consider an* ALG*-round* $[\tau_i, \tau_{i+1})$ *with* $p$ *phases produced by* DTrack$-RR$*. Then,*

$$\texttt{cost}(\sigma, [\tau_i, \tau_{i+1})) \leq \texttt{hold}(\sigma', [\tau_i, \tau_{i+1})) + pC(k + (k-1)\alpha)$$

*Proof :* Consider a DTrack$-RR$-round $[t, t') \subseteq \mathcal{P}_{i,j}$, and denote $s = \sigma(t)$.

If $s = s'$, then $\texttt{hold}(\sigma', [t, t')) = \texttt{hold}(\sigma, [t, t'))$. Otherwise, by the definition of def, $\texttt{hold}(\sigma, [t, t')) - \texttt{hold}(\sigma', [t, t')) \leq \texttt{def}(s, s', [t, t'))$. By Lemma 3.1, $\texttt{def}(s, s', [t, t')) \leq \alpha C$. Therefore, $\texttt{hold}(\sigma, [t, t')) - \texttt{hold}(\sigma', [t, t')) \leq \alpha C$. There are at most $k-1$ rounds during $\mathcal{P}_{i,j}$ in which the assignment is different from $s'$, and hence,

$$\texttt{hold}(\sigma, \mathcal{P}_{i,j}) - \texttt{hold}(\sigma', \mathcal{P}_{i,j}) \leq (k-1)\alpha C.$$

DTrack$-RR$ performs at most $k$ transitions during $\mathcal{P}_{i,j}$, paying at most $kC$ for setup. Therefore,

$$\texttt{cost}(\sigma, \mathcal{P}_{i,j}) \leq \texttt{hold}(\sigma', \mathcal{P}_{i,j}) + (k-1)\alpha C + kC.$$

$\{\mathcal{P}_{i,j}\}$ is a partition of $[\tau_i, \tau_{i+1})$, and hence,

$$\texttt{cost}(\sigma, [\tau_i, \tau_{i+1})) = \sum_{j=1}^{p} \texttt{cost}(\sigma, \mathcal{P}_{i,j}) \leq$$

$$\sum_{j=1}^{p} \texttt{hold}(\sigma', \mathcal{P}_{i,j}) + pC(k + (k-1)\alpha) = \texttt{hold}(\sigma', [\tau_i, \tau_{i+1})) + pC(k + (k-1)\alpha). \square$$

**Lemma 3.3.** *Consider an* ALG*-round* $[\tau_i, \tau_{i+1})$ *with* $p$ *phases produced by* DTrack$-RR$*, such that either* $\sigma(\tau_i - 1) \neq \sigma'(\tau_i)$*, or* $\sigma(\tau_i) \neq \sigma'(\tau_i)$*. Then,* $\texttt{hold}(\sigma', [\tau_i, \tau_{i+1})) \geq (p-1)\alpha C$*.*

*Proof :* If $p = 1$, the claim trivially holds because the hold costs are non-negative.

Otherwise, consider a phase $\mathcal{P}_{i,j}$ such that $j < p$. This phase ends at $t_{i,j+1}$ that is strictly smaller than $\tau_{i+1}$. We first prove a claim that $\texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > \alpha C$. Consider DTrack$-RR$'s assignment $s$ during the last DTrack$-RR$-round $[t, t_{i,j+1})$ in $\mathcal{P}_{i,j}$, that is, $s = \sigma(t)$, and $\sigma$ overtakes $s'$ at time $t_{i,j+1}$. By definition, $\overrightarrow{\mathcal{P}_{i,j}}$ ends at time $t_{i,j+1} + 1$. Consider two possible cases:

1. If $s \neq s'$, then the algorithm considers picking $s'$ upon the transition from $s$ at $t_{i,j+1}$, and does not select it because there exists a server $\tilde{s}$ such that $\texttt{hold}(s', t_{i,j+1}) - \texttt{hold}(\tilde{s}, t_{i,j+1}) > \alpha C$, and hence, $\texttt{hold}(s', t_{i,j+1}) > \alpha C$. By definition of a shifted phase, $[t_{i,j+1}, t_{i,j+1} + 1) \subseteq \overrightarrow{\mathcal{P}_{i,j}}$. It follows that $\texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > \alpha C$, and the claim holds.

2. Otherwise, $s = s'$. Since the algorithm transitions from $s'$ at time $t_{i,j+1}$, there exists $\tilde{s}$ such that $\texttt{def}(s', \tilde{s}, [t, t_{i,j+1}+1)) > \alpha C$, that is, $\texttt{hold}(\sigma', [t, t_{i,j+1}+1)) > \alpha C$. Assume that $\mathcal{P}_{i,j}$ is the first phase in $[\tau_i, \tau_{i+1})$. Since either $\sigma(\tau_i - 1) \neq \sigma'(\tau_i)$, or $\sigma(\tau_i) \neq \sigma'(\tau_i)$, DTrack$-RR$'s assignment to $s'$ did not happen before $\tau_i$, i.e., $t \geq \tau_i$. Hence, $[t, t_{i,j+1} + 1) \subseteq \overrightarrow{\mathcal{P}_{i,j}}$, by definition of a shifted phase. Otherwise, consider the preceding phase $\mathcal{P}_{i,j-1}$. By definition, $\sigma$ overtakes $s'$ at time $t_{i,j}$. In particular, $\sigma(t_{i,j}) \neq s'$. Since at least one time slot is spent at every assignment, $\sigma$ transitions to $s'$ at time $t_{i,j} < t < t_{i,j+1}$, that is, $[t, t_{i,j+1} + 1) \subseteq \overrightarrow{\mathcal{P}_{i,j}}$. It follows that $\texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > \alpha C$, and the claim holds.

It follows that $\texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > \alpha C$. $\{\overrightarrow{\mathcal{P}_{i,j}}\}$ is a partition of $[\tau_i, \tau_{i+1})$, and therefore,

$$\texttt{hold}(\sigma', [\tau_i, \tau_{i+1})) \geq \sum_{j=1}^{p-1} \texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > (p-1)\alpha C. \square$$

**Lemma 3.4.** *Consider an* ALG*-round* $[\tau_i, \tau_{i+1})$*, such that either* $\sigma(\tau_i - 1) \neq \sigma'(\tau_i)$*, or* $\sigma(\tau_i) \neq$

$\sigma'(\tau_i)$. *Then,*

$$\begin{array}{ll} \frac{\texttt{cost}(\sigma,[\tau_i,\tau_{i+1}))}{\texttt{cost}(\sigma',[\tau_i,\tau_{i+1}))} < k(1+\frac{1}{\alpha}) & \alpha \leq 1 \\ \frac{\texttt{cost}(\sigma,[\tau_i,\tau_{i+1}))}{\texttt{cost}(\sigma',[\tau_i,\tau_{i+1}))} < 1+(k-1)\alpha+k & \alpha \geq 1 \end{array}$$

*Proof :* ALG pays the setup cost $C$ for a single transition during $[\tau_i, \tau_{i+1})$ (at $\tau_i$), and therefore,

$$\texttt{cost}(\sigma', [\tau_i, \tau_{i+1})) = C + \texttt{hold}(\sigma', [\tau_i, \tau_{i+1})).$$

Substituting the ratio's numerator from Lemma 3.2, we receive

$$\frac{\texttt{cost}(\sigma, [\tau_i, \tau_{i+1}))}{\texttt{cost}(\sigma', [\tau_i, \tau_{i+1}))} \leq \frac{\texttt{hold}(\sigma', [\tau_i, \tau_{i+1})) + pC((k-1)\alpha+k)}{C + \texttt{hold}(\sigma', [\tau_i, \tau_{i+1}))} < 1 + \frac{pC((k-1)\alpha+k)}{C + \texttt{hold}(\sigma', [\tau_i, \tau_{i+1}))}.$$

Substituting the denominator from Lemma 3.3,

$$\frac{\texttt{cost}(\sigma, [\tau_i, \tau_{i+1}))}{\texttt{cost}(\sigma', [\tau_i, \tau_{i+1}))} < 1 + \frac{pC(k+(k-1)\alpha)}{C+(p-1)\alpha C} = 1 + \frac{p((k-1)\alpha+k)}{1+(p-1)\alpha}.$$

We denote $\varrho(\alpha, p) \triangleq 1 + \frac{p(k+(k-1)\alpha)}{1+(p-1)\alpha}$. In order to compute $p$ that produces the maximum ratio for a given $\alpha$, we derive $\frac{\partial \varrho}{\partial p}$. We get that $\frac{\partial \varrho}{\partial p} = 0$ for $\alpha = 1$, that is, the function is constant when $\alpha = 1$: $\varrho(1, p) = 2k$ for all $p$. The derivative is strictly positive for $\alpha < 1$ and strictly negative for $\alpha > 1$, therefore, the function is monotonically increasing for $\alpha < 1$ and monotonically decreasing for $\alpha > 1$. For $\alpha < 1$,

$$\sup_{1 \leq p < \infty} \varrho(\alpha, p) = \lim_{p \to \infty} \varrho(\alpha, p) = 1 + \frac{(k-1)\alpha + k}{\alpha} = k(1 + \frac{1}{\alpha}),$$

whereas for $\alpha > 1$,

$$\sup_{1 \leq p < \infty} \varrho(\alpha, p) = \varrho(\alpha, 1) = 1 + (k-1)\alpha + k. \square$$

**Theorem 3.2.** *The competitive ratio of* DTrack$-RR$ *is bounded as follows:*

$$\begin{array}{ll} r(\texttt{DTrack}-RR) < k(1+\frac{1}{\alpha}) & \alpha \leq 1 \\ r(\texttt{DTrack}-RR) < 1+(k-1)\alpha+k & \alpha \geq 1 \end{array}$$

*Proof.* We prove the upper bound on DTrack$-RR$'s competitive ratio for every OPT-round, and conclude the same result for the entire run.

Consider the local ratio between the costs incurred by DTrack$-RR$ and OPT during a single OPT-round $[\tau_i, \tau_{i+1})$, that is, $\frac{\texttt{cost}(\sigma,[\tau_i,\tau_{i+1}))}{\texttt{cost}(\sigma^*,[\tau_i,\tau_{i+1}))}$. If either $\sigma(\tau_i - 1) \neq \sigma(\tau_i)$, or $\sigma(\tau_i - 1) \neq \sigma(\tau_i)$, the claim follows immediately from Lemma 3.4. Otherwise, $\sigma(\tau_i - 1) = \sigma(\tau_i) = \sigma^*(\tau_i) = s^*$. If DTrack$-RR$ never transitions during the OPT-round, then

$$\texttt{cost}(\sigma, [\tau_i, \tau_{i+1})) = \texttt{hold}(\sigma, [\tau_i, \tau_{i+1})) = \texttt{hold}(\sigma^*, [\tau_i, \tau_{i+1})) < \texttt{cost}(\sigma^*, [\tau_i, \tau_{i+1})),$$

and the claim trivially holds. Otherwise, let $\tau_i < t < \tau_{i+1}$ be the first time after $\tau_i$ such that

$\sigma(t) \neq s^*$. Consider a schedule $\sigma'$ that is obtained from $\sigma^*$ by shifting the assignment to $s^*$ from $\tau_i$ to $t$ (assume that this schedule is produced by some algorithm ALG). Note that $\texttt{hold}(\sigma^*, [\tau_i, t)) = \texttt{hold}(\sigma, [\tau_i, t)) \geq 0$, and $\texttt{cost}(\sigma, [t, \tau_{i+1})) = \texttt{cost}(\sigma, [\tau_i, \tau_{i+1})) - \texttt{hold}(\sigma, [\tau_i, t)) \geq \texttt{cost}(\sigma^*, [\tau_i, \tau_{i+1})) - \texttt{hold}(\sigma, [\tau_i, t)) = \texttt{cost}(\sigma', [t, \tau_{i+1})) \geq 0$. By applying a well-known inequality $\frac{a+x}{b+x} \leq \frac{a}{b}$ for $0 \leq b \leq a$ and $x \geq 0$ to the sought ratio, we get:

$$\frac{\texttt{cost}(\sigma, [\tau_i, \tau_{i+1}))}{\texttt{cost}(\sigma^*, [\tau_i, \tau_{i+1}))} = \frac{\texttt{hold}(\sigma, [\tau_i, t)) + \texttt{cost}(\sigma, [t, \tau_{i+1}))}{\texttt{hold}(\sigma, [\tau_i, t)) + \texttt{cost}(\sigma', [t, \tau_{i+1}))} \leq \frac{\texttt{cost}(\sigma, [t, \tau_{i+1}))}{\texttt{cost}(\sigma', [t, \tau_{i+1}))}.$$

. Since $s^* = \sigma(t-1) = \sigma'(t) \neq \sigma(t)$, the bound from Lemma 3.4 is applicable to the ALG-round $[t, \tau_{i+1})$, and the claim follows. □

### 3.7.2 A Competitive Analysis of CTrack-RR

**Theorem 3.3.** *If* $\texttt{hold}(s, t) \leq aC$ *for all* $s$ *and* $t$*, then* $r(\texttt{CTrack}{-}RR) < (2+a)k$ *for* $\alpha = 1$.

*Proof.* Consider an OPT-round $[\tau_i, \tau_{i+1})$ with $p$ phases produced by $\texttt{CTrack}{-}RR$ as defined in Section 3.7.1, in which $s^*$ is OPT's choice.

Consider a $\texttt{CTrack}{-}RR$ round $[t, t')$ in which server $s$ is $\texttt{CTrack}{-}RR$'s choice. If $t < t' - 1$, then

$$\texttt{hold}(\sigma, [t, t')) = \texttt{hold}(\sigma, [t, t'-1)) + \texttt{hold}(s, t'-1) \leq \texttt{hold}(\sigma, [t, t'-1)) + aC.$$

$\texttt{hold}(\sigma, [t, t'-1)) \leq \alpha C$ since no transition happened at $t'-1$, and hence, $\texttt{hold}(\sigma, [t, t')) \leq (\alpha + a)C$. If $t = t' - 1$, the same result holds trivially. There are $p$ phases in $[\tau_i, \tau_{i+1})$ and at most $k$ rounds in each phase. Summarizing over all $\texttt{CTrack}{-}RR$'s rounds, we get

$$\texttt{cost}(\sigma, [\tau_i, \tau_{i+1})) \leq pkC + \texttt{hold}(\sigma, [\tau_i, \tau_{i+1})) \leq pk(\alpha + a)C + pkC = pk(\alpha + a + 1)C.$$

Consider the last $\texttt{CTrack}{-}RR$ round $[t, t')$ in phase $\mathcal{P}_{i,j}$ such that $j < p$. By definition, $s^*$ is the algorithm's choice in this round. A transition happens, therefore, $\texttt{hold}(\sigma, [t, t')) > \alpha C$. Hence, $\texttt{hold}(\sigma^*, [t, t')) > \alpha C$. Summarizing over all phases in $[\tau_i, \tau_{i+1})$, we get

$$\texttt{cost}(\sigma^*, [\tau_i, \tau_{i+1})) = C + \texttt{hold}(\sigma^*, [\tau_i, \tau_{i+1})) > (1 + (p-1)\alpha)C.$$

Hence,

$$\frac{\texttt{cost}(\sigma, [\tau_i, \tau_{i+1}))}{\texttt{cost}(\sigma^*, [\tau_i, \tau_{i+1}))} < k\frac{p(\alpha + a + 1)}{1 + (p-1)\alpha}.$$

For $\alpha = 1$, this ratio is smaller than $(2+a)k$ for all $p$. Since this upper bound limits the algorithm's competitive ratio for every OPT-round, we conclude the same result for the entire run. □

### 3.7.3 A Competitive Analysis of DTrack-B

In this section, we prove the upper bound on the competitive ratio of $\texttt{DTrack}-B$ for arbitrary $\beta$ values. The following lemma is an adaptation of Lemma 3.3 for $\texttt{DTrack}-B$.

**Lemma 3.5.** *Consider an* ALG*-round* $[\tau_i, \tau_{i+1})$ *with* $p$ *phases produced by* $\texttt{DTrack}-B$*, such that either* $\sigma(\tau_i - 1) \neq \sigma'(\tau_i)$*, or* $\sigma(\tau_i) \neq \sigma'(\tau_i)$*. Then,*

$$\texttt{hold}(\sigma', [\tau_i, \tau_{i+1})) \geq (p-1)C \min(\alpha, \alpha - \beta).$$

*Proof.* Like in Lemma 3.3, we consider a phase $\mathcal{P}_{i,j}$ such that $j < p$, which ends at $t_{i,j+1}$. We first prove a claim that $\texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > \max(\alpha, \alpha - \beta)C$. Consider $\texttt{DTrack}-B$'s assignment $s$ during the last $\texttt{DTrack}-B$-round $[t, t_{i,j+1})$ in $\mathcal{P}_{i,j}$, that is, $s = \sigma(t)$, and $\sigma$ overtakes $s'$ at time $t_{i,j+1}$. Consider the case when $s \neq s'$. This happens for one of two reasons:

1. There exists a server $\tilde{s}$ such that $\texttt{hold}(s', t_{i,j+1}) - \texttt{hold}(\tilde{s}, t_{i,j+1}) > \alpha C$, and therefore, $\texttt{hold}(s', t_{i,j+1}) > \alpha C$. $[t_{i,j+1}, t_{i,j+1} + 1) \subseteq \overrightarrow{\mathcal{P}_{i,j}}$, hence, $\texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > \alpha C$, and the claim follows.

2. $\texttt{def}(s, s', [t, t_{i,j+1} + 1)) \leq \beta C$. There exists a server $\tilde{s}$ that triggered the transition, and therefore, $\texttt{def}(s, \tilde{s}, [t, t_{i,j+1} + 1)) > \alpha C$. Hence, $\texttt{def}(s', \tilde{s}, [t, t_{i,j+1} + 1)) > (\alpha - \beta)C$, that is, $\texttt{hold}(\sigma', \overrightarrow{\mathcal{P}_{i,j}}) > (\alpha - \beta)C$, and the claim follows.

The rest of the proof is identical to that of Lemma 3.3. □

**Theorem 3.4.** *The competitive ratio of* $\texttt{DTrack}-B$ *is bounded as follows:*

$$\begin{array}{lll} r(\texttt{DTrack}) < 1 + (k-1)\alpha + k & \alpha \geq 1 \text{ and } \beta \leq \alpha - 1 & (1) \\ r(\texttt{DTrack}) < k(1 + \frac{1}{\alpha}) & \alpha \leq 1 \text{ and } \beta \leq 0 & (2) \\ r(\texttt{DTrack}) < 1 + \frac{(k-1)\alpha + k}{\alpha - \beta} & \max(0, \alpha - 1) \leq \beta \leq \alpha & (3) \end{array}$$

*Proof :* Consider the local ratio between the costs incurred by $\texttt{DTrack}-RR$ and OPT during a single OPT-round $[\tau_i, \tau_{i+1})$. Similarly to the proof of Theorem 3.2, we derive

$$\frac{\texttt{cost}(\sigma, [\tau_i, \tau_{i+1}))}{\texttt{cost}(\sigma^*, [\tau_i, \tau_{i+1}))} < 1 + \frac{p((k-1)\alpha + k)}{1 + (p-1)\min(\alpha, \alpha - \beta)}.$$

We denote $\varrho(\alpha, \beta, p) \triangleq 1 + \frac{p((k-1)\alpha+k)}{1+(p-1)\min(\alpha,\alpha-\beta)}$. If $\min(\alpha, \alpha - \beta) \geq 1$ (i.e., $\alpha \geq 1$ and $\alpha - \beta \geq 1$), then the derivative $\frac{\partial \varrho}{\partial p}$ is non-negative, and hence,

$$\sup_{1 \leq p < \infty} \varrho(\alpha, \beta, p) = \varrho(\alpha, \beta, 1) = 1 + (k-1)\alpha + k \quad \text{if } \alpha \geq 1 \text{ and } \beta \leq \alpha - 1. \tag{3.1}$$

If $\min(\alpha, \alpha - \beta) \leq 1$, then $\frac{\partial \varrho}{\partial p}$ is non-positive, and hence,

$$\sup_{1 \leq p < \infty} \varrho(\alpha, \beta, p) = \lim_{p \to \infty} \varrho(\alpha, \beta, p) = 1 + \frac{(k-1)\alpha + k}{\min(\alpha, \alpha - \beta)}.$$

Consider the case when $\min(\alpha, \alpha-\beta) = \alpha$, i.e., $\beta \leq 0$. Combining this with $\alpha \leq 1$, we get:

$$\sup_{1\leq p<\infty} \varrho(\alpha,\beta,p) = 1 + \frac{(k-1)\alpha + k}{\alpha} = k(1+\frac{1}{\alpha}) \quad \text{if } \alpha \leq 1 \text{ and } \beta \leq 0. \tag{3.2}$$

Consider the case when $\min(\alpha, \alpha-\beta) = \alpha-\beta$, i.e., $\beta \geq 0$. Combining this with $\alpha - \beta \leq 1$ and $\beta \leq \alpha$ (by definition), we get:

$$\sup_{1\leq p<\infty} \varrho(\alpha,\beta,p) = 1 + \frac{(k-1)\alpha + k}{\alpha-\beta} \quad \text{if } \max(0,\alpha-1) \leq \beta \leq \alpha, \tag{3.3}$$

and the claim follows. □

### 3.7.4 Non-Competitiveness of Opportunistic Algorithms

In this section, we show that the opportunistic versions of `DTrack` are not competitive, that is, the worst-case competitive ratio depends on $C$, rather than on the problem size $k$.

**Theorem 3.5.** *The competitive ratio of* `DTrack`$-F$ *and* `DTrack`$-B$ *with* $\alpha = \beta$ *is* $\Omega(C)$.

*Proof.* Assume wlog that $C$ is a positive integer (otherwise, the theorem can be proved for $C' = \lfloor C \rfloor$). Let $\varepsilon$ be a small number s.t. $0 < \varepsilon < \frac{\alpha}{C+2}$. Consider three servers $s_0$, $s_1$ and $s_2$. Let $\texttt{hold}(s_2, t) = \varepsilon$ for all $t$, whereas $\texttt{hold}(s_0, t)$ and $\texttt{hold}(s_1, t)$ are defined as follows for integer values of $0 \leq i < \lceil \frac{C}{2} \rceil$:

$$\texttt{hold}(s_0,t) = \begin{cases} (2i+3)\varepsilon & t = (2i+1)(C+1) \\ \alpha & 2i(C+1) < t < (2i+1)(C+1) \\ 0 & \text{otherwise} \end{cases}$$

and

$$\texttt{hold}(s_1,t) = \begin{cases} (2i+2)\varepsilon & t = 2i(C+1) \\ \alpha & (2i+1)(C+1) < t < (2i+2)(C+1) \\ 0 & \text{otherwise} \end{cases}$$

The hold costs during the interval $[0, 3C+3]$ are depicted in Figure 3.10. Note that for $0 \leq i < \lceil \frac{C}{2} \rceil$, it holds that $(2i+3)\varepsilon \leq (C+2)\varepsilon < \alpha$. Therefore, $\texttt{hold}(s_0, t) \leq \alpha$, and $\texttt{hold}(s_1, t) \leq \alpha$ for all $t$ during this interval.

**Lemma 3.6.** *Both* `DTrack`$-F$ *and* `DTrack`$-B$ *assign* $s_0$ *at times* $t = 2i(C+1)$, *and* $s_1$ *at times* $t = (2i+1)(C+1)$, *for* $0 \leq i < \lceil \frac{C}{2} \rceil$.

*Proof.* By induction on $i$. At time $t = 0$, both algorithms choose $s_0$ because it offers the minimal hold cost. The induction step considers two cases:

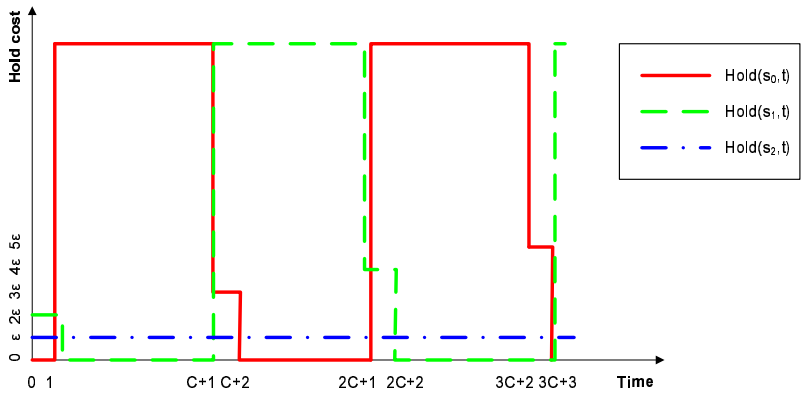



Figure 3.10: **An example of hold costs for which** DTrack$-F$ **and** DTrack$-B$ **with** $\alpha = \beta$ **are** $\Omega(C)$**-competitive.**

1. $t = (2i+1)(C+1)$. Both algorithms transitioned to $s_0$ at $2i(C+1)$ by induction hypothesis. We compute $\texttt{def}(s_0, s_1, [2i(C+1), t))$ and $\texttt{def}(s_0, s_2, [2i(C+1), t))$.

$$\texttt{def}(s_0, s_1, [2i(C+1), (2i+1)(C+1)+1)) = \\ -(2i+2)\varepsilon + \alpha C + ((2i+1)+2)\varepsilon = \alpha C + \varepsilon > \alpha C,$$

   whereas

$$\texttt{def}(s_0, s_2, [2i(C+1), (2i+1)(C+1)+1)) = \\ -\varepsilon + C(\alpha - \varepsilon) + ((2i+1)+2-1)\varepsilon < \alpha C - \varepsilon(C - (2i+1)) \leq \alpha C.$$

   Note that both $\texttt{def}(s_0, s_1, [2i(C+1), t')$ and $\texttt{def}(s_0, s_2, [2i(C+1), t')$ are strictly smaller than $\alpha C$ for $t' < t$. Therefore, the transition happens at $(2i+1)(C+1)$ for the first time since $2i(C+1)$.

2. $t = (2i+2)(C+1)$. This case is proved analogously to the previous one.

Upon every transition, DTrack$-F$ selects the server with the zero hold cost, i.e., $s_0$ at times $2i(C+1)$, and $s_1$ at times $t = (2i+1)(C+1)$. DTrack$-B$ selects the server that achieves the largest deficit, i.e., it makes the same choice. □

Consider a run of DTrack$-F$ and DTrack$-B$ during the interval $[0, C^2-1)$. Both algorithms behave identically. They transition $C$ times during this interval (at $t = i(C+1)$, for $0 \leq i \leq C-1$). Hence, the total setup cost is $C^2$. The total hold cost exceeds $\alpha C^2$ since a hold cost of above $\alpha C$ is incurred between every two transitions. Hence, the total cost during the interval exceeds $(\alpha+1)C^2$. In the same setting, OPT selects $s_2$ at $t = 0$ and never changes its assignment, thus paying a total setup cost of $C$ and a total hold cost of $\varepsilon(C^2-1) < \alpha C$. Hence, the competitive ratio of both online algorithms is $\Omega(C)$. □

# Chapter 4

# The Load-Distance Balancing Problem

The increasing demand for real-time access to networked services is driving service providers to deploy multiple geographically dispersed service points, or servers. This trend can be observed in various systems, such as content delivery networks (CDNs) [63] and massively multiplayer online gaming (MMOG) grids [42]. Another example can be found in wireless mesh networks (WMNs) [16]. A WMN is a large collection of wireless routers, jointly providing Internet access in residential areas with limited wireline infrastructure via a handful of wired gateways. WMNs are envisaged to provide citywide "last-mile" access for numerous mobile devices running media-rich applications with stringent quality of service (QoS) requirements, e.g., VoIP, VoD, and online gaming. Gateway functionality is anticipated to expand, and to deploy application server logic [16].

Employing distributed servers instead of centralized server farms enables location-dependent QoS optimizations, which enhance the users' soft real-time experience. Service responsiveness is one of the most important QoS parameters. For example, in the first-person shooter (FPS) online game [42], the system must provide an end-to-end delay guarantee of below 100ms. In VoIP, the typical one-way delay required to sustain a normal conversation quality is below 120ms [58]. Such guarantees are nontrivial to implement in mesh networks, due to multiple hops and a limited number of gateways.

Deploying multiple servers gives rise to the problem of *service assignment*, namely associating each user session with a server or gateway. For example, each CDN user gets its content from some proxy server, a player in a MMOG is connected to one game server, and the Internet traffic of a WMN user is typically routed via a single gateway [16]. In this context, we identify the need to model the service delay of a session as a sum of a *network delay*, incurred by the network connecting the user to its server, and a *congestion delay*, caused by queueing and processing at the assigned server. Due to the twofold nature of the overall delay, simple heuristics that either greedily map every session to the closest server, or spread the load evenly regardless of geography do not work well in many cases. In this work, we present a novel approach to service assignment, which is based on both metrics. We call the new problem, which seeks to minimize the service delay among all users, *load-distance balancing*, or `LDB`.

In this chapter, we address the `LDB` problem in a centralized setting. The problem, which is precisely defined in Section 5.2, seeks to minimize the service delay among all users, and has two flavors: (1) *maximum* delay minimization and (2) *average* delay minimization. We demonstrate

that the min-max LDB problem is both NP-hard and non-approximable within the factor of two for general distance and load functions (Section 4.3.1), and present the best possible 2-approximation algorithm (Section 4.3.2). For a special case when the users and the servers are located on a line segment, and the network delays are Euclidean distances, we demonstrate a polynomial dynamic-programming algorithm for this problem (Section 4.3.3). Following this, we present a polynomial algorithm for the min-average LDB, which applies for convex load functions, and finally, a dynamic-programming solution for the linear setting which has an improved time complexity (Section 4.4.2).

## 4.1 Related Work

Load-distance balancing is an extension of the load balancing problem, which has been comprehensively addressed in the context of tightly coupled systems like multiprocessors, compute clusters etc. (e.g., [25]). However, in large-scale networks, simple load balancing is insufficient because servers are not co-located. While some prior work [42, 63] indicated the importance of considering both distance and load in wide-area settings, we are not aware of any study that provides a cost model that combines these two metrics and can be analyzed.

While a centralized approach is feasible for small-scale environments, it cannot scale to large networks, e.g., city-wide WMNs. We therefore seek for *local distributed* solutions, which spread as few information as possible for achieving the *desired* approximation of the optimal solution. Chapter 5 explores this approach for the min-max LDB problem, and demonstrates how centralized optimization algorithms can serve as local building blocks for scalable distributed solutions.

## 4.2 Problem Definition

Consider a set of $k$ servers $S$ and a set of $n$ user sessions $U$. The *network delay* function, $D : (U \times S) \rightarrow \mathbb{R}^+$, captures the network distance between a user and a server. The users and the servers do not necessarily reside in a metric space (i.e., $D$ is not necessarily subject to the triangle inequality).

Consider an assignment $\lambda : U \rightarrow S$ that maps every user to a single server. Each server $s$ has a monotonic non-decreasing *congestion delay* function, $\delta_s : \mathbb{N} \rightarrow \mathbb{R}^+$, reflecting the delay it incurs to every assigned session. For simplicity, all users incur the same load. Different servers can have different congestion delay functions. The service delay $\Delta(u, \lambda)$ of session $u$ in assignment $\lambda$ is the sum of the two delays:

$$\Delta(u, \lambda) \triangleq D(u, \lambda(u)) + \delta_{\lambda(u)}(|\{v : \lambda(v) = \lambda(u)\}|).$$

Note that our model does not include congestion within the network. Typically, application-induced congestion bottlenecks tend to occur at the servers or the last-hop network links, which can be also attributed to their adjacent servers. For example, in a CDN [63], the assignment of users to content servers has a more significant impact on the load on these servers and their access links than on the congestion within the public Internet. In WMNs, the effect of load on wireless

links is reduced by flow aggregation [58], which is applied for increasing the wireless capacity attainable for real-time traffic. The last-hop infrastructure, i.e., the gateways' wireless and wired links, is mostly affected by network congestion [16].

The min-max LDB problem is defined as follows. The cost of an assignment $\lambda$ is the *maximum* delay it incurs on a user:

$$\Delta^M(\lambda(U)) \triangleq \max_{u \in U} \Delta(u, \lambda).$$

The optimization goal is to find an assignment $\lambda^*$ such that $\Delta^M(\lambda^*(U))$ is minimized.

The min-average LDB problem is defined as follows. The cost of an assignment $\lambda$ is the *total* delay incurred by it:

$$\Delta^T(\lambda(U)) \triangleq \sum_{u \in U} \Delta(u, \lambda).$$

In this context, the optimization goal goal is to find an assignment $\lambda^*$ such that $\Delta^T(\lambda^*(U))$ is minimized.

## 4.3 Min-Max Load-Distance Balancing

This chapter studies the min-max LDB problem. We first analyze its computational complexity (Section 4.3.1), and present the best possible approximation algorithm for general cost functions (Section 4.3.2). Following this, we present an efficient polynomial algorithm for a special case in which network distances are captured by a linear Euclidean metric (Section 4.3.3).

### 4.3.1 Computational Hardness

We first prove that the min-max LDB optimization problem is NP-hard. This result stems from the hardness of the decision variation of LDB, denoted LDB−D. In this context, the problem is to decide whether delay $\Delta^*$ is feasible, i.e., whether there exists an assignment $\lambda$ such that $\Delta^M(\lambda(U)) \leq \Delta^*$.

In what follows, we prove the show a reduction from the *exact set cover* (XSC) problem [6]. An instance of XSC is a collection $S$ of subsets over a finite set $U$. The solution is a set cover for $U$, i.e., a subset $S' \subseteq S$ such that every element in $U$ belongs to at least one member of $S'$. The decision problem is whether there is a cover such that each element belongs to precisely one set in the cover. XSC is NP-hard even if all subsets in $S$ have the same size.

**Theorem 4.1.** *The* LDB−D *problem is NP-hard.*

*Proof.* Consider an instance of XSC in which $|U| = n$, $|S| = k$, and each set contains exactly $m$ elements, such that $\frac{n}{m} < k$. The problem is therefore whether there is a cover containing $\frac{n}{m}$ sets.

The transformation of this instance to an instance of LDB−D is as follows. In addition to the elements in $U$, we define a set $U'$ of $M(k - \frac{n}{m})$ dummy elements, where $M > m$. We construct a bipartite graph (Figure 4.1), in which the one side contains the elements in $U \bigcup U'$ (the users), and the other side contains the sets in $S$ (the servers). The dummy users are at distance $d_1$ from each server. The real users are at distance $d_2 > d_1$ from each server that covers them, and at distance $\infty$

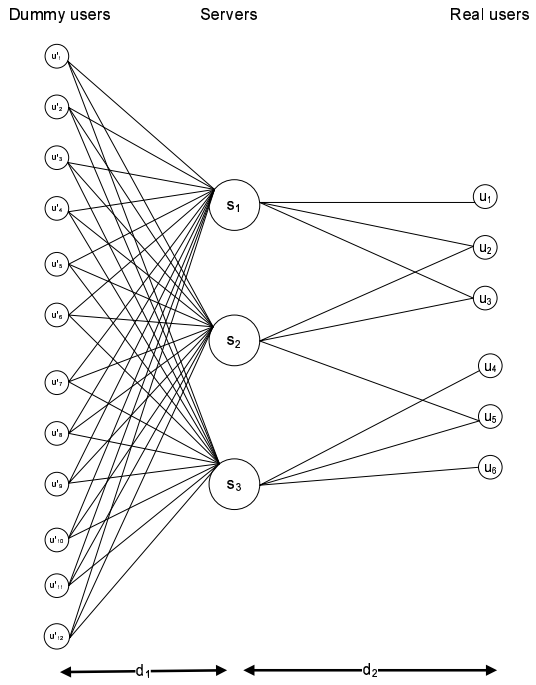



Figure 4.1: **Reduction from exact set cover to LDB-D.**

from all the other servers. The capacity of each server for distance $d_1$ is $M$, and for distance $d_2$ is $k$, i.e., $\delta_s^{-1}(\Delta^* - d_1) = M$, and $\delta_s^{-1}(\Delta^* - d_2) = m$.

A feasible XSC solution induces a feasible LDB assignment: each real user is assigned to a server representing the unique set that covers the corresponding element, and the dummy users are evenly spread among the remaining $k - \frac{n}{m}$ servers. We argue that no other feasible assignment exists. Consider a server utilized by a feasible $\lambda$. It can accommodate either $M$ dummy users, or any combination of $0 < m' \leq m$ original users and $m - m'$ dummy users (any other assignment incurs a delay above $\Delta^*$ to some user). Assume that both real and dummy users are assigned to at least one server. Then, the total number of servers that have real users assigned to them is $k' > \frac{n}{m}$. All these servers have capacity $m$, and hence, they serve at most $mk' - n$ dummy users. The remaining servers can host $M(k - k')$ dummy users. Hence, the total number of assigned dummy users is bounded by $M(k-k') + mk' - n = M(k - \frac{n}{m}) - M(k' - \frac{n}{m}) + m(k' - \frac{n}{m}) < M(k - \frac{n}{m})$, that is, the assignment is not feasible. Hence, exactly $\frac{n}{m}$ servers must be allocated to real users, thus solving the XSC instance. □

A slight modification of the above proof demonstrates that even a $2 - \varepsilon$ approximation of LDB is NP-hard, for an arbitrarily small $\varepsilon$. In this context, the $c$-approximate LDB−D is to decide, given the delay $\Delta^*$, whether there exists an assignment $\lambda$ such that $\Delta^M(\lambda(U)) \leq c\Delta^*$, for some $c > 1$.

**Theorem 4.2.** *The $(2 - \varepsilon)$-approximate* LDB−D *is NP-hard, for all $\varepsilon > 0$.*

*Proof.* We construct a bipartite user-server graph, identically to the proof of Theorem 4.1. The

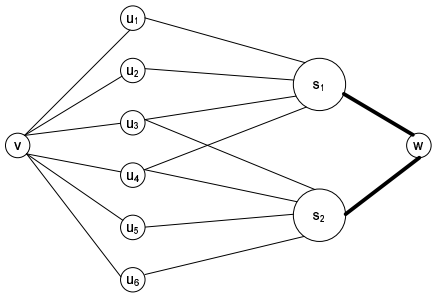



Figure 4.2: **The bipartite graph for a single phase of** `BFlow`**.**

congestion delay function is uniform among all servers:

$$\delta(i) = \begin{cases} \Delta^* - d_2 & \text{if } \ i \leq m \\ \Delta^* - d_1 & \text{if } \ m \leq i \leq M \\ \infty & \text{otherwise} \end{cases}$$

It is easy to verify that a feasible XSC solution induces a feasible LDB assignment.

Suppose we let $d_1 = \varepsilon$ and $d_2 = \Delta^* - \varepsilon$. If an element is not a member of a set, the distance to that server is infinite. If there is no solution for exact cover, i.e., any collection of $\frac{n}{m}$ sets leaves some element uncovered, the corresponding real user will have to be assigned to a server that is also hosting $M-1$ dummy users. The delay experienced by this user is thus $d_2+(\Delta^*-d_1) = 2(\Delta^*-\varepsilon)$. Therefore, the XSC problem reduces to a $(2-\varepsilon)$-approximate LDB−D. □

Note, however, that the distance function considered in the proof does not satisfy the triangle inequality. Indeed, since each dummy user is connected to all servers, the distance between any pair of servers does not exceed $2d_1$. Hence, it is impossible for the distance between any real user and some server to exceed $2d_1 + d_2$, and in particular, it cannot be infinite - a contradiction.

**Claim 4.1.** *There exists a distance function subject to the triangle inequality, for which a* $\frac{5}{3}$*-approximate* LDB−D *is NP-hard.*

*Proof.* We consider the same graph as the proof of Theorem 4.1, and choose $d_1 = \frac{\Delta^*}{3}$ and $d_2 = \Delta^*$. The distance of a real user to a server is either $\Delta^*$ or $2d_1 + d_2 = \frac{5}{3}\Delta^*$. If there is no solution to exact cover, then the best solution can have delay no lower than $\frac{5}{3}\Delta^*$. □

### 4.3.2 BFlow – a 2-Approximation Algorithm

In this section, we present a simple algorithm, called `BFlow`, which computes a 2-approximate solution for min-max LDB. By Theorem 4.3.1, this is also the best possible approximation for general distance functions.

`BFlow` works in phases. In each phase, the algorithm guesses $\Delta^* = \Delta^M(\lambda^*(U))$, and checks the feasibility of a specific assignment, in which neither the network nor the congestion delay exceeds $\Delta^*$, and hence, its cost is bounded by $2\Delta^*$. `BFlow` performs a binary search on the value of $\Delta^*$. A single phase works as follows:

1. Each user $u$ marks all servers $s$ that are at distance $D(u,s) \leq \Delta^*$. These are its feasible servers.

2. Each server $s$ announces how many users it can serve by computing the inverse of $\delta_s(\Delta^*)$.

3. We have a generalized matching problem where an edge means that a server is feasible for the user. The degree of each user in the matching is exactly one, and the degree of server $s$ is at most $\delta_s^{-1}(\Delta^*)$. A feasible solution, if one exists, can be solved via a max-flow min-cut algorithm in a bipartite user-server graph with auxiliary source and sink vertices. Figure 4.2 depicts an example of such a graph.

**Theorem 4.3.** `BFlow` *computes a 2-approximation of an optimal assignment for min-max* `LDB`.

*Proof.* Consider an optimal assignment $\lambda^*$ with cost $\Delta^*$. It holds that $\Delta_1 = \max_u D(u, \lambda^*(u)) \leq \Delta^*$, and $\Delta_2 = \max_s \delta_s(\mathcal{L}(s)) \leq \Delta^*$. A phase of `BFlow` that tests an estimate $\Delta = \max(\Delta_1, \Delta_2)$ is guaranteed to find a feasible solution with cost $\Delta' \leq \Delta_1 + \Delta_2 \leq 2\Delta^*$. □

Since there are at most $kn$ distinct $D$ values, the number of the binary search phases that attributes to covering all of them is logarithmic in $n$. The number of phases that attributes to covering all the possible capacities of server $s$ is $O(\log \delta_s(n))$, which is at linear in $n$ or below for any reasonable $\delta_s$. Hence, `BFlow` is a polynomial algorithm.

### 4.3.3 Optimal Assignment on a Line with Euclidean Distances

In this section, we consider the case when the users and the servers are located on a line segment $[0, L]$, and the network delays are Euclidean distances. We show that min-max `LDB` is polynomially solvable in this model through dynamic programming.

We start with some definitions. For simplicity of presentation, we assume that every user or server $i$ has a distinct location $x_i$. The distance between user $u$ and server $s$ is therefore $D(u,s) = |x_s - x_u|$. Assignment $\lambda$ is called *order-preserving* if for every pair of users $u_1$ and $u_2$ such that $x_{u_1} < x_{u_2}$ it holds that $x_{\lambda(u_1)} < x_{\lambda(u_2)}$. Otherwise, both $\lambda$ and every pair $(u_1, u_2)$ for which this condition does not hold are called *order-violating*.

Every order-preserving assignment partitions the line into a series of non-overlapping segments such that every user within segment $i$ is assigned to server $s_i$. Segment $i$ is located to the left from segment $j$ iff $i < j$. Note that $s_i$ does not necessarily located inside segment $i$.

**Theorem 4.4.** *The min-max* `LDB` *problem on a line has an order-preserving optimal assignment.*

*Proof.* Consider an order-violating assignment $\lambda$. We show how it can be transformed into an order-preserving assignment that incurs smaller or equal cost.

Since $\lambda$ is order-violating, there exists a pair of users $u_1$ and $u_2$ assigned to servers $s_2$ and $s_1$ such that $x_{u_1} < x_{u_2}$ but $x_{s_2} > x_{s_1}$. We transform $\lambda$ to a new assignment $\lambda'$ from by switching the assignments of $u_1$ and $u_2$, i.e., $\lambda'(u_1) = s_1$ and $\lambda'(u_2) = s_2$. Since this switch does not affect the load on $s_1$ and $s_2$, no change is incurred to any user's processing delay. Therefore, only the network delays incurred to $u_1$ and $u_2$ are affected. We therefore need to show that $\lambda'$ does not incur greater maximum network delay values than $\lambda$, that is,

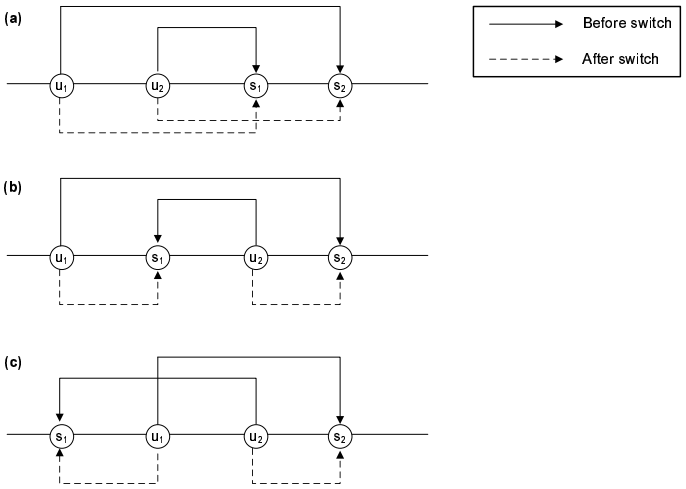



Figure 4.3: **Switching the assignment of an order-violating pair** $(u_1, u_2)$**.**

**Claim 4.2.** *It holds that* $\max(D(u_1, s_1), D(u_2, s_2)) \leq \max(D(u_1, s_2), D(u_2, s_1))$.

*Proof :* Consider the following cases:

1. $x_{u_1} < x_{u_2} < x_{s_1} < x_{s_2}$ (Figure 4.3(a)). Then, $D(u_1, s_1) < D(u_1, s_2)$ and $D(u_2, s_2) < D(u_1, s_2)$, hence, $\max(D(u_1, s_1), D(u_2, s_2)) < \max(D(u_1, s_2), D(u_2, s_1))$.

2. $x_{u_1} < x_{s_1} < x_{u_2} < x_{s_2}$ (Figure 4.3(b)). Then, $D(u_1, s_1) < D(u_1, s_2)$ and $D(u_2, s_2) < D(u_1, s_2)$, hence, $\max(D(u_1, s_1), D(u_2, s_2)) < \max(D(u_1, s_2), D(u_2, s_1))$.

3. $x_{s_1} < x_{u_1} < x_{u_2} < x_{s_2}$ (Figure 4.3(c)). Then, $D(u_1, s_1) < D(u_2, s_1)$ and $D(u_2, s_2) < D(u_1, s_2)$, hence, $\max(D(u_1, s_1), D(u_2, s_2)) < \max(D(u_1, s_2), D(u_2, s_1))$.

4. $x_{s_1} < x_{u_1} < x_{s_2} < x_{u_2}$. Symmetric to case (2).

5. $x_{s_1} < x_{s_2} < x_{u_1} < x_{u_2}$. Symmetric to case (1). □

This way, we switch the assignment of every order-violating pair of users until eventually an order-preserving assignment is achieved.

It follows that every optimal assignment for min-max LDB is either order-preserving, or can be transformed into an order-preserving assignment that incurs an equal service delay. □

We now identify the recursive structure of an optimal assignment $\lambda^*$. Let $\lambda^*_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq k$ be an optimal assignment for users $\{u_i, \ldots, u_n\}$ that employs servers $\{s_j, \ldots, s_k\}$. We can assign $l = 0, \ldots, n - i + 1$ leftmost users to server $s_j$. This assignment defines the maximum delay among the leftmost users. From the optimality of $\lambda^*_{i,j}$, the assignment $\lambda^*_{i+l,j+1}$ of the remaining users to the remaining servers is also an optimal one. Hence,

$$\Delta^M(\lambda^*_{i,j}) = \min_{0 \leq l \leq n-i+1} [\max(\delta_{s_j}(l) + \max_{0 \leq l' < l} |x_{s_j} - x_{u_{i+l'}}|, \Delta^M(\lambda^*_{i+l,j+1}))], \tag{4.1}$$

The boundary conditions are: $\Delta^M(\lambda^* n+1, j) = 0$ (no users), and $\Delta^M(\lambda^* i, k+1) = \infty$ (no servers), for $1 \leq i \leq n$ and $1 \leq j \leq k$. The global optimal assignment cost is $\Delta^M(\lambda^*(U,S)) = \Delta^M(\lambda^*_{1,1})$.

Optimal assignments can be computed through dynamic programming [48], using the above recurrence. An optimal algorithm employs a two-dimensional table $\texttt{Table}[1..n+1, 1..k+1]$, where an entry $\texttt{Table}[i,j]$ holds the value of $\Delta^M(\lambda^*_{i,j})$, and the number of users assigned to $s_j$. Note that

$$\max_{0 \leq l' < l} |x_{s_j} - x_{u_{i+l'}}| = \max(|x_{s_j} - x_{u_i}|, |x_{s_j} - x_{u_{i+l-1}}|),$$

and hence, the computation of a single entry $\texttt{Table}[i,j]$ incurs $O(1)$ operations for each examined entry $\texttt{Table}[i+l, j+1]$. A naive implementation examines $O(n)$ such entries, an therefore, the time complexity of filling the whole table is $O(kn^2)$. This result can be improved by noting that Eq. (4.1) defines a min-max among the value pairs of $f_{i,j}(l) = \delta_{s_j}(l) + \max_{0 \leq l' < l} |x_{s_j} - x_{u_{i+l'}}|$ (a non-decreasing function of $l$) and $g_{i,j}(l) = \Delta^M(\lambda^*_{i+l,j+1})$ (a non-increasing function of $l$). Hence, the min-max is achieved for the value of $l$ for which $f_{i,j}(l) - g_{i,j}(l)$ is closest to zero. It can be efficiently found through binary search, which yields $O(\log n)$ operations for a single table entry, and $O(kn \log n)$ operations altogether.

## 4.4 Min-Average Load-Distance Balancing

We now demonstrate an efficient polynomial-time algorithm for min-average LDB assignment, and an alternative solution for the linear case, which has an improved running time.

### 4.4.1 The Optimal Algorithm

**Assumption:** We assume that for each server $s$, the function $x\delta_s(x)$ is convex (most practical congestion delay functions satisfy this requirement).

The algorithm reduces the assignment problem to minimum-cost matching in a bipartite graph. The left part contains $n$ users, and the right part contains $n$ copies of each server (i.e., $nk$ nodes). The cost of connecting user $u$ to the $i$'th instance of server $s$ is defined as

$$\Delta_i(u,s) = D(u,s) + i\delta_s(i) - (i-1)\delta_s(i-1).$$

Intuitively, these costs are *marginal* costs in the assignment, that is, $\Delta_i(u,s)$ is the cost of connecting user $u$ to server $s$ after $i-1$ other users.

The algorithm computes a minimum-cost matching in the constructed graph (i.e., each user is assigned to exactly one server copy), and turns this matching into a legal assignment by assigning each user to the server it is matched to, regardless of the instance number.

**Theorem 4.5.** *The algorithm computes an optimal assignment for min-average* LDB.

*Proof.* We first claim that if the copy $s_i$ of server $s$ is utilized by the matching, then all the copies $s_j$ for $j \leq i$ are used too. Indeed, suppose by contradiction that user $u$ is matched to some copy $s_i$

($i > 1$), and $s_{i-1}$ is not used. If $u$ is switched from $s_i$ to $s_{i-1}$, the matching cost can be reduced by

$$\Delta_i(u,s) - \Delta_{i-1}(u,s) = i\delta_s(i) + (i-2)\delta_s(i-2) - 2\delta_s(i-1),$$

which is a positive value since $x\delta_s(x)$ is a convex function. Hence, the matching's cost can be improved, in contradiction to optimality.

Consider a matching $\mu$ in the bipartite graph for which the set of used instances of each server is contiguous, and the corresponding assignment $\lambda$ for the original problem. We denote the set of users assigned to some instance of server $s$ by $\mu(s)$, and the user assigned to the $i$'th copy of server $s$ by $\mu_i(s)$. Since the used copy set is contiguous, the sum of individual matching cost of the users in $\mu(s)$ telescopes to

$$\begin{aligned}\textstyle\sum_{i=1}^{|\mu(s)|} \Delta_i(\mu_i(s),s) &= |\mu(s)|\delta_s(|\mu(s)|) + \textstyle\sum_{i=1}^{|\mu(s)|} D(\mu_i(s),s) \\ &= \textstyle\sum_{i=1}^{|\mu(s)|}[D(\mu_i(s),s) + \delta_s(|\mu(s)|)] \\ &= \textstyle\sum_{u:\lambda(u)=s} \Delta(u,\lambda).\end{aligned}$$

Hence, the matching's cost is equal to the cost of an assignment for the original problem. Therefore, since the minimum-cost matching $\mu^*$ has the desired instance continuity property, it produces a minimum-cost assignment $\lambda^*$. □

### 4.4.2 Improving the Running Time on a Line with Euclidean Distances

The fastest known minimum-cost flow algorithm on a graph $G(V,E)$ runs in $O(|E|\log|V|(|E| + |V|\log|V|))$ time [83]. We construct a bipartite graph in which $|V| = O(nk)$ and $|E| = O(kn^2)$, hence the running time is $O(kn^2\log(nk)(kn^2 + nk\log(nk))) = O(k^2n^4\log n)$. In a special case when users and servers are located on a line segment, and network delays are modeled as Euclidean distances, this running time can be significantly improved. Similarly to the min-max `LDB` problem, the min-average `LDB` on a line has an order-preserving optimal assignment. Hence, a polynomial time dynamic programming algorithm similar to the one presented in Section 4.3.3 is applicable in this case. The algorithm's running time is $O(kn^2)$ (in contrast to the min-max `LDB`, it cannot apply the binary search optimization to reduce the number of operations on a single table entry to $\log n$).

# Chapter 5

# Scalable Load-Distance Balancing

Chapter 4 introduced a novel *load-distance balancing* problem, or LDB, which arises in the context of assigning multiple users of delay-sensitive network applications to geographically scattered servers in a way that minimizes the delays incurred to these users. The service-level delay is affected by network distances as well as by server loads. Hence, computing an assignment that minimizes this delay requires to consider both factors together. For example, straightforward approaches like always assigning every user to the closest server or spreading all users evenly across random servers produce unsatisfactory results, since they cannot adapt to varying distributions of location among the users.

Resource management problems in which a naïve local assignment leads to suboptimal results are often solved centrally. For example, Cisco wireless local area network (WLAN) controllers [1] perform global optimization in assigning wireless users to access points (APs), after collecting the signal strength information from all managed APs. While this approach is feasible for medium-size installations like enterprise WLANs, its scalability may be challenged in large networks like an urban WMN. For large-scale network management, a distributed protocol with local communication is required.

We observe, however, that load-distance-balanced assignment cannot always be done in a completely local manner. For example, if some part of the network is heavily congested, then a large number of servers around it must be harnessed to balance the load. In extreme cases, the whole network may need to be involved in order to dissipate the excessive load. A major challenge is therefore to provide an *adaptive* solution that performs communication to a distance proportional to that required for handling the given load in each problem instance. We address this challenge, drawing inspiration from workload-adaptive distributed algorithms [31, 72].

In Section 5.3, we present two distributed algorithms for load-distance balancing, `Tree` and `Ripple`, which adjust their communication requirements to the congestion distribution, and produce constant approximations of the optimal cost. `Tree` and `Ripple` dynamically partition the user and server space into *clusters* whose sizes vary according to the network congestion, and solve the problem in a centralized manner within every such cluster. `Tree` does this by using a fixed hierarchy of clusters, so that whenever a small cluster is over-congested and needs to offload users, this cluster is merged with its sibling in the hierarchy, and the problem is solved in the parent cluster. While `Tree` is simple and guarantees a logarithmic convergence time, it suffers from two

drawbacks. First, it requires maintaining a hierarchy among the servers, which may be difficult in a dynamic network. Second, `Tree` fails to load-balance across the boundaries of the hierarchy. To overcome these shortcomings, we present a second distributed algorithm, `Ripple`, which does not require maintaining a complex infrastructure, and achieves lower costs and better scalability, through a more careful load sharing policy. The absence of a fixed hierarchical structure turns out to be quite subtle, as the unstructured merges introduce race conditions. In Appendices 5.5.1 and 5.5.2 we prove that `Tree` and `Ripple` always converge to solutions that approximate the optimal one within a constant factor. For simplicity, we present both algorithms for a static workload. In Section 5.5.3, we discuss how they can be extended to cope with dynamic workloads.

We note that even as a centralized optimization problem, the min-max variation of `LDB` that seeks minimizing the *maximum* delay is NP-hard, as we showed in Chapter 4. Therefore, `Tree` and `Ripple` employ a centralized polynomial 2-approximation algorithm, `BFlow`, within each cluster. The details of `BFlow` were presented in Section 4.3.2.

Finally, we empirically evaluate our algorithms using a case study in an urban WMN environment (Section 5.4). Our simulation results show that both algorithms achieve significantly better costs than naïve nearest-neighbor and perfect load-balancing heuristics (which are the only previous solutions that we are aware of), while communicating to small distances and converging promptly. The algorithms' metrics (obtained cost, convergence time, and communication distance) are scalable and congestion-sensitive, that is, they depend on the distribution of workload rather than the network size. The simulation results demonstrate a consistent advantage of `Ripple` in the achieved cost, due to its higher adaptiveness to user workload.

## 5.1 Related Work

Load-distance balancing is an extension of the well-studied load balancing problem (e.g., [25]). In contrast with distributed algorithms for traditional load balancing (e.g., [61]), our solutions explicitly use the cost function's distance-sensitive nature to achieve locality.

A number of papers addressed geographic load-balancing in cellular networks and wireless LANs (e.g., [29, 52]), and proposed local solutions that dynamically adjust cell sizes. While the motivation of these works is similar to ours, their model is constrained by the rigid requirement that a user can only be assigned to a base station within its transmission range. Our model, in which network distance is part of cost rather than a constraint, is a better match for wide-area networks like WMNs, CDNs, and gaming grids. Dealing with locality in this setting is more challenging because the potential assignment space is very large.

Workload-adaptive server selection was handled in the context of CDNs, e.g., [63]. In contrast with our approach, in which the servers collectively decide on the assignment, they chose a different solution, in which users probe the servers to make a selfish choice. The practical downside of this design is a need to either install client software, or to run probing at a dedicated tier, e.g., [56].

Local solutions of network optimization problems have been addressed starting from [81], in which the question "what can be computed locally?" was first asked by Naor and Stockmeyer. Recently, different optimization problems have been studied in the local distributed setting, e.g., Facility Location [79], Minimum Dominating Set and Maximum Independent Set [70]. While

some papers explore the tradeoff between the allowed running time and the approximation ratio (e.g., [79]), we take another approach – namely, the algorithm achieves a *given* approximation ratio, while adapting its running time and communication distance to the workload. Similar methods have been applied in related areas, e.g., fault-local self-stabilizing consensus [72], and local distributed aggregation [31]. For example, in [72], only a close neighborhood of the compromised nodes participates in the failure recovery process.

## 5.2 Definitions and System Model

For completeness of presentation, we define the min-max LDB problem (in this context, LDB for brevity). This definition also appears in Chapter 4.

Consider a set of $k$ servers $S$ and a set of $n$ user sessions $U$, such that $k \ll n$. The *network delay* function, $D : (U \times S) \rightarrow \mathbb{R}^+$, captures the network distance between a user and a server. The users and the servers do not necessarily reside in a metric space (i.e., $D$ is not necessarily subject to the triangle inequality).

Consider an assignment $\lambda : U \rightarrow S$ that maps every user to a single server. Each server $s$ has a monotonic non-decreasing *congestion delay* function, $\delta_s : \mathbb{N} \rightarrow \mathbb{R}^+$, reflecting the delay it incurs to every assigned session. For simplicity, all users incur the same load. Different servers can have different congestion delay functions. The service delay $\Delta(u, \lambda)$ of session $u$ in assignment $\lambda$ is the sum of the two delays:

$$\Delta(u, \lambda) \triangleq D(u, \lambda(u)) + \delta_{\lambda(u)}(|\{v : \lambda(v) = \lambda(u)\}|).$$

The cost of an assignment $\lambda$ is the *maximum* delay it incurs on a user:

$$\Delta^M(\lambda(U)) \triangleq \max_{u \in U} \Delta(u, \lambda).$$

The goal is to find an assignment $\lambda^*$ such that $\Delta^M(\lambda^*(U))$ is minimized. An assignment that yields the minimum cost is called *optimal*. This problem is NP-hard (Chapter 4). Our optimization goal is therefore to find a constant approximation algorithm for this problem. We denote the problem of computing an $\alpha$-approximation for LDB as $\alpha-$LDB.

We solve the $\alpha-$LDB problem in a failure-free distributed setting, in which servers can communicate directly and reliably. The set of server congestion functions $\{\delta_s\}$ is known to all servers. The network delay function $D$ is known as well, i.e., given a user's location, the network distance between this user and any one of the servers can be computed. However, the location of each user is initially known only to the closest server, i.e., the location information is not globally available.

We concentrate on synchronous protocols, whereby the execution proceeds in phases. In each phase, a server can send messages to other servers, receive messages sent by other servers in the same phase, and perform local computation. This form of presentation is chosen for simplicity, since in our context synchronizers can be used handle asynchrony (e.g., [21]).

Throughout the protocol, every server knows which users are assigned to it. At startup, every user is assigned to the closest server (this is called a NearestServer assignment). Servers can

then exchange the user information, and alter this initial assignment. Eventually, the following conditions must hold: (1) the assignment stops changing; (2) all inter-server communication stops; and (3) the assignment solves $\alpha-$LDB for a given $\alpha$.

In addition to the cost, in the distributed case we also measure for each individual server its *convergence time* (the number of phases that this server is engaged in communication), and *locality* (the number of servers that it communicates with).

## 5.3 Distributed LD-Balanced Assignment

In this section, we present two synchronous distributed algorithms, `Tree` and `Ripple`, for $\alpha-$LDB assignment. These algorithms use as a black box a centralized algorithm ALG (e.g., `BFlow` (Chapter 4) which computes an $r_{\text{ALG}}$-approximation for a given instance of the LDB problem. They are also parametrized by the *required* approximation ratio $\alpha$, which is greater or equal to $r_{\text{ALG}}$. Both algorithms assume some linear ordering of the servers, $S = \{s_1, \ldots, s_k\}$. In order to improve communication locality, it is desirable to employ a locality-preserving ordering (e.g., a Hilbert space-filling curve on a plane [82]), but this is not required for correctness.

Both `Tree` and `Ripple` partition the network into non-overlapping zones called *clusters*, and restrict user assignments to servers residing in the same cluster (we call these *internal* assignments). Every cluster contains a contiguous range of servers with respect to the given ordering. The number of servers in a cluster is called the *cluster size*.

Initially, every cluster consists of a single server. Subsequently, clusters can grow through merging. The clusters' growth is congestion-sensitive, i.e., loaded areas are surrounded by large clusters. This clustering approach balances between a centralized assignment, which requires collecting all the user information at a single site, and the nearest-server assignment, which can produce an unacceptably high cost if the distribution of users is skewed. The distance-sensitive nature of the cost function typically leads to small clusters. The cluster sizes also depend on $\alpha$: the larger $\alpha$ is, the smaller the constructed clusters are.

We call a value $\varepsilon$, such that $\alpha = (1+\varepsilon)r_{\text{ALG}}$, the algorithm's *slack factor*. A cluster is called *$\varepsilon$-improvable* with respect to ALG if the cluster's cost can be reduced by a factor of $1+\varepsilon$ by harnessing all the servers in the network for the users of this cluster alone. Note that $\varepsilon$-improvability is a locally computable property, i.e., the information required for its computation is confined to the locations of users within the cluster, and the locations and congestion functions of the servers outside it (a small-scale shared information). $\varepsilon$-improvability provides a local bound on how far this cluster's current cost can be from the optimal cost achievable with ALG. Specifically, if no cluster is $\varepsilon$-improvable, then the current local assignment is a $(1+\varepsilon)$-approximation of the centralized assignment with ALG. A cluster containing the entire network is vacuously non-improvable.

Within each cluster, a designated *leader* server collects full information, and computes the internal assignment. Under this assignment, a cluster's *cost* is defined as the maximum service delay among the users in this cluster. Only cluster leaders engage in inter-cluster communication. The distance between the communicating servers is proportional to the larger cluster's diameter. When two or more clusters merge, a leader of one of them becomes the leader of the union. `Tree` and `Ripple` differ in their merging policies, i.e., which clusters can merge (and which leaders can

communicate for that).

### 5.3.1 Tree - a Simple Distributed Algorithm

We present a simple algorithm, `Tree`, which employs a *fixed* binary hierarchy among servers. Every server belongs to level zero, every second server belongs to level one, and so forth (that is, a single server can belong to up to $\lceil \log_2 k \rceil$ levels). For $i \geq 0$ and $l > 0$, server $i \times 2^l$ is a level-$l$ *parent* of servers $2i \times 2^{l-1}$ (i.e., itself) and $(2i+1) \times 2^{l-1}$ at level $l-1$.

The algorithm proceeds in rounds. Initially, every cluster consists of a single server. During round $l > 0$, the leader of every cluster created in the previous round (i.e., a server at level $l-1$) checks whether its cluster is $\varepsilon$-improvable. If it is, the leader sends a merge request to its parent at level $l$. Upon receiving this request from at least one child, the parent server merges all its descendants into a single cluster, i.e., collects full information from these descendants, computes the internal assignment using `ALG`, and becomes the new cluster's leader. Collecting full information during a merge is implemented through a sending a query from the level-$l$ leader to all the servers in the new cluster, and collecting the replies.

A single round consists of three synchronous phases: the first phase initiates the process with a *"merge"* message (from a child to its parent), the second disseminates the *"query"* message (from a leader to all its descendants), and the third collects the *"reply"* messages (from all descendants back to the leader). Communication during the last two phases can be optimized by exploiting the fact that a server at level $l-1$ that initiates the merge already possesses full information from all the servers in its own cluster (that is, half of the servers in the new one), and hence, this information can be queried by its parent directly from it. If the same server is both the merge initiator and the new leader, this query can be eliminated altogether.

Figure 5.1(a) depicts a sample clustering of `Tree` where 16 servers reside on a $4 \times 4$ grid and are ordered using a a Hilbert curve. The small clusters did not grow because they were not improvable, and the large clusters were formed because their sub-clusters were improvable. Note that the size of each cluster is a power of 2.

`Tree` guarantees that no $\varepsilon$-improvable clusters remain at the end of some round $1 \leq L \leq \lceil \log_2 k \rceil$, and all communication ceases. We conclude the following (the proof appears in Section 5.5.1):

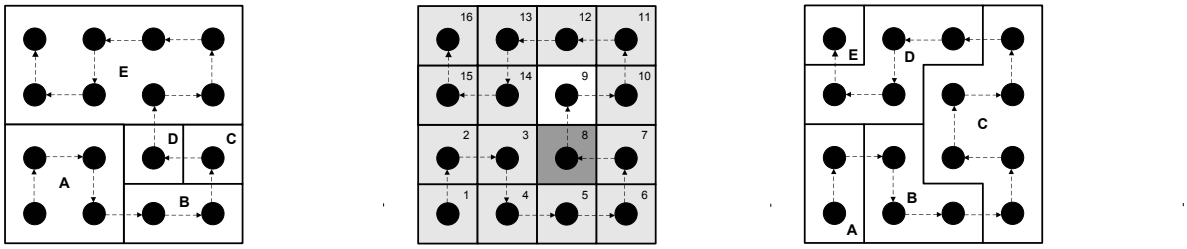
**Theorem 5.1. (`Tree`'s convergence and cost)**

1. *If the last communication round is $1 \leq L \leq \lceil \log_2 k \rceil$, then there exists an $\varepsilon$-improvable cluster of size $2^{L-1}$. The size of the largest constructed cluster is $\min(k, 2^L)$.*

2. *The final (stable) assignment's cost is an $\alpha$-approximation of the optimal cost.*

In Section 5.4, we conduct a case study which demonstrates that in practice, `Tree`'s *average* convergence time and cluster size remain nearly constant with the network's growth.

`Tree` has some shortcomings. First, it requires maintaining a hierarchy among all servers. Second, the use of this static hierarchy leads it to make sub-optimal merges. For example, a loaded cluster may have an unloaded neighbor on one side, but the rigid hierarchy causes it to merge with




(a) Sample `Tree` clustering (b) Hard workload for `Tree` (c) Sample `Ripple` clustering

Figure 5.1: **Example workloads for the algorithms and clusters formed by them in a $4 \times 4$ grid with Hilbert ordering. (a) A sample clustering $\{A, B, C, D, E\}$ produced by `Tree`. (b) A hard workload for `Tree`: $2N$ users in cell 8 (dark gray), no users in cell 9 (white), and $N$ users in every other cell (light gray). (c) A sample clustering $\{A, B, C, D, E\}$ produced by `Ripple`.**

neighbors on the other side, which are also loaded, and hence reduce its cost less. Figure 5.1(b) shows an example workload on the network in Figure 5.1(a). The congestion delay of each server is zero for a load below $N+1$, and infinite otherwise. Assume that cell 8 contains $2N$ users (depicted dark gray in the figure), cell 9 is empty of users (white), and every other cell contains $N$ users (light gray). An execution of `Tree` eventually merges the whole graph into a single cluster, for any value of $\varepsilon$, because no clustering of $s_1, \ldots, s_8$ that achieves the maximum load of at most $N$ (and hence, a finite cost) exists. Therefore, due to the rigid hierarchy, the algorithm misses the opportunity to merge $s_8$ and $s_9$ into a single cluster, and solve the problem within a small neighborhood.

### 5.3.2 Ripple - an Adaptive Distributed Algorithm

`Ripple`, a workload-adaptive algorithm, remedies the shortcomings of `Tree` by providing more flexibility in the choice of the neighboring clusters to merge with. Unlike `Tree`, in which an $\varepsilon$-improvable cluster always expands within a pre-defined hierarchy, in `Ripple`, this cluster tries to merge only with neighboring clusters of *smaller* costs. This typically results in better load-sharing, which reduces the cost compared to the previous algorithm. The clusters constructed by `Ripple` may be therefore highly unstructured (e.g., Figure 5.1(c)). The elimination of the hierarchy also introduces some challenges and race conditions between requests from different neighbors.

Section 5.3.2 makes some formal definitions and presents `Ripple` at a high level. The algorithm's technical details are provided in Section 5.3.2. Section 5.3.2 claims `Ripple`'s properties; their formal proofs appear in Section 5.5.2.

#### Overview

We introduce some definitions. A cluster is denoted $C_i$ if its current leader is $s_i$. The cluster's cost and improvability flag are denoted by $C_i.cost$ and $C_i.imp$, respectively. Two clusters $C_i$ and $C_j$ ($1 \leq i < j \leq k$) are called *neighbors* if there exists an $l$ such that server $s_l$ belongs to cluster $C_i$ and server $s_{l+1}$ belongs to cluster $C_j$. Cluster $C_i$ is said to *dominate* cluster $C_j$ if:

1. $C_i.imp =$ `true`, and

| **Message** | **Semantics** | **Size** |
|---|---|---|
| ⟨*"probe",id,cost,imp*⟩ | Assignment summary (cost and $\varepsilon$-improvability) | small, fixed |
| ⟨*"propose",id*⟩ | Proposal to join | small, fixed |
| ⟨*"accept",id,λ,nid*⟩ | Accept to join, includes full assignment information | large, depends on #users |

| **Constants** | **Value** |
|---|---|
| $L, R$, `Id` | 0, 1, the server's id |

| **Variable** | **Semantics** | **Initial value** |
|---|---|---|
| `LeaderId` | the cluster leader's id | `Id` |
| $\Lambda$ | the internal assignment | `NearestServer` |
| `cost` | the cluster's cost | $\Delta^M$(`NearestServer`) |
| `NbrId`[2] | the L/R neighbor cluster leader's id | {`Id` $-1$, `Id` $+1$} |
| `ProbeSent`[2] | *"probe"* to L/R neighbor sent? | {`false`, `false`} |
| `ProbeRecv`[2] | *"probe"* from the L/R neighbor received? | {`false`, `false`} |
| `ProposeRecv`[2] | *"propose"* from L/R neighbor received? | {`false`, `false`} |
| `ProbeFwd`[2] | need to forward *"probe"* to L/R? | {`false`, `false`} |
| `Probe`[2] | need to send *"probe"* to L/R in the next round? | {`true`, `true`} |
| `Propose`[2] | need to send *"propose"* to L/R? | {`false`, `false`} |
| `Accept`[2] | need to send *"accept"* to L/R? | {`false`, `false`} |

Table 5.1: `Ripple`**'s messages, constants, and state variables.**

2. $(C_i.cost, C_i.imp, i) > (C_j.cost, C_j.imp, j)$, in lexicographic order (*imp* and cluster index are used to break ties).

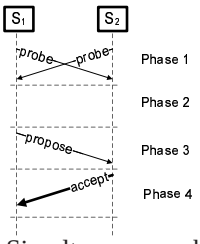
`Ripple` proceeds in rounds, each consisting of four synchronous phases. During a round, a cluster that dominates some (left or right) neighbor tries to reduce its cost by inviting this neighbor to merge with it. A cluster that dominates two neighbors can merge with both in the same round. A dominated cluster can only merge with a single neighbor and cannot split. When two clusters merge, the leader of the dominating cluster becomes the union's leader.

Dominance alone cannot be used to decide about merging clusters, because the decisions made by multiple neighbors may be conflicting. It is possible for a cluster to dominate one neighbor and be dominated by the other neighbor, or to be dominated by both neighbors. The algorithm resolves these conflicts by uniform coin-tossing. If a cluster leader has two choices, it selects one of them at random. If the chosen neighbor also has a conflict and it decides differently, no merge happens. When no cluster dominates any of its neighbors, all communication stops, and the assignment remains globally stable.

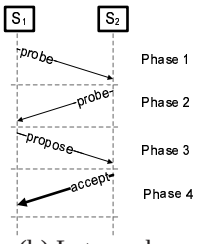
The algorithm guarantees that in every round in which communication happens, the number of clusters decreases by at least one. Moreover, since the first round in which no cluster leader sends a message, all communication stops.

### Detailed Description

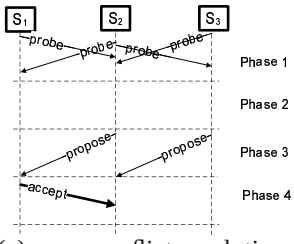
In this section, we present `Ripple`'s technical details. Table 5.1 provides a summary of the protocol's messages, constants, and state variables. See Figure 5.3 for the pseudo-code. We assume



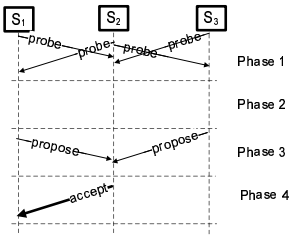

(a) Simultaneous probe: $s_1$ and $s_2$ send messages in Phase 1.



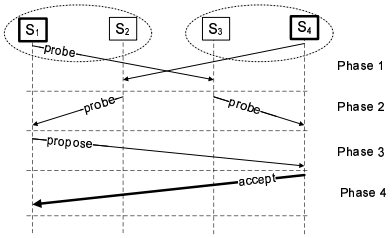

(b) Late probe: $s_2$ sends message in Phase 2.




(c) $\Leftarrow\Leftarrow$ conflict resolution: $s_2$ proposes to $s_1$ and rejects $s_3$.




(d) $\Rightarrow\Leftarrow$ conflict resolution: $s_2$ accepts $s_1$ and rejects $s_3$.




(e) Probe forwarding: $s_2$ forwards to $s_1$, $s_3$ forwards to $s_4$.

Figure 5.2: `Ripple`**'s scenarios. Nodes in solid frames are cluster leaders. Dashed ovals encircle servers in the same cluster.**

the existence of local functions $\texttt{ALG} : (U, S) \to \lambda$, $\Delta^M : \lambda \to \mathbb{R}^+$, and $\texttt{improvable} : (\lambda, \varepsilon) \to \{\texttt{true}, \texttt{false}\}$, which compute the assignment, its cost, and the improvability flag.

In each round, neighbors that do not have each other's cost and improvability data exchange *"probe"* messages with this information. Subsequently, dominating cluster leaders send *"propose"* messages to invite others to merge with them, and cluster leaders that agree respond with *"accept"* messages with full assignment information. More specifically, a round consists of four phases:

**Phase 1 - probe initiation.** A cluster leader sends a *"probe"* message to neighbor $i$ if $\texttt{Probe}[i]$ is `true` (Lines 4–5). Upon receiving a probe from a neighbor, if the cluster dominates this neighbor, the cluster's leader schedules a proposal to merge (Line 50), and also decides to send a probe to the neighbor in this direction in the next round (Line 52). If the neighbor dominates the cluster, the cluster's leader decides to accept the neighbor's proposal to merge, should it later arrive (Line 51). Figure 5.2(a) depicts a simultaneous mutual probe. If neither of two neighbors sends a probe, no further communication between these neighbors occurs during the round.

**Phase 2 - probe completion.** If a cluster leader does not send a *"probe"* message to some neighbor in Phase 1 and receives one from this neighbor, it sends a late *"probe"* in Phase 2 (Lines 13–14). Figure 5.2(b) depicts this late probe scenario. Another case that is handled during Phase 2 is probe forwarding. A *"probe"* message sent in Phase 1 can arrive to a non-leader due to a stale neighbor id at the sender. The receiver then forwards the message to its leader (Lines 17–18). Figure 5.2(e) depicts this scenario: server $s_2$ forwards a message from $s_1$ to $s_4$, and $s_3$ forwards a message from $s_4$ to $s_1$.

**Phase 3 - conflict resolution and proposal.** A cluster leader locally resolves all conflicts, by randomly choosing whether to cancel the scheduled proposal to one neighbor, or to reject the

expected proposal from one neighbor (Lines 56–65). Figures 5.2(c) and 5.2(d) illustrate the resolution scenarios. The rejection is implicit: simply, no *"accept"* is sent. Finally, the leader sends *"propose"* messages to one or two neighbors, as needed (Lines 26–27).

**Phase 4 - acceptance.** If a cluster leader receives a proposal from a neighbor and accepts this proposal, then it updates the leader id, and replies with an *"accept"* message with full information about the current assignment within the cluster, including the locations of all the users (Line 35). The message also includes the id of the leader of the neighboring cluster in the opposite direction, which is anticipated to be the new neighbor of the consuming cluster. If the neighboring cluster itself is consumed too, then this information will be stale. The latter situation is addressed by the forwarding mechanism in Phase 2, as illustrated by Figure 5.2(e). At the end of the round, a consuming cluster's leader re-computes the assignment within its cluster (Lines 67–69). Note that a merger does not necessarily improve the assignment cost, since a local assignment procedure `ALG` is not an optimal algorithm. If this happens, the assignment within each of the original clusters remains intact. If the assignment cost is reduced, then the new leader decides to send a *"probe"* message to both neighbors in the next round (Lines 70–71).

### Ripple's Properties

We now discuss `Ripple`'s properties. Their proofs appear in Section 5.5.2.

**Theorem 5.2. (`Ripple`'s convergence and cost)**

1. *Within at most $k$ rounds of* `Ripple`*, all communication ceases, and the assignment does not change.*

2. *The final (stable) assignment's cost is an $\alpha$-approximation of the optimal cost.*

Note that the theoretical upper bound on the convergence time is $k$ despite potentially conflicting coin flips. This bound is tight (see Section 5.5.2). However, the worst-case scenario is not representative. Our case study (Section 5.4) shows that in realistic scenarios, `Ripple`'s *average* convergence time and cluster size remain flat as the network grows.

For some workloads, we can prove `Ripple`'s near-optimal locality, e.g., when the workload has a single congestion peak:

**Theorem 5.3. (`Ripple`'s locality)** *Consider a workload in which server $s_i$ is the nearest server for all users. Let $C$ be the smallest non-$\varepsilon$-improvable cluster that includes $s_i$. Then, the size of the largest cluster constructed by* `Ripple` *is at most $2|C| - 1$, and the convergence time is at most $|C| - 1$.*

An immediate generalization of this claim is that if the workload is a set of isolated congestion peaks that have independent local solutions, then `Ripple` builds these solutions in parallel, and stabilizes in a time required to resolve the largest peak.

```
1:  Phase 1 {Probe initiation} :
2:    for all dir ∈ {L, R} do
3:      initState(dir)
4:      if (LeaderId = Id ∧ Probe[dir]) then
5:        send ⟨"probe", Id, cost, improvable(Λ, ε)⟩
              to NbrId[dir]
6:        ProbeSent[dir] ← true
7:        Probe[dir] ← false
8:    for all received ⟨"probe", id, cost, imp⟩ do
9:      handleProbe(id, cost, imp)

10: Phase 2 {Probe completion} :
11:   if (LeaderId = Id) then
12:     for all dir ∈ {L, R} do
13:       if (¬ProbeSent[dir] ∧ ProbeRecv[dir])
          then
14:         send ⟨"probe", Id, cost, improvable(Λ, ε)⟩
              to NbrId[dir]
15:   else
16:     for all dir ∈ {L, R} do
17:       if (ProbeFwd[dir]) then
18:         send the latest "probe" to LeaderId
19:   for all received ⟨"probe", id, cost, imp⟩ do
20:     handleProbe(id, cost, imp)

21: Phase 3 {Conflict resolution and proposal} :
22:   if (LeaderId = Id) then
23:     resolveConflicts()
24:   {Send proposals to merge}
25:   for all dir ∈ {L, R} do
26:     if (Propose[dir]) then
27:       send ⟨"propose", Id⟩ to NbrId[dir]
28:   for all received ⟨"propose", id⟩ do
29:     ProposeRecv[direction(id)] ← true

30: Phase 4 {Acceptance or rejection} :
31:   for all dir ∈ {L, R} do
32:     if (ProposeRecv(dir) ∧ Accept[dir]) then
33:       {I do not object joining with this neighbor}
34:       LeaderId ← NbrId[dir]
35:       send ⟨"accept", Id, Λ, NbrId[dir̄]⟩ to LeaderId
36:   for all received ⟨"accept", id, λ, nid⟩ do
37:     Λ ← Λ ∪ λ; cost ← Δ^M(Λ)
38:     NbrId[direction(id)] ← nid
39:   if (LeaderId = Id) then
40:     computeAssignment()

41: procedure initState(dir)
42:   ProbeSent[dir] ← ProbeRecv[dir] ← false
43:   Propose[dir] ← Accept[dir] ← false
44:   ProbeFwd[dir] ← false

45: procedure handleProbe(id, cost, imp)
46:   dir ← direction(id)
47:   ProbeRecv[dir] ← true
48:   NbrId[dir] ← id
49:   if (LeaderId = Id) then
50:     Propose[dir] ←
          dominates(Id, cost, improvable(Λ, ε), id, cost, imp)
51:     Accept[dir] ←
          dominates(id, cost, imp, Id, cost, improvable(Λ, ε))
52:     Probe[dir] ← Propose[dir]
53:   else
54:     ProbeFwd[dir] ← true

55: procedure resolveConflicts()
56:   { Resolve ⇐⇐ or ⇒⇒ conflicts}
57:   for all dir ∈ {L, R} do
58:     if (Propose[dir] ∧ Accept[dir̄]) then
59:       if (randomBit() = 0) then
60:         Propose[dir] ← false
61:       else
62:         Accept[dir̄] ← false
63:   {Resolve ⇒⇐ conflict}
64:   if (Accept[L] ∧ Accept[R]) conflicts then
65:     Accept[randomBit()] ← false

66: procedure computeAssignment()
67:   Λ′ ← ALG(Users(Λ), Servers(Λ))
68:   if (Δ^M(Λ′) < Δ^M(Λ)) then
69:     Λ ← Λ′; cost ← Δ^M(Λ′)
70:     for all dir ∈ {L, R} do
71:       Probe[dir] ← true

72: function dominates(id1, cost1, imp1,
                       id2, cost2, imp2)
73:   return (imp1 ∧
          (imp1, cost1, id1) > (imp2, cost2, id2))

74: function direction(id)
75:   return (id < Id) ? L : R
```

Figure 5.3: Ripple**'s pseudo-code: single round.**

## 5.4 Numerical Evaluation

In this section, we employ `Tree` and `Ripple` for gateway assignment in an urban WMN, using the `BFlow` centralized algorithm (Chapter 4) for local assignment. We compare our algorithms with `NearestServer`. Due to lack of space, we omit the results of comparison with perfect load-balancing, which performs much worse than `NearestServer`.

The WMN provides access to a real-time service (e.g., a network game). The mesh gateways, which are also application servers, form a rectangular grid. This topology induces a partitioning of the space into cells. The wireless backbone within each cell is a $16 \times 16$ grid of mesh routers, which route the traffic either to the gateway, or to the neighboring cells. The routers apply flow aggregation [58], thus smoothing the impact of network congestion on link latencies. Each wireless hop introduces an average delay of 6ms. The congestion delay of every gateway (in ms) is equal to the load. For example, consider a workload of 100 users uniformly distributed within a single cell, under the `NearestServer` assignment. With high probability, there is some user close to the corner of the cell. The network distance between this user and the gateway is is 16 wireless hops, incurring a network delay of $16 \times 6\text{ms} \approx 100$ms, and yielding a maximum service delay close to $100 + 100 = 200$ms (i.e., the two delay types have equal contribution).

Every experiment employs a superposition of uniform and peaky workloads. We call a normal distribution with variance $R$ around a randomly chosen point on a plane a *congestion peak*. $R$ is called the *effective radius* of this peak. Every data point is averaged over 20 runs. For instance, the maximal convergence time in the plot is an average over all runs of the maximal convergence time among all servers in individual runs.

**Sensitivity to slack factor** We first consider a 64-gateway WMN (this size will be increased in the next experiments), and evaluate how the algorithms' costs, convergence times, and locality depend on the slack factor. The workload is a mix of a uniform distribution of 6400 users with 6400 additional users in ten congestion peaks with effective radii of 200m. We consider values of $\varepsilon$ ranging from 0 to 2. The results show that both `Tree` and `Ripple` significantly improve the cost achieved by `NearestServer` (Figure 5.4(a)). For comparison, we also depict the theoretical cost guarantee of both algorithms, i.e., $(1+\varepsilon)$ times the cost of `BFlow` with global information. We see that for $\varepsilon > 0$, the algorithms' costs are well below this upper bound.

Figure 5.4(b) demonstrates how the algorithms' convergence time (in rounds) depends on the slack factor. For $\varepsilon = 0$ (the best possible approximation), the whole network eventually merges into a single cluster. We see that although theoretically `Ripple` may require 64 rounds to converge, in practice it completes in 8 rounds even with minimal slack. As expected, `Tree` converges in $\log_2 64 = 6$ rounds in this setting. Note that for $\varepsilon = 0$, `Tree`'s average convergence time is also 6 rounds (versus 2.1 for `Ripple`) because the algorithm employs broadcasting that involves all servers in every round. Both algorithms complete faster as $\varepsilon$ is increased.

Figure 5.4(c) depicts how the algorithms' average and maximal cluster sizes depend on $\varepsilon$. The average cluster size does not exceed 2.5 servers for $\varepsilon \geq 0.5$. The maximal size drops fast as $\varepsilon$ increases. Note that for the same value of $\varepsilon$, `Ripple` builds slightly larger maximal-size clusters than `Tree`, while the average cluster size is the same (hence, most clusters formed by `Ripple` are

smaller). This reflects Ripple's workload-adaptive nature: it builds bigger clusters where there is a bigger need to balance the load, and smaller ones where there is less need. This will become more pronounced as the system grows, as we shall see in the next section.

**Sensitivity to network size** Next, we explore Tree's and Ripple's scalability with the network size, for $\varepsilon = 0.5$ and the same workload as in the previous section. We gradually increase the number of gateways from 64 to 1024. Figure 5.5 depicts the results in logarithmic scale. We see that thanks to Ripple's flexibility, its cost scales better than Tree's, remaining almost constant with the network growth (Figure 5.5(a)). Note that NearestServer becomes even more inferior in large networks, since it is affected by the growth of the expected *maximum* load among all cells as the network expands.

Figure 5.5(b) and Figure 5.5(c) demonstrate that Ripple's advantage in cost does not entail longer convergence times or less locality: it converges faster and builds smaller clusters than Tree. This happens because Tree's rigid cluster construction policy becomes more costly as the network grows (the cluster sizes in the hierarchy grow exponentially).

**Sensitivity to user distribution** We study the algorithms' sensitivity to varying workload parameters, like congestion skew and the size of congested areas, for $\varepsilon = 0.5$. We first compare the cost of Tree, Ripple and NearestServer on different partitions of 12800 users between the uniform and peaky distributions, the latter consisting of ten peaks of effective radius 200m each (Figure 5.6(a)). For a uniform workload, all the algorithms achieve equal cost, because Tree and Ripple start from the nearest-server assignment and cannot improve its cost. For increasingly peaky workloads, the cost of the distributed local algorithms remains almost flat (Ripple is consistently better), while NearestServer fails to adapt to the skew.

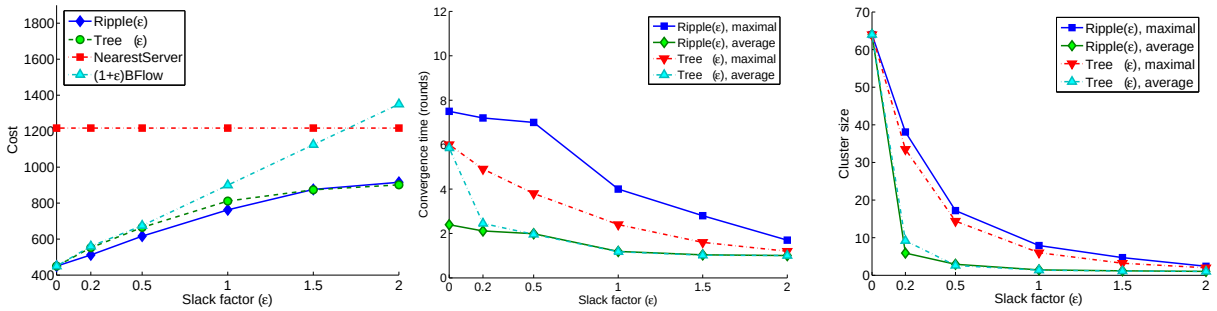
Following this, we compare Tree, Ripple and NearestServer on a workload of 12800 users concentrated in ten peaks of varying radius $500\text{m} \leq R \leq 5000\text{m}$ (see Figure 5.6(b)). For large values of $R$, this workload approaches to the uniform one, and consequently, NearestServer achieves a better cost than for more peaky distributions. Like previously, Ripple achieves a lower cost than Tree.

In both experiments, the average convergence time and average cluster size remain low and almost constant as the workload becomes more peaky (below 2.3 rounds and 2.7 servers per cluster). The respective maximal metrics grow with $p$ and $R$, which demonstrates that both algorithms build larger clusters and converge slower as the population peaks become more congested. As before, Ripple's maximal convergence time and maximal cluster size are slightly larger than Tree's, due to its load-sensitive nature.

## 5.5 Analysis and Extensions

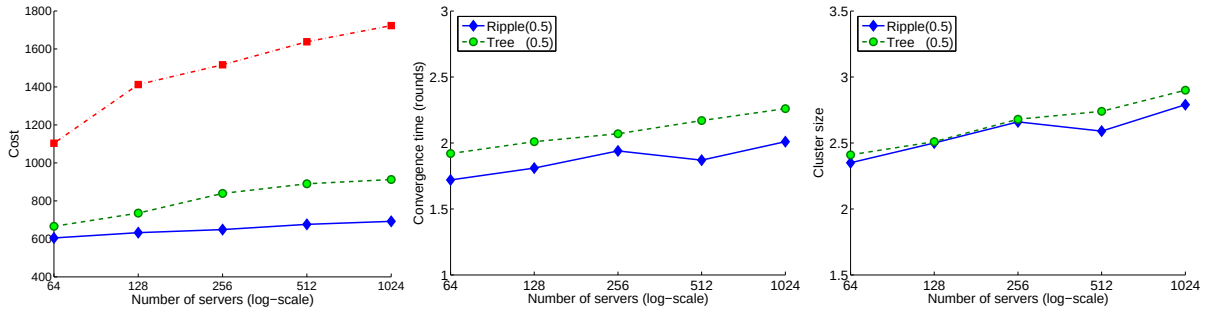
### 5.5.1 Correctness and Performance Analysis of Tree

In this section, we prove that the Tree algorithm converges in $O(\log k)$ rounds and computes an $r_{\texttt{ALG}}(1+\varepsilon)$ approximation of the optimal cost for a local assignment procedure ALG. For conve-




(a) Cost (b) Convergence time (maximal/average) (c) Cluster size (maximal/average)

Figure 5.4: **Sensitivity of `Tree`$(\varepsilon)$'s and `Ripple`$(\varepsilon)$'s cost, convergence time (rounds), and locality (cluster size) to the slack factor, for mixed user workload: 50%uniform/50%peaky (10 peaks of effective radius 200m).**




(a) Cost (b) Average convergence time (c) Average cluster size

Figure 5.5: **Scalability of `Ripple`$(0.5)$ and `Tree`$(0.5)$ with the network's size (log-scale), for mixed workload: 50% uniform/50% peaky (10 peaks of effective radius 200m).**

nience, we assume that the initial clusters are formed by the algorithm during round $i = 0$.

**Theorem 5.1. (`Tree`'s convergence and cost)**

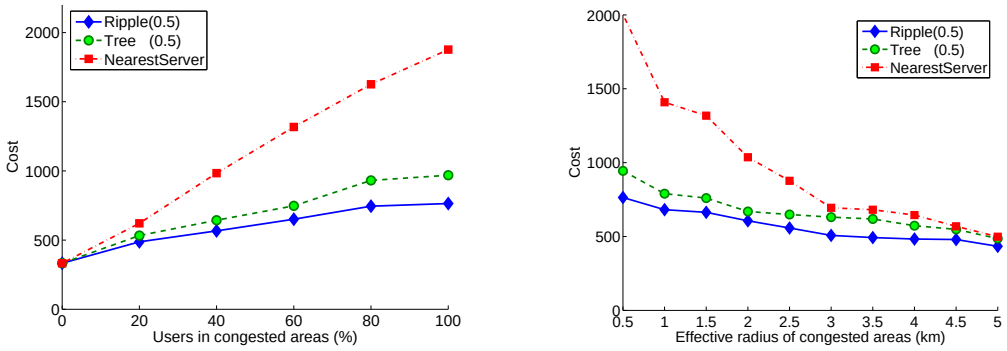
1. *If the last communication round is* $1 \leq L \leq \lceil \log_2 k \rceil$*, then there exists an* $\varepsilon$*-improvable cluster of size* $2^{L-1}$*. The size of the largest constructed cluster is* $\min(k, 2^L)$*.*

2. *The final (stable) assignment's cost is an* $\alpha$*-approximation of the optimal cost.*

*Proof :*

1. It is straightforward that the algorithm runs for at most $\lceil \log_2 k \rceil$ rounds. If the last communication round is $L > 1$, then some server sent a *"merge"* message at the beginning of this round. By the algorithm, this server must be a leader of an $\varepsilon$-improvable cluster of size $2^{L-1}$.

2. Consider cluster $C$ that has the highest cost when communication stops. The cost of this cluster is also `Tree`'s assignment cost, because the optimization goal is a min-max user cost. Either this is the only cluster in the network, or it is not $\varepsilon$-improvable. In the first case, the




(a) Cost for variable peak load

(b) Cost for variable peak size

Figure 5.6: **Sensitivity of** $\texttt{Tree}(0.5)$**'s and** $\texttt{Ripple}(0.5)$**'s cost to user distribution. (a) Varying percent of users in congestion peaks, mixed workload: (100-p)% uniform/p% peaky (10 peaks of effective radius 200m),** $0 \leq p \leq 100$**. (b) Varying effective radius of congestion peaks, mixed workload: 50% uniform/50% peaky (10 peaks of effective radius** $R$**,** $500\text{m} \leq R \leq 5000\text{m}$**).**

assignment's cost is smaller or equal to the cost of a centralized solution ALG, whereas in the second case, the assignment's cost is at most $(1+\varepsilon)$ times ALG's cost. In all cases, the algorithm's approximation factor is bounded by $\alpha = r_{\texttt{ALG}}(1+\varepsilon)$. □

### 5.5.2 Correctness and Performance Analysis of Ripple

In this section, we prove that the Ripple algorithm converges in $O(k)$ rounds and computes an $r_{\texttt{ALG}}(1+\varepsilon)$ approximation of the optimal cost for a local assignment procedure ALG. Following this, we prove Ripple's locality property, that is, if the workload contains a single congestion peak, then the algorithm does not expand the cluster around it further than required to dissipate the load.

**Lemma 5.1.** *Consider two neighboring cluster leaders $C$ or $C'$, such that $C.\texttt{Id} < C'.\texttt{Id}$. If either of them sends a* "probe" *message to the other in Phase 1 of some round $i \geq 1$, then by the end of Phase 2 of the same round:*

1. $C.\texttt{NbrId}[R] = C'$*, and* $C'.\texttt{NbrId}[L] = C$*.*
2. $C$ *and* $C'$ *receive* "probe" *messages from each other.*

*Proof :* By induction on $i$. If $i = 1$, then every cluster includes a single server, the NbrId vector is updated to its predecessor and successor in the linear order, and the *"probe"* messages are sent in both directions since $\texttt{Probe}[L] = \texttt{Probe}[R] = \texttt{true}$. Hence, these messages arrive by the end of Phase 1. If $i > 1$, consider three possible cases:

1. $C$ and $C'$ were neighbors in round $i-1$. Then, claim (1) follows from the induction hypothesis. Consider a leader (e.g., $C$) that sends the message in Phase 1. Hence, it arrives by the

end of this phase. If $C'$ does not send a *"probe"* in Phase 1, it does so in Phase 2 (Lines 13–14), and claim (2) follows.

2. $C$ and $C'$ were separated by a single cluster $\tilde{C}$ in round $i-1$. Hence, either $C$ or $C'$ merged with $\tilde{C}$ (e.g., $C$). By the induction hypothesis, after Phase 2 of round $i-1$, $\tilde{C}.\texttt{NbrId}[R] = C'$. This information appears in the *"accept"* message sent by $\tilde{C}$ to $C$ (Line 35), and hence, at the end of Phase 4 of round $i-1$, $C.\texttt{NbrId}[R] = C'$. Analogously, $C'.\texttt{NbrId}[L] = C$. Claim (2) follows as in the previous case.

3. $C$ and $C'$ were separated by two clusters, $\tilde{C}$ and $\tilde{C}'$ in round $i-1$. Then, $C$ merged with $\tilde{C}$, and $C'$ merged with $\tilde{C}'$, and they updated their neighbor pointers as follows: $C.\texttt{NbrId}[R] = \tilde{C}'$, and $C.\texttt{NbrId}[L] = \tilde{C}$. By the algorithm, both $C$ and $C'$ send *"probe"* messages to each other in round $i$. These messages arrive to $\tilde{C}'$ and $\tilde{C}$, respectively, which forward them to their correct destinations in Phase 2 (Lines 17–18). When these messages are received, the neighbor information is updated. □

**Lemma 5.2.** *Since the first round in which no cluster leader sends a message, all communication stops.*

*Proof.* Since no *"probe"* messages are sent in this round, it holds that $\texttt{Probe}[L] = \texttt{Probe}[R] = \texttt{false}$ in every cluster leader at the beginning of the round. These values do not change since no communication happens, and hence, no message is sent in the following rounds, by induction. □

We say that cluster $C$ *wishes* to merge with cluster $C$ if it either proposes $C'$ to merge, or is ready to accept a proposal from $C'$.

**Lemma 5.3.** *If there is a round $i$ since which the leaders of two neighboring clusters $C$ and $C'$ do not send messages to each other, then neither of these clusters dominates the other starting from this round.*

*Proof.* Since $C$ and $C'$ do not communicate in round $i$, the following conditions hold:

1. Neither of $C$ and $C'$ dominates the other at the beginning of round $i-1$ (lines 54–55).

2. Neither of $C$ and $C'$ reduces its cost at the end of round $i-1$ (lines 49–51).

The first condition implies one of the following two cases:

1. Neither $C$ nor $C'$ is $\varepsilon$-improvable. This property cannot change in future rounds.

2. One cluster (e.g., $C$) is $\varepsilon$-improvable, but its cost is smaller or equal the neighbor's cost. By the algorithm, neither cluster's cost grows in round $i-1$, and hence, both costs remain the same.

Therefore, neither $C$ nor $C'$ dominates its neighbor at the end of round $i$, and this property holds by induction. □

Consequently, by the end of Phase 2, both neighbors possess the same probe information. Hence, the values of `Propose` and `Accept` are evaluated correctly, and the *"propose"* and *"accept"* messages arrive to their destinations directly in a single phase.

**Lemma 5.4.** *In every round except the last one when communication happens, the number of clusters decreases by at least one.*

*Proof.* Consider a round in which some communication happens. By Lemma 5.3, at least one cluster dominates its neighbor in the previous round. Assume that no mergers occur in this round nevertheless. Consider a cluster leader $C$ that wishes to merge with its right (resp., left) neighbor $C'$. Then necessarily $C'$ wishes to merge with its own right (resp., left) neighbor, and fails too, since no mergers occur. By induction, the rightmost (resp., leftmost) cluster leader wishes to join its right (resp., left) neighbor - a contradiction. □

**Theorem 5.2. (`Ripple`'s convergence and cost)**

1. *Within at most $k$ rounds of* `Ripple`*, all communication ceases, and the assignment does not change.*
2. *The final (stable) assignment's cost is an $\alpha$-approximation of the optimal cost.*

*Proof :*

1. Assume that some message is sent in round $i \geq k$. Then, at least one message is sent during every round $j < i$, because otherwise, by Lemma 5.2, all communication would cease starting from the first round in which no messages are sent. By Lemma 5.4, at least one merger happens during every round $j < i$. Therefore, by the beginning of round $k$, at the latest, a single cluster remains, and no more communication occurs - a contradiction.
2. Consider cluster $C$ that has the highest cost when communication stops. The cost of this cluster is also `Ripple`'s assignment cost. Either this is the only cluster in the network, or it does not dominate its neighbors, by Lemma 5.3. In the first case, the assignment's cost is smaller or equal to the cost of a centralized solution `ALG`. In the second case, either the cluster is not $\varepsilon$-improvable, or it has a neighbor of equal cost that is not $\varepsilon$-improvable. Hence, the assignment's cost is at most $(1+\varepsilon)$ times `ALG`'s cost. In all cases, the algorithm's approximation factor is bounded by $\alpha = r_{\texttt{ALG}}(1+\varepsilon)$. □

The theoretical upper bound on the convergence time is tight. Consider, for example, a network in which distances are negligible, and initially, the cluster with the smallest id is heavily congested, whereas the others are empty of users. The congested cluster merges with a single neighbor in each round, due to the algorithm's communication restriction. This process takes $k-1$ rounds, until all the servers are pooled into a single cluster.

We now turn to proving the algorithm's locality property:

**Theorem 5.3. (`Ripple`'s locality)** *Consider a workload in which server $s_i$ is the nearest server for all users. Let $C$ be the smallest non-$\varepsilon$-improvable cluster that includes $s_i$. Then, the size of the largest cluster constructed by* `Ripple` *is at most $2|C|-1$, and the convergence time is at most $|C|-1$.*

*Proof.* The only cluster that can expand throughout the algorithm is $C_i$, in which $s_i$ is the leader. The other clusters, which are empty of users, are vacuously non-improvable, and hence, contain a single server each. If $C_i$ is initially non-improvable, then it will not merge with any other cluster, and the claim holds trivially. Otherwise, in every round $1 \geq j$ before the completion, $C_i$ absorbs server $s_{i-j}$, if $i - j \geq 1$, and server $s_{i+j}$, if $i + j \leq k$.

At all times, $C_i$ is uniquely identified by its set of servers (it is clear that it contains all the users). Assume that the execution stabilizes after constructing a cluster $\{s_{i-l}, \ldots, s_{i+r}\}$. By definition, $C_i$ was still $\varepsilon$-improvable after the penultimate round of communication. Therefore, its sub-clusters $C'_i = \{s_{i-l+1}, \ldots, s_i\}$ and $C''_i = \{s_i, \ldots, s_{i+r-1}\}$ are $\varepsilon$-improvable too, since removing servers from a cluster without removing any user from it cannot lead to a better local solution. Hence, either $C'_i$ or $C''_i$ are proper subsets of the smallest non-improvable cluster $C$ that includes $s_i$, and therefore, $|C| \geq \max(l, r)+1$. The final size of $C_i$ is $l+r+1 \leq 2|C|+1$. `Ripple`'s communication stops after $\max(l, r) \leq |C| - 1$ rounds. □

### 5.5.3 Handling a Dynamic Workload

For the sake of simplicity, both `Tree` and `Ripple` have been presented in a static setting. However, it is clear that the assignment must change as the users join, leave, or move, in order to meet the optimization goal. In this section, we outline how our distributed algorithms can be extended to handle this dynamic setting.

We observe that the clustering produced by `Tree` and `Ripple` is a partition of a plane into regions, where all users in a region are associated with servers in this region. As long as this spatial partition is stable, it can be employed for dynamic assignment of new users that arrive to a region. In a given region, the leader can either (1) re-arrange the internal assignment by re-running the centralized algorithm in the cluster, or (2) leave all previously assigned users on their servers, and choose assignments for new users so as to minimize the increase in the cluster's cost.

`Tree` and `Ripple` can be re-run to adjust the partition either periodically, or upon changes in the distribution of load. Simulation results in Section 5.4 suggest that the overhead of re-running both algorithms is not high. However, this approach may force many users to move, since the centralized algorithm is non-incremental. In order to reduce handoffs, we would like to avoid a global change as would occur by running the algorithm from scratch, and instead make local adjustments in areas whose load characteristics have changed.

In order to allow such local adjustments, we change the algorithms in two ways. First, we allow a cluster leader to initiate a merge whenever there is a change in the conditions that caused it not to initiate a merge in the past. That is, the merge process can resume after any number of quiet rounds. Second, we add a new cluster operation, *split*, which is initiated by a cluster leader when a previously congested cluster becomes lightly loaded, and its sub-clusters can be satisfied with internal assignments that are no longer improvable. Note that barring the future load changes, a split cluster will not re-merge, since non-improvable clusters do not initiate merges.

This dynamic approach eliminates, e.g., periodic cluster re-construction when the initial distribution of load remains stationary. Race conditions that emerge between cluster-splitting decisions and concurrent proposals to merge with the neighboring clusters can be resolved with the conflict resolution mechanism described in Section 5.3.2.

# Chapter 6

# QMesh

Wireless mesh networks, or WMNs, are a rapidly maturing technology for providing inexpensive Internet access to residential areas with limited wired connectivity [16]. While initially designed for small-scale installations (e.g., isolated neighborhoods), WMNs are now envisioned to provide citywide access and beyond through deploying thousands of access points and supporting thousands of simultaneous users [10, 41].

WMN users access the Internet through a multihop backbone of fixed wireless routers. Some of these routers, called gateways, are connected to the wired infrastructure. The WMN assigns each user to a gateway upon initial connection, and can migrate it between gateways over time. In traditional implementations, the gateways provide only Internet access. However, QoS-sensitive applications will probably be supported by high-level services at the network edge, similarly to the recent trend in wireline networks [9]. We envision a future WMN gateway that also provides application-level support, e.g., acts as a SIP proxy, a media server cache, or a full-fledged game server [42]. This trend extends the scope of the gateway assignment problem to a large variety of applications and services.

We consider gateway assignment – a traffic engineering (TE) problem that seeks optimizing the QoS or fully exploiting the network's capacity for a specific application. The solution must take into account the parameters that incur QoS degradation and additional costs, e.g., network distances and congestion, server (gateway) loads, and application-level handoffs. Mature networking systems employ TE technologies (e.g., MPLS [68]) on top of their existing routing infrastructure, to allow scalability of management. We believe that in future WMN's, traffic engineering solutions like gateway assignment will deployed atop other performance optimizations that are already in place (e.g., multiple radios [17], smart routing metrics [51], etc.).

It is common practice in small-scale WMNs to always assign a user to the nearest gateway (e.g., [18]). In this approach, gateway handoffs (macro-mobility) are tightly coupled with link-layer access point (AP) handoffs (micro-mobility). That is, when a user moves and associates with an AP that is closer to a different gateway than its current one, it automatically performs a gateway handoff too. This simple approach suffers from two drawbacks. First, it cannot adapt to load peaks within the WMN by load-balancing among multiple gateways. Second, it does not consider the application-level impact of such gateway handoffs. For example, in VoIP, handoffs are relatively low-cost, due to a small state associated with a session, whereas in online gaming, the performance

penalty of transferring the cached application state between two servers may be very high. Hence, there is a need to decouple AP transitions from gateway handoffs. While the former are purely location-based, application-transparent, and do not incur a high performance impact [18], the latter are not transparent, and should be driven by service-specific QoS considerations.

We propose QMesh (Section 6.3) – a framework for dynamically managing gateway assignments in future WMNs that can be instantiated with application-specific policies. QMesh is most beneficial for applications that allow gateway handoffs. Traditional applications that do not handle handoffs are supported, but might receive a degraded QoS. QMesh manages two types of decisions for each mobile user: (1) *when* to migrate it between two gateways, and (2) *which* gateway to choose upon a transition. QMesh employs application-specific considerations to balance the trade-off between two conflicting goals: assigning the user to a gateway that provides it with the best QoS at any given time, and reducing the number of costly gateway handoffs. QMesh does not require any extension of the underlying routing infrastructure, in particular, it does not introduce any non-scalable mechanisms like host-specific routes. Since QMesh makes decisions on a per-user basis, migrating a single user does not directly affect others, thus avoiding traffic oscillations.

QMesh manages gateway handoffs in a scalable distributed way, through a low-overhead signaling protocol that runs within the mesh transparently to the mobile user's networking stack. It *monitors* the QoS of application traffic flows to determine the handoff times, and *probes* the prospective QoS to in a shadow process to select the candidate handoff targets. Probing is scalable since it is performed within the mesh rather than separately by each user. The key to the protocol's efficiency is its adaptive approach, which performs probing (1) at distances proportional to those required for dissipating the load, and (2) at the frequency required to satisfy the QoS needs. For example, in a low-utilized mesh with little mobility, where a near gateway is likely to provide a good performance, QMesh infrequently performs very few probes limited to the close neighborhood. In contrast, if load is high and current QoS is unsatisfactory, QMesh is more aggressive in probing distant gateways more frequently.

QMesh can tolerate a gateway's failure by rapidly re-assigning its users to a backup gateway. The maintenance of backup gateways is a by-product of the probing protocol, i.e., it does not incur any additional communication overhead.

We evaluate QMesh's impact on the application QoS in a WMN through extensive simulations, mostly of VoIP but also of other real-time applications that are more handoff-sensitive (e.g., online games). The studied network topologies and mobility models are described in Section 6.4, whereas Section 6.5 extends on the assumed MAC architecture and the traffic scheduling policies. We first explore a campus-scale WMN (600 APs) with topology and mobility traces drawn from the public CRAWDAD database [2]. Since our main interest is in large-scale networks, we also study a citywide WMN (4000 APs) with highly mobile users. To this end, we experiment with two user populations: (1) a near-uniform distribution, generated by the popular random waypoint (RWP) mobility model [96], and (2) a more realistic distribution biased toward the residential centers, induced by an *alternating weighted waypoint* (AWWP) model for urban traffic [65]. The numerical results demonstrate QMesh's significant advantage over naïve nearest-gateway assignment for all workloads. The QoS achieved by QMesh is close to that of a theoretical BestMatch algorithm that uses instantaneous perfect information. Finally, we show that QMesh adjusts its overhead to

workload in a scalable way.

## 6.1 Related Work

Handoff optimizations in mobile systems have been extensively addressed since the early 1990's, mostly in the context of cellular networks (e.g., [86]). These studies primarily focused on optimizing the network capacity. Handoffs in cellular networks are triggered by physical metrics, and are handled at the link layer. The early research of mobility in 802.11 networks focused on link-layer issues, and on integration with the cellular networks (e.g., [76]). Our work is different, because we consider the network layer and above. In this context, handoffs are optional, they can improve the QoS over time, but their potential performance hit is not negligible.

Recently, Amir et al. presented a design and implementation of SMesh - a prototype WMN with mobility support [18]. They concentrated on seamless mobility of users between mesh access points. SMesh adopts the nearest-gateway handoff policy, i.e., the users of each AP are automatically assigned to the gateway closest to this AP. This approach is appropriate in a small-size installation described in that paper (about 20 access points and two gateways on the same LAN segment). However, this policy can lead to poor QoS in a wide-area mesh, as shown herein.

Many mature networking solutions address QoS optimizations as as a traffic engineering (TE) problem on top of the existing routing infrastructure (e.g., MPLS in carrier networks [68]). Almost all modern routing protocols (e.g., OSPF [80]) are traffic-independent, thus separating the concern of optimizing the QoS of individual flows to higher-level TE solutions. A different approach, adaptive QoS routing, has been actively studied by the research community (e.g., [62, 75]), originating at Gallagher's seminal work on minimum delay routing [57]. Many load-adaptive routing algorithms are designed for static or quasi-static workloads and suffer from slow convergence in highly dynamic situations. Moreover, they are complex to implement, and their behavior is hard to predict and manage. QMesh's design adopts the first approach for WMNs.

While most TE solutions optimize the unicast OoS, the problem of instantaneously optimal gateway assignment is equivalent to *anycast* routing [98] that seeks connecting each user to some service node among a given set, so as to minimize the average delay. This problem is common to multiple domains – for example, some papers pointed out the importance of joint handling of distance and load in content delivery networks [63]. However, we are not aware of any work that handles dynamic anycast of flows with mobile endpoints while considering handoff costs, and proposes scalable real-time solutions.

Adaptive probing of multiple mobile anchor points (MAPs) was proposed in the context of hierarchical mobile IPv6 routing [45]. However, in that work, handoffs are fully dictated by geography (rather than by QoS), and the simulation scale is small (a few MAPs, and a few tens of users). Ganguly et al. [58] suggested a number of VoIP performance optimizations in a WMN. In particular, they proposed maintaining the assignment of each flow to a single gateway, while constantly probing multiple user-gateway paths and opportunistically re-routing the traffic through the best path. Unlike QMesh, this approach tightly couples between gateway selection and routing, and induces non-scalable host-specific paths within the mesh.

## 6.2 Design Goals

The QMesh framework handles dynamic assignment of mobile users to WMN gateways. We pursue the following goals for this service:

- Satisfying application QoS requirements as closely as possible, in the presence of user mobility.
- Handling a variety of applications with different QoS requirements and handoff penalties.
- Tolerating infrequent gateway failures.
- Maximizing the service capacity in the presence of load peaks.
- Low-overhead, scalable, and fully distributed network management.
- No proprietary client protocol stack extensions.

## 6.3 QMesh Framework

In this section, we introduce the QMesh solution, which implements the design goals listed in Section 6.2. Section 6.3.1 outlines the QMesh network architecture, and describes the methods and parameters that must be deployed within a WMN to support QMesh. Section 6.3.2 introduces QMesh's gateway assignment protocol.

### 6.3.1 Network Architecture

QMesh provides mobile mesh users with access to real-time application services. The users perceive the WMN as a standard 802.11 LAN, and are oblivious to the mesh's internal multihop structure. At all times, each user associates at the link level with some mesh router within the radio transmission range, called the user's current AP. APs provide basic connectivity within the WMN, including address resolution and packet delivery by MAC address. As the user moves out of the radio range of its current AP, it associates with a new AP to preserve connectivity. Upon initial connection, QMesh associates each user with a single gateway, which provides it with the high-level service (e.g., Internet access, SIP proxy, or game server). QMesh may later migrate this user to a new gateway when the QoS of the original one becomes poor due to mobility or congestion, while considering an application-specific handoff penalty. QMesh gateway handoffs (macro-mobility) are completely independent of the underlying WMN's AP handoffs (micro-mobility).

Applications that seek optimal QoS must explicitly register with QMesh to receive gateway identity change notifications. This can be done through the application's standard signaling protocol, e.g., SIP. For traditional applications that cannot function correctly in the presence of gateway handoffs, QMesh can be configured to either never re-assign the gateway, or to employ tunneling through the initially assigned one (e.g., [20]), at the cost of QoS degradation. Below, we focus on the former kind of applications.

| Method | Semantics | Example | Implementation |
|---|---|---|---|
| $monitor(u)$ | return the *monitored* QoS of user $u$'s gateway. | VoIP delay/jitter | RTCP within the user's flow |
| $probe(g)$ | query the *prospective* QoS of gateway $g$ | VoIP delay/jitter | RTCP over a test connection |
| $cost(q)$ | return the *cumulative* cost incurred by the QoS measure $q$ | VoIP packet loss | |

| Parameter | Semantics |
|---|---|
| $\tau_m$ | Monitoring interval: the rate of running $monitor()$. |
| $T_{min}, T_{max}$ | The lower and upper bounds on the probing rate (the actual interval $\tau_p$ is set adaptively, depending on the QoS level). |
| $P$ | The number of simultaneous random probes (a larger $P$ can offer better QoS at the cost of higher overhead). |
| $H$ | Handoff threshold: the cumulative cost since the last transition that triggers a gateway handoff (a smaller $H$ means more aggressive handoffs). |
| $\Delta$ | QoS threshold for the probing rate control (the probes are run more frequently if the QoS is poor). |

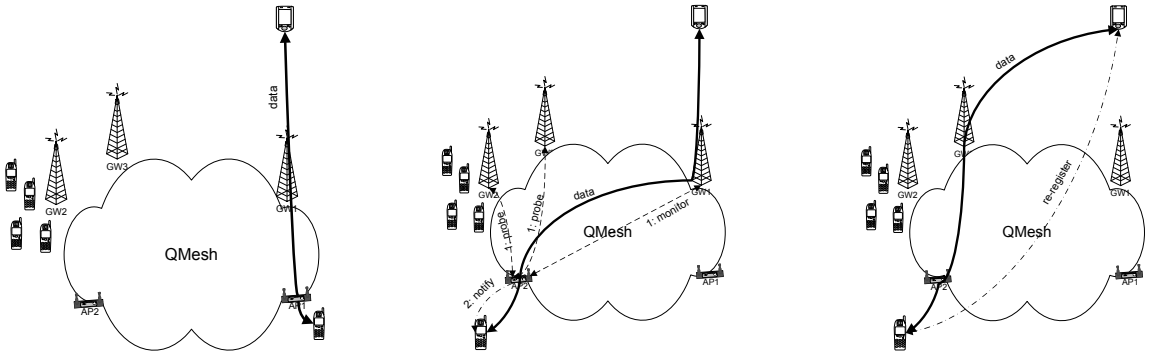
Table 6.1: **Methods and parameters deployed at the mesh nodes by applications using QMesh.**

**Application Deployment:** QMesh offers a generic framework for supporting multiple applications. The needs of each application are captured by its *service cost* which combines multiple QoS-degrading factors. This cost is accumulated over time. For example, the cost of a VoIP application can be reflected as the number of dropped or late voice packets. We distinguish between *continuous* costs, which stem from network distances and load peaks, and *one-time* costs incurred upon gateway transitions. The gateway assignment algorithm balances the tradeoff between these two kinds of cost (Section 6.3.2). Table 6.1 specifies the methods and parameters that applications using QMesh deploy at the mesh nodes.

### 6.3.2 Gateway Assignment Protocol

QMesh manages gateway handoffs in a fully distributed fashion, by running the assignment protocol independently on each mesh router. Each AP router performs the protocol on behalf of its users. Handoff management entails two kinds of decisions for each user, namely, *when* to request a gateway handoff, and *which* gateway to transition to. The first decision is driven by *monitoring* the user's recent QoS (e.g, by tracking the RTCP control packets within a VoIP media flow). The second one is based on *probing* multiple gateways (e.g., in VoIP, the AP-gateway delay can be tested over a low-bandwidth dedicated connection; in an online game, an AP can predict the average request delay by reading the response time statistics from a server, through a remote invocation of a standard application resource monitoring (ARM) API [93]). Monitoring and probing are performed by each AP in the background, transparently to the mobile users. When an AP decides to re-assign some user to a different gateway, it selects the one that offered the best QoS in the last probe.

Figure 6.1 illustrates a handoff of a media session (e.g., VoIP). The gateways provide an Internet connection service. Each gateway is attached to a different IP subnet, and functions as a NAT router. Initially, the mobile user is served by access point AP1, which associates it with gateway GW1 (Figure 6.1(a)). The second party resides in the public Internet and communicates with the




(a) Initial connection (b) After micro-mobility (c) After macro-mobility

Figure 6.1: **Handoff of a VoIP session between two NAT gateways in QMesh. (a) Initial assignment to GW1 by access point AP1. (b) Micro-mobility to access point AP2, in parallel with monitoring and probing. (c) Macro-mobility to gateway GW3. GW2 is congested and consequently not selected.**

user through GW1's IP address. The user then moves to access point AP2 (Figure 6.1(b)), which forwards its packets to GW1 over mesh links. Consequently, the packet latency is degraded. AP2 monitors the session's quality, and in parallel probes gateways GW2 and GW3 for their prospective QoS. At some point, AP2 decides to transfer the user from GW1 to GW3. GW2 is not selected despite its proximity to AP2 because it is currently congested with other users. AP2 sends a notification with GW3's IP address to the user, through the application's natural signaling protocol (e.g., SIP). In parallel, it re-routes the UDP media flow within the mesh via the new gateway (Figure 6.1(c)). The user re-registers its new IP address with its peer. Before the re-registration is complete, the peer's traffic continues to arrive to GW1, and is dropped there. This loss is the handoff cost. (In an alternative implementation, GW1 could temporarily duplicate the traffic to GW3 during the transition, in order to minimize the packet loss – e.g., [18]. In this case, the handoff cost is the number of duplicate packets.)

A handoff management algorithm must balance the tradeoff between two conflicting goals. On the one hand, it would like to always assign each user to the best gateway, in order to minimize continuous costs. On the other hand, one would like to decrease the number of handoffs, in order to reduce one-time costs. QMesh balances this tradeoff by controlling the fraction of one-time costs in the total cost. The algorithm is configured with a *handoff threshold* $H$.

QMesh monitors each user's cumulative cost since the last handoff (not inclusive), and allows a new transition only when this cost exceeds $H$. For example, if the application-dependent handoff cost is $C$, then the *total* cost of each assignment period (including the handoff in the end) is bounded by $C + H$, and therefore, the fraction of the handoff cost within the total cost is bounded by $\frac{C}{C+H}$.

The pseudocode of the QMesh assignment protocol appears in Figure 6.2. Cost monitoring (Lines 4–16) happens every $\tau_m$ time units. Once the cumulative cost of user $u$, denoted $\texttt{cost}[u]$, exceeds $H$, the user's gateway is re-assigned. $\texttt{cost}[u]$ is tracked by its current AP and sent to the new one upon an AP handoff (Lines 17–19). A gateway's failure is manifested by a rapid

accumulation of cost of all users assigned to this gateway, which triggers a fast handoff.

The AP runs the gateway selection procedure `nextchoice()` (Lines 30–45) once in $\tau_p$ time units, independently of cost monitoring. `nextchoice()` selects the next assignment for *all* local users of the same application jointly. The `best` variable holds the selected gateway's identity, and is used upon subsequent handoffs of all users served by this AP. QMesh maintains the identity of the second-best gateway, denoted $\texttt{best}_2$, to ensure failover in the case that the current optimal choice fails. This approach tolerates a single failure between two consecutive invocations of `nextchoice()`.

Waiting a long time between invocations results in using stale choices, which translates to suboptimal assignments in dynamic workloads. On the other hand, running `nextchoice()` at a high rate incurs undesirable control overhead. In order to balance between the two, each AP sets the value of $\tau_p$ *adaptively*, using the feedback on the quality of the current choice. If the QoS below a configured threshold $\Delta$, then $\tau_p$ is exponentially reduced, otherwise, it is linearly increased. The possible values of $\tau_p$ are constrained by the lower and upper bounds $T_{min}$ and $T_{max}$, respectively. If the current choice's failure is suspected, immediate re-selection is scheduled (Line 12).

Most QoS metrics are distance-sensitive, i.e., an optimal gateway is likely to be near to the user, and the primary reason for picking a remote gateway is network congestion around the close ones. Therefore, QMesh always probes the nearest gateway first, and probes further gateways only if they can help dissipating the local load. More distant gateways are probed only if moving further continues to improve QoS (which happens in case of high load peaks). Remote gateways are randomly load-balanced.

Assume that the distance between the AP and the closest gateway is $D$ network hops. The algorithm works in phases. In phase $i \geq 0$, it probes in parallel $P$ random candidates at distances $2^{i-1}D < d \leq 2^iD$ from the AP. That is, the probed nodes are drawn from concentric rings of doubling width around the AP (the empty rings are skipped, Line 34) – see Figure 6.3 for illustration. The number of rings is logarithmic with the network diameter $\psi$, and hence, the worst-case number of probes in a time unit is $\frac{P \log \psi}{T_{min}}$. Note that in the first phase, only the nearest gateway is probed. A gateway chosen multiple times is probed only once. The algorithm stops either if the result of a phase does not improve the result of the previous phases, or if all the rings are sampled.

**Discussion:** Using a small number of probes is the key to the algorithm's scalability with the network size. We later show through simulation (Section 6.4) that $P = 1$ suffices for most workloads, and the average number of probes in a time unit is very close to $\frac{2}{T_{max}}$ – far below the pessimistic upper bound. One more remarkable property is that the algorithm's fault-tolerance does not incur any additional overhead since every invocation of `nextchoice()` performs at least two probes, i.e., the maintenance of $\texttt{best}_2$ comes for free.

We further empirically show that QMesh closely approximates the unrealistic best-match assignment policy that possesses complete instantaneous information about the network state. This is in concert with Tasiulas's work [92], which demonstrated that choosing the best candidate among the current assignment and a handful of random choices is as powerful as the exhaustive search.

Note that QMesh's distributed opportunistic assignment policy cannot guarantee the best system-wide cost at all times. For example, an AP in a congested area may start choosing different gateways, thus using longer routes and amplifying the network load in other regions. In some cases,

1: **Initialization:**
2: $\tau_p \leftarrow T_{max}$
3: $\texttt{best} \leftarrow \texttt{best}_2 \leftarrow \arg\min_{g \in G} \texttt{distance}(g)$

4: {*Cost monitoring and handoff - per user*}
5: **Every** $\tau_m$ **time do for user** $u$
6: $\texttt{cost}[u] \leftarrow \texttt{cost}[u] + cost(monitor(u))$
7: $\texttt{q}[\texttt{GWID}[u]] \leftarrow monitor(u)$
8: **if** $\texttt{cost}[u] \geq H$ **then**
9: $\texttt{cost}[u] \leftarrow 0$
10: **if** $\texttt{GWID}[u] \neq \texttt{best}$ **then**
11: $\texttt{best} \leftarrow \texttt{best}_2$
12: $\tau_p \leftarrow T_{min}$
13: $\texttt{GWID}[u] \leftarrow \texttt{best}$

14: **upon AP handoff**$(u)$ **do**
15: $send(\texttt{cost}[u])$ to the new AP

16: **Every time slot do** $\tau_p$
17: {*Gateway selection - shared for all users*}
18: nextchoice()
19: {*Adjust the invocation period*}
20: **if** $(\texttt{q}[\texttt{best}] < \Delta)$ **then**
21: $\tau_p \leftarrow \max(\tau_p/2, T_{min})$
22: **else**
23: $\tau_p \leftarrow \min(\tau_p + 1, T_{max})$

24: **procedure** nextchoice()
25: $G' \leftarrow \emptyset,\ D \leftarrow \min_{g \in G} \texttt{distance}(g)$
26: **while** $(G' \neq G)$ **do**
27: $\texttt{ring} \leftarrow \{g \in G | \frac{D}{2} < \texttt{distance}(g) \leq D\}$
28: **if** $(\texttt{ring} \neq \emptyset)$ **then**
29: $\texttt{choices} \leftarrow \{P \text{ random choices from } \texttt{ring}\}$
30: $\texttt{oldbest} \leftarrow \texttt{best}$
31: $\texttt{results} \leftarrow probe(\texttt{choices}) \bigcup \{\texttt{q}[\texttt{best}]\}$
32: $(\texttt{best}, \texttt{best}_2) \leftarrow \arg\max_2 \texttt{results}[c]$
33: **if** $\texttt{best} = \texttt{oldbest}$ **then**
34: **return**
35: $D \leftarrow 2D,\ G' \leftarrow G' \bigcup \texttt{ring}$

Figure 6.2: **The QMesh gateway assignment.**

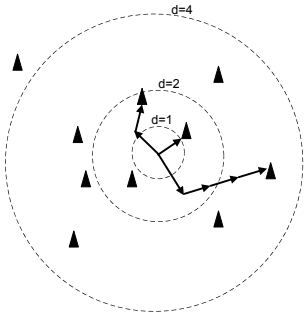



Figure 6.3: **Selecting candidates for a probe in the QMesh gateway assignment protocol. The number of random probes in each phase is $P = 1$. The selection process stops after probing the third gateway that fails to provide a better QoS than the second one.**

the network may even stabilize in an equilibrium point which is far from optimal. This problem is common to many game-theoretic scenarios (e.g., [89]). However, Section 6.4 shows that under the VoIP traffic, most of the congestion happens close to the gateways, and hence, the route length affects the network delay only weakly. Therefore, our setting is close to a load-balancing game on unrelated machines – a concept extensively studied by the theory community. A load-balancing game always converges to a Nash equilibrium point [54]. While in general this game admits arbitrarily bad equilibria, its stochastically expected operating points are near-optimal [44]. Furthermore, our simulations show that on average, QMesh does not stretch the user-gateway routes by much, and hence, the probability of the worst-case scenarios is small.

## 6.4 Evaluation

We empirically compare QMesh to alternative assignment policies, through extensive simulations. Most of our simulation focus is on VoIP. We study the algorithms' QoS and service capacity, as well as their adaptiveness to mobility and load. Section 6.4.1 presents our cost model for VoIP QoS evaluation, and Section 6.4.2 describes two policies that QMesh is compared to.

We first evaluate the protocols in a campus network with real user mobility traces extracted from a public dataset (Section 6.4.3). However, the scale of this network is around 600 APs, and a limited capacity (150 users). Therefore, we turn to simulating a projected citywide mesh (Section 6.4.4) with 4096 APs, and address two spatial distributions of mobile users: a near-uniform distribution, as induced by the widely adopted random waypoint (RWP) mobility model [96], and a more realistic distribution with load peaks in residential and business centers, produced by an Alternating Weighted Waypoint (AWWP) model of urban traffic. Finally (Section 6.4.5), we show the importance of service-specific handoff policies using an example an application which is more sensitive to handoffs (e.g., an online game).

### 6.4.1 VoIP Traffic and Cost Model

We consider RTP-over-UDP VoIP flows generated by a standard G.729 codec, i.e., a constant bit rate (CBR) flow of 50 packets per second (20ms inter-packet delay). The typical one-way delay required to sustain a normal conversation quality is 100ms [58]. A VoIP packet is considered lost if it fails to arrive to its destination within an admissible delay. We attribute most of the delay to the mesh infrastructure, and set the admissible threshold to 80 ms, thus allowing a small slack for additional delay incurred by the wired Internet.

We evaluate the VoIP QoS in terms of average packet loss ratio, which is the most dominant component in Mean Opinion Score (MOS) – the standard VoIP quality metric [7]. MOS values range from 0 to 5; values above 3.8 are considered acceptable; values above 4.0 are considered good. For a given workload, we define the service *capacity* as the maximum number of users that can be served within an acceptable MOS. In order to visualize our simple metric, we draw two MOS levels, 4.0 (corresponding to 1% of loss) and 3.8 (2% of loss) on most of our performance plots.

We focus on VoIP calls between mesh users and peers in the public Internet. In this context, a gateway handoff involves a change in the user's external IP address, and triggers application-level signaling to re-route the traffic. This results in one second of connectivity loss, during which all the VoIP packets are lost. Thus, the handoff cost is $C = 50$ (packets).

A VoIP flow starts losing packets if its path to the currently assigned gateway becomes long or congested. Excessive packet delays are the primary reason for continuous loss. Network delay is incurred by accessing the various kinds of mesh links (user, backbone, and gateway connection), and by queuing at the mesh routers. Section 6.5 extends on the models used by MeshSim. The link-level delays are characterized by the MAC architecture, whereas the queuing delays depend on the VoIP traffic scheduling policy.

In order to allow for large-scale simulations with thousands of users and access points, we developed a flow-level mesh network simulator, MeshSim [12]. Packet-level simulation tools [4, 13] cannot handle such a scale. MeshSim models the delays incurred to VoIP flows at each infrastructure node and link. It uses an accurate 802.11 link delay model [94], and implements two state-of-the-art optimizations: (1) multiple antennae at each node, with channels carefully allocated to minimize cross-link interference, and (2) VoIP aggregation (e.g., [58, 97], and also supported by the 802.11n standard). We describe MeshSim in more detail in Section 6.5.

### 6.4.2 Assignment Policies

We compare QMesh to two simple assignment policies, NearestGateway and BestMatch. NearestGateway assigns the user to the gateway closest to its current AP. That is, gateway handoffs are tightly bound to AP transitions. The BestMatch policy is a realistically impossible variant of QMesh, which runs the greedy selection procedure upon every AP handoff request, and assumes instantaneous correct information. That is, it performs an exhaustive search of the best candidate rather than random sampling of one, and moreover never uses stale information.

QMesh and BestMatch are instantiated with cumulative packet loss as the QoS cost function. The handoff threshold is set to $H = 10$ packets. This relatively small value is chosen because the

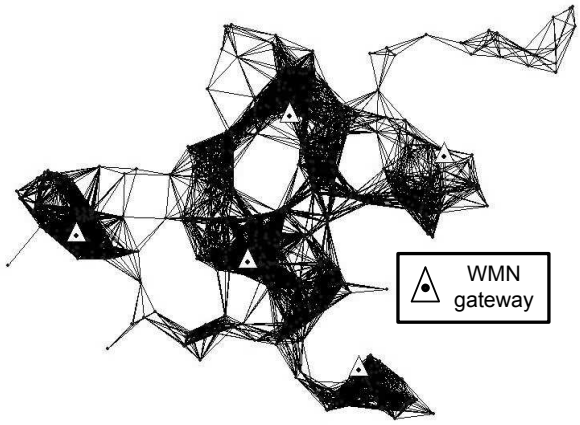
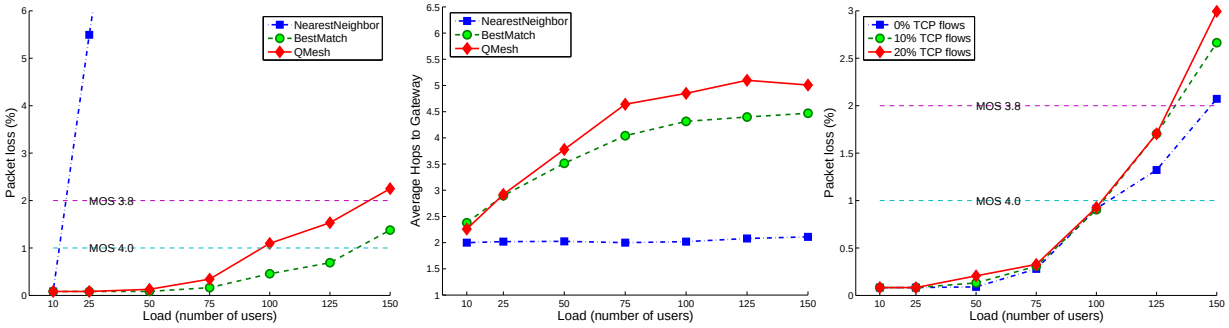



Figure 6.4: **The Dartmouth network map and gateway placement.**




(a) Loss ratio scalability: BestMatch and QMesh vs NearestGateway

(b) User-gateway distance adaptation by BestMatch and QMesh

(c) Loss ratio scalability: QMesh in the presence of TCP flows

Figure 6.5: **Scalability evaluation of the gateway assignment algorithms in an unplanned campus WMN, with topology and user mobility traces drawn from the Dartmouth CRAWDAD public dataset.**

handoff cost is low ($C = 50$ packets), and given the user speeds, the loss of 10 packets is a sufficient indication for changing the assignment. QMesh uses a single probe in each phase of `nextchoice()` (i.e., $P = 1$). It adaptively adjusts the interval between invocations of `nextchoice()` within the range $[T_{min} = 1\text{sec}, T_{max} = 15\text{sec}]$. The QoS threshold for accelerating the probes is $\Delta = 50$ ms.

### 6.4.3 Campus Scale Simulation (CRAWDAD)

Our first case study is mobile VoIP performance in an unplanned mesh deployed within a large neighborhood or a campus. We draw the network topology and the mobile users' motion traces from CRAWDAD [2], a community resource for archiving wireless data at Dartmouth college, thus avoiding the need to speculate about the simulation's input. The original Dartmouth network is a single-hop WLAN. The network includes over 600 irregularly placed access points. While in a WLAN, APs are connected via a wired infrastructure, in our WMN setting, they communicate

through wireless interfaces. All routers use omnidirectional antennas with a transmission radius of 133m – a minimal value for which the network remains connected. We place the Internet gateways in a way that minimizes the mean distance (in the number of hops) from each AP to the nearest gateway. For this purpose, the network is partitioned into 5 clusters using a K-Means algorithm [66], and within each cluster, the router closest to the centroid as selected to serve as a gateway. Figure 6.4 illustrates the WMN's topology (the campus map is due to [3]). The APs are depicted as dark dots, and the selected gateways as triangles with a dot in the middle.

We employ the 2001–2003 movement dataset [90] that contains the mobility traces of more than 6200 users, collected over a period of many months. Each trace contains a sequence of (timestamp, AP id) pairs that describe the history of the user's associations with wireless APs. The majority of users are either static or quasi-static (occasionally hopping between close APs once in a few minutes) most of the time. Their locations are heavily biased toward the faculty buildings.

We explore the scalability of the assignment policies of with network load, as follows. For each data point $L$, we build a set of scenarios in which $L$ users generate a continuous VoIP stream, as follows. We extract from the trace a set of time intervals, all at least 10 minutes long, in which the number of online users is exactly $L$. Since the database is very large, each set contains hundreds of intervals for each $L$. We simulate NearestGateway, BestMatch and QMesh on the traces of 50 intervals selected uniformly at random from each set, and average the loss rates among the runs. Figure 6.5(a) depicts the results. The loss of BestMatch and QMesh remains acceptable as long as the number of users does not exceed 125 (the service capacity). Therefore, in the absence of mobility, BestMatch and QMesh efficiently balance the costs incurred by network distances and gateway loads. Only under high loads, some differentiation between the two appears, because the latter searches for the candidate more carefully and locates it immediately. On the other hand, NearestGateway cannot handle even 25 users, due to its inability to exploit multiple gateways. QMesh's adaptive nature becomes even more pronounced as we study the dependency between the congestion and the user-gateway distances (Figure 6.5(b)). For small loads, QMesh and Nearest-Gateway produce an identical average distance of 2.1, while for high loads, QMesh stretches the routes to 4.9 to optimize the assignment.

Following this, we examine QMesh's scalability in the presence of concurrent TCP flows generated by traditional data applications. We repeat the previous experiment, for a varying number of TCP connections (0% to 20% of the number of users, with the rest running VoIP flows). All TCP flows are handled in a traditional way, namely, each of them is initially assigned to the closest gateway, and never reassigned again. In order to prevent starvation of the VoIP traffic by TCP flows, we allocate the latter with at most 50% of available transmission bandwidth, and schedule their packets at a lower priority. Thus, the VoIP capacity of the shared links decreases, but the QoS of the admitted flows is guarantee. Figure 6.5(c) shows that the average loss ratio increases with the fraction of TCP flows, but the impact is not dramatic within the admissible load range.

### 6.4.4 City Scale Simulation

Our ultimate goal is studying the performance of QMesh in a very large-scale WMN with highly mobile users. For this, we turn to simulating a citywide mesh that exceeds the campus deployment by an order of magnitude in the spanned area and the population.

We consider an urban geography of size $8 \times 8$ km$^2$. There are five population areas – four residential neighborhoods and a commercial downtown. User locations within each area follow a Gaussian distribution around the area's center with variance $\sigma$, which is called the area's *effective radius*. The downtown's effective radius is 1km, and its center is co-located with the center of the grid at coordinates (4km, 4km). Each neighborhood's effective radius is 500m, and their centers are located at coordinates (1km,1km), (1km, 7km), (7km, 1km), and (7km, 7km). Figure 6.6(a) depicts this topology. Areas are depicted as circles, and gateways as small triangles. The Internet access is provided through a regular grid of 64 gateways, spaced 1km apart. The wireless backbone is a fine grid of 4096 mesh routers, spaced 125m apart. The transmission radius is 125m.

Our simulation employs two stationary distributions of mobile users, each generated by a different mobility model:

1. A near-uniform distribution, produced by the popular random waypoint model (RWP) [96]. The node uniformly chooses the destination and moves toward it at a constant speed $v = 20$ m/s (an urban driving speed).

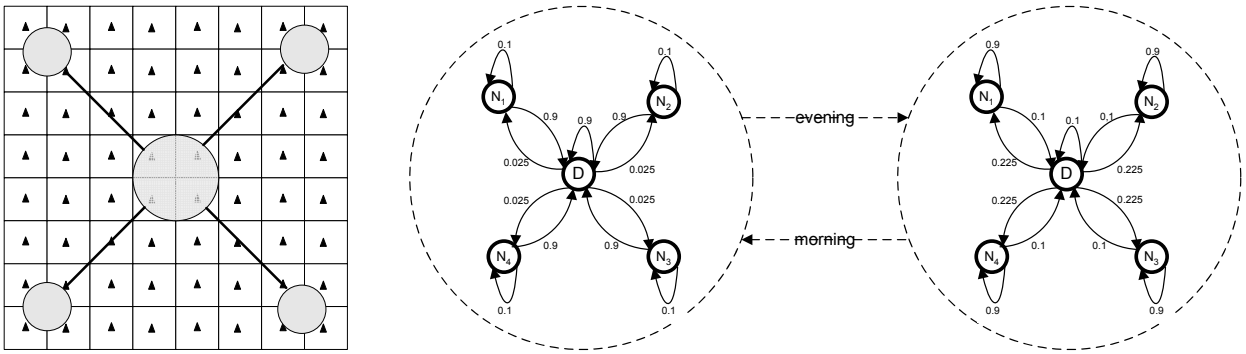
2. A more realistic distribution that biases the users toward the population areas (e.g., neighborhoods or downtown), produced by the projected alternating weighted waypoint (AWWP) model. At any given time, a mobile node is either stationary in some area, or moving on a highway between two areas at a constant speed $v = 20$ m/s. The popularity of different areas varies during the day.

### The Alternating Weighted Waypoint Model

AWWP is one plausible way to create a clustered user distribution. It is inspired in part by [65], which explored preferences in choosing destinations of pedestrian mobility patterns. The nodes' transitions between the areas are governed by a Markov process that switches its transition probability matrix every 12 hours. The system is modeled by two super-states, each of which is a Markov chain. Each state in a chain corresponds to a single area. Each probability matrix designates the users' preferred locations at a certain time of day. The moving node's destination point within the target area is a random variable, drawn from the Gaussian distribution described above. In the morning, most users drive to the downtown and stay there during the working hours, whereas in the evening, most users drive back to their neighborhood and stay at home during the night. Direct transitions between the neighborhoods are not allowed.

Figure 6.6(b) depicts this random process. We denote the downtown by $D$, and neighborhood $i$ by $N_i$. The transition probabilities are (symmetric for all $i$):

| | **Morning/day** | **Evening/night** |
|---|---|---|
| $p_{N_i,D}$ | 0.9 | 0.1 |
| $p_{D,N_i}$ | 0.025 | 0.225 |
| $p_{D,D}$ | 0.9 | 0.1 |
| $p_{N_i,N_i}$ | 0.1 | 0.9 |
| $p_{N_i,N_j}$ | 0 | 0 |




(a) City topology (b) Alternating Weighted Waypoint

Figure 6.6: **Urban Simulation Settings: (a) The city's topology (downtown and four neighborhoods) and the gateway grid. (b) The random process behind the AWWP mobility model.**

The stationary distributions of the Markov chains are:

| | **Morning/day** | **Evening/night** |
|---|---|---|
| $\pi_D$ | 0.9 | 0.1 |
| $\pi_{N_i}$ | 0.025 | 0.225 |

A mobile user's behavior is deterministic between transition times. Upon a self-transition, a node remains at its current location for a period of $t$. In case of a transition of the user to another area, it picks a destination point from the distribution induced by the destination area, and moves to it with a speed of $v$. For simplicity, we assume that all users wait for the same time and move with the same speed. We set $t = 4$ min. Note that the waiting time is equal to the driving time between the centers of the downtown and neighborhood areas. In this setting, the motion can be approximated as a discrete-time Markov chain, in which the time slot length is 4 min. All state transitions (including the probability matrix switch) happen on slot boundaries. During a single slot, the user either moves between two areas, or remains in one of them.

In each super-state (day or night), the users are mostly stationary, except in a short time after the transition, when they mostly move to their new preferred areas. Upon switching the super-state, the convergence to a new matrix's stationary distribution is short (3-4 time slots). Therefore, the 15 min following the super-state transition are considered a *transition period*, after which the system enters a *stable* period.

We also experimented with richer models, e.g., non-straight movement trajectories, and constrained motion within the population areas. However, they yield almost the same results because the most important factor is the load peaks. Hence, our simulations focus on the presented simple model.

### Numerical Results

We compare the loss rates and overhead of QMesh to BestMatch and NearestGateway, for the near-uniform and skewed stationary distributions produced by the RWP and AWWP mobility models,

respectively. Every data point is averaged over 20 runs. For AWWP, we separately study four different times of day: morning (neighborhoods-to-downtown movement), day (mostly staying in the downtown), evening (downtown-to-neighborhoods movement), and night (mostly staying in the neighborhoods). Day and night are stable periods, morning and evening are transition. The morning and evening scenarios are simulated for 15 min (the transition period time, see Section 6.4.4). The day and night scenarios are insensitive to the measurement period; we used 30 min periods for them. The RWP experiments were initialized with the uniform distribution of users, and preserved it over time [96]. Each experiment simulated 15 min of user motion.

We first study the the dependency between load and loss for the three algorithms. Figure 6.7 depicts their behavior for near-uniform distribution induced by the RWP mobility pattern, with loads ranging from 200 to 2000 users. At all times, NearestGateway succeeds in accommodating each user at the closest gateway, because no cell's load exceeds its capacity. All loss is due to handoffs, and depends only on the user's speed, and hence, it is constant for all loads. The BestMatch and QMesh policies incur identical costs, since upon a handoff, the local gateway is almost always the best choice that cannot be improved by further probing. They improve the loss over NearestGatewayby sustaining a user's association with its gateway beyond the grid cell's boundaries, as long as the QoS permits. The maximal admissible user-gateway distance diminishes with load, and hence, handoffs become more frequent, thus causing BestMatch's and QMesh's loss.

The shortcomings of NearestGateway become evident as we apply the same experiment for a more realistic biased distribution of load generated by the AWWP model. We separately explore the morning scenario featuring a transition of load from the periphery to the center (Figure 6.8(a)), and the day scenario that reflects a stationary congestion in the downtown (Figure 6.8(c)). In both cases, NearestGateway does not scale beyond 300 users due to its inability to resolve the congestion in the downtown area to the other gateways. On the other hand, QMesh can accommodate 600 users – just slightly below the baseline BestMatch. Figure 6.8(b) differentiates the part of handoffs in the packet loss (by depicting the average handoff frequency), in the morning scenario. QMesh's frequency is low and congestion-adaptive (growing slowly with load), while NearestGateway's is high and load-insensitive.

In the next experiments, we continue using the more challenging AWWP workload. Figure 6.9(a) depicts the distribution of costs achieved by NearestGateway, BestMatch and QMesh by the time of day, for a load of 600 users. Note that NearestGateway's loss is even higher during the day than in the morning, due to the stationary congestion in the downtown. The price of this congestion is higher than the cost of excessive handoffs during the morning transition. Since the measured transition period also captures some resting time in the steady-state area for most nodes, NearestGateway's loss in the morning is higher than in the evening, when these areas are not congested.

The same disadvantage of NearestGateway is observed when we examine the relationship between a user's mobility level (the fraction of time in which the user changes its location) and its loss rate. Figure 6.9(b) and Figure 6.9(c) depict the distribution of loss among the mostly stationary users (below 20% mobility) and the mostly mobile ones (above 20%) achieved by NearestGateway and QMesh, respectively. (Note that a small fraction of users remains highly mobile even in the stable regime, since transitions between population centers are not instantaneous). QMesh

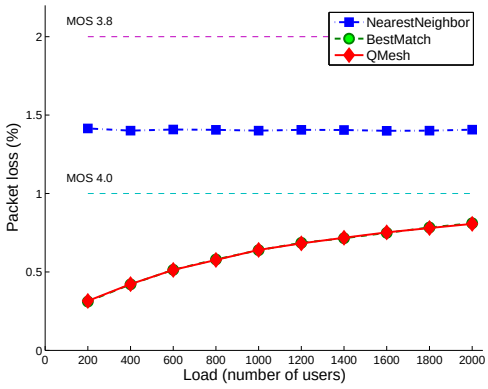
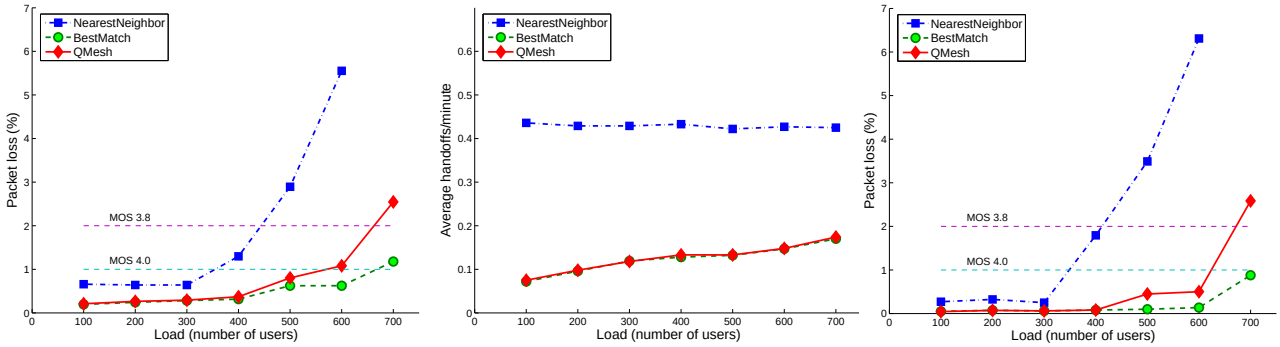



Figure 6.7: **Scalability evaluation of the gateway assignment algorithms in a citywide WMN, for a near-uniform distribution (RWP model).**

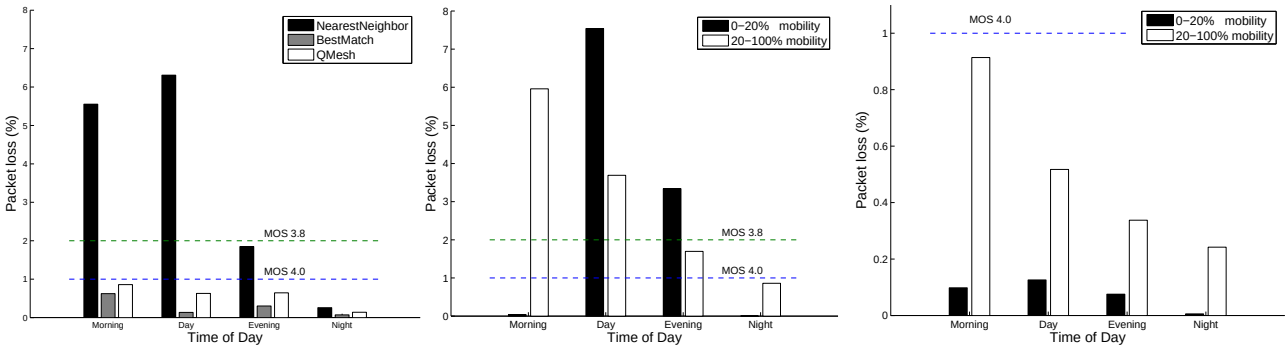



(a) Loss ratio, transition period (b) Handoff frequency, transition period (c) Loss ratio, stable state

Figure 6.8: **Scalability evaluation for a clustered distribution (AWWP model): (a) Loss ratio – morning. (b) Handoff frequency (average number of handoffs per minute) – morning. (c) Loss ratio – day.**

has the desirable property that the stationary users experience smaller loss rates than the mobile ones. That is, most of the mobile users' packet loss stems from handoffs (which do not happen to the stationary users), while the congestion-oriented loss is minimized for both categories thanks to opportunistic assignment. In contrast, under NearestGateway, stationary users in congested areas suffer from continuous loss, which exceeds the occasional handoff-related loss incurred to mobile users.

We study the distribution of load on mesh links, in order to show that under the QMesh assignment, the network operates close to its optimal equilibrium point. We focus on $20\%$ most congested grid cells in a stable state (day), for 600 users. Figure 6.10 depicts the dependency between a link's distance from a gateway and the average number of VoIP flows assigned to this link. Note that most of the load is concentrated on WMN links adjacent to gateways (QMesh and BestMatch perform very close). VoIP packet aggregation (Section 6.5) further reduces the number of physical flows that contend for the same link. Therefore, all hops of any user-to-gateway route are unlikely to be congested, except for the last one, i.e., long user-gateway routes do not create




(a) QMesh vs NearestGateway and BestMatch

(b) NearestGateway: stationary vs mobile users

(c) QMesh: stationary vs mobile users

Figure 6.9: **Average loss ratio distribution by the time of day, for the a skewed workload of 600 users (AWWP mobility model): (a) Comparison between 3 assignment policies, (b,c) Comparison between the mostly stationary (below 20% mobility) and the mostly mobile users, for two separate policies.**

new bottlenecks in the system. Hence, the distributed operation of QMesh is close to that of selfish load-balancing, which is expected to stabilize in near-optimal configurations [44].

Following this, we examine QMesh's control overhead – the average number of probes per minute performed by each AP. We focus on the day scenario when the network congestion is most heavy. The overhead depends on the number of probes per selection as well as on the probing rate. Our measurements show that for most values of load, it is enough to apply `nextchoice()` once in 15 seconds to achieve an acceptable loss ratio. The average number of probes applied upon gateway selection never exceeds 2.5, as opposed to the theoretical limit of the logarithm of the network size. Moreover, for most values of the load, the number of probes is almost exactly 2 – the minimal possible value. Figure 6.11(a) summarizes these results in a single plot, which shows that the overhead is very small for most workloads.

Finally, we study the potential QoS benefit of increasing the number of random probes made by QMesh. We compare two instantiations of the algorithm using $P = 1$ and $P = 2$, in the day scenario. Figure 6.11(b) shows that increasing $P$ does not bring any performance impact for light loads (below 400), and has a minor impact for heavy loads. Moreover, QMesh partially masks the disadvantage of applying a single random probe by adaptively adjusting the probing interval $\tau_p$. Note that at a load of 600, QMesh with $P = 1$ starts increasing its probing rate due to QoS degradation, which reduces the gap between it and QMesh with $P = 2$.

### 6.4.5 Service-Specific Handoff Policies

In all the above experiments, QMesh used a very low handoff threshold, and migrated each user almost immediately as the user's delay became inadmissible. Setting a low threshold ($H = 10$) was correct because the handoff cost was also low ($C = 50$), and hence, there was no benefit in delaying the new assignment. However, this policy is not necessarily true if the handoff cost is very high, e.g., in an online game, in which a handoff entails a substantial state transfer. Consider,

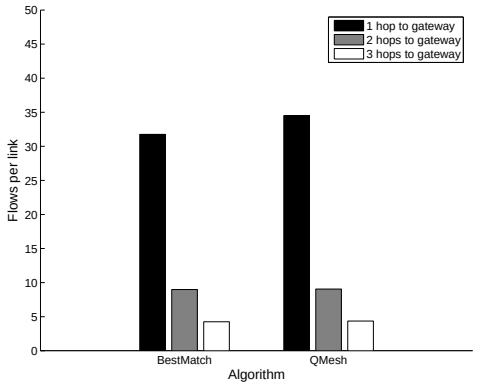
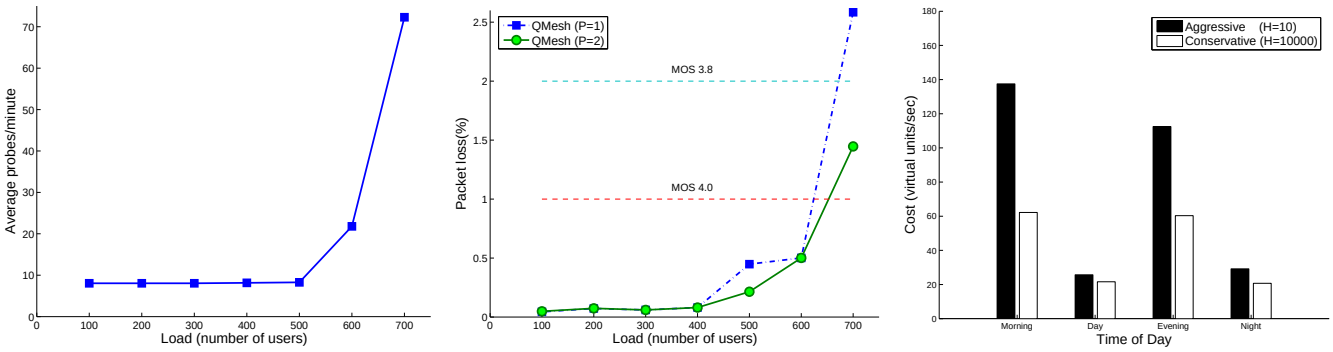



Figure 6.10: **Load distribution on mesh links in congested areas: most of the congestion happens close to the gateways.**




(a) Scalability of the number of probes (b) Impact of $P$ on the loss rate (c) Impact of $H$ on the loss rate

Figure 6.11: **Studying the effect of QMesh's tuning parameters: (a) Scalability of the number of probes per minute with load, (b) Impact of increasing the number of simultaneous probes $P$. (c) Impact of handoff threshold for an application with a high handoff cost (50000): aggressive policy ($H = 10$) vs. conservative policy ($H = 10000$).**

for example, the same traffic model as described in Section 6.4.1, the same continuous cost (1 lost packet = 1 unit), and the handoff cost of $C' = 50000$ units. We provide this example for insight only, and do not claim that a realistic online game's traffic/cost model is used.

Figure 6.11(c) illustrates the comparison between two instances of QMesh parametrized by $H = 10$ and $H = 10000$, respectively, under a light load (400 users). The second instance, which is much more conservative in applying costly handoffs, consistently achieves a better cost with all mobility patterns. Hence, tuning the handoff threshold in accordance with the application-specific handoff cost is crucial for achieving a good overall cost.

## 6.5 Delay Modeling in MeshSim

We briefly describe the delay models used by MeshSim, a flow-level simulator we developed to provide network scalability beyond that of packet-level simulation tools [13, 4].

**MAC Architecture and Link Delays:** We assume that each router is equipped with distinct interfaces for user access (802.11b) and backbone (802.11a) communication. These interfaces use different wireless bands, and hence, the access and backhaul traffic flows do not interfere. 802.11a is chosen for its abundance of orthogonal wireless channels (12), which are exploited to minimize interference among the mesh links (this is also a common practice in commercial WMNs [10]). A router employs two cards for communicating within the mesh - one for egress traffic and the other for ingress traffic. This facilitates a parallel transmission and reception at the backbone, and hence, a simultaneous upstream and downstream forwarding. The ingress interface is operated at a fixed wireless channel. Whenever a router needs to communicate with some neighbor, it switches its egress interface to the channel of this neighbor's ingress card. Hence, a single ingress interface is shared by the links emerging from the router's neighbors.

The low-degree topologies utilized by our experiments[1] and a substantial number of available channels allow performing ingress channel assignment in a way that no pair of routers within two hops from each other share the same ingress channel. Therefore, the only kind of MAC contention at the backbone arises when two nodes simultaneously transmit to the same neighbor. That is, we assume that no interference exists between two backbone links without a common endpoint.

Since at each mesh node, all the incoming backbone links share the same ingress interface, the delay on each outgoing link depends on the cumulative load on this link's target. The mesh forwards each flow along the shortest path between its AP and gateway. Therefore, a particular assignment of users to gateways determines the load on each link, and hence, the total link delay incurred to each user. We use the model by Tickoo and Sikdar [94] to compute the expected latency of traversing a shared 802.11 link (either access or backbone).

**VoIP Aggregation and Queueing Delays:** We assume that VoIP flow aggregation(e.g., [58], also adopted by 802.11n) is employed in order to overcome the capacity limitation that is inherent to wireless VoIP, namely, a high overhead of transmitting small packets over the 802.11 medium. The VoIP traffic at mesh routers is handled through a VoIP-specific scheduling policy. A packet that needs to be forwarded over an egress link is placed into the queue of this link. The link's scheduler sets the time for transmitting the next outgoing packet. At this time, the queued packets are aggregated into a super-packet, which is transmitted over the medium as a single frame. Upon arrival to the neighbor, the super-packet is de-multiplexed, and the individual packets are handled independently.

By rate-limiting the super-packet generation process, the scheduler controls the capacity/delay tradeoff at the wireless link. The scheduler transmits a single packet in a fixed-length time slot, which can be implemented, e.g., through a simple token-based traffic shaping. With this policy, if the arrival rate exceeds the transmission rate, the packets are queued on average for a half-slot time, and otherwise, they are forwarded immediately.

[1] A sparse subnetwork of the Dartmouth WMN (Section 6.4.3) or a grid (Section 6.4.4)

In the chosen delay model [94], a link can sustain an inter-packet delay of 20ms for at most 10 independent flows without dropping packets. For the backbone links, we take a conservative approach, and rate-limit each egress queue to one packet in 10ms. Since the maximal node degree is 4, at most 8 (aggregated) packets contend for each shared ingress link in 20ms, thus approximating the behavior of eight concurrent VoIP flows. The average queueing time is therefore 5ms for a fully backlogged egress queue.

The maximal capacity of the backbone links is constrained by the number of RTP packets that can be multiplexed into a single super-packet. The size of an RTP packet with a G.729 voice payload is 60 bytes. Assuming the super-packet size of 1500 bytes, without RTP header compression [58], the number of voice packets that can be multiplexed into a super-packet is 25. Since a single egress queue schedules transmissions each 10ms (twice the packet arrival rate in a single flow), its capacity is $2 \times 25 = 50$. Hence, the capacity of a shared ingress link is $4 \times 50 = 200$ flows (4.7 Mbps bandwidth).

Finally, the gateway connection introduces its own delay, which depends on the wired link's capacity. Since a typical WMN is expected to use an available inexpensive wired infrastructure, assume the use of the ADSL technology, in which the uplink is the bandwidth bottleneck. The fastest available ADSL2 uplink rate today is 3.5 Mbps. We assume that it supports 120 flows (2.75 Mbps effective bandwidth), and employ the M/M/1 model for delay calculation.

# Chapter 7

# QMesh Implementation

Wireless mesh networks, or WMNs, is a rapidly maturing technology for providing inexpensive Internet access to residential areas with limited wired connectivity [16]. While initially designed for small-scale installations (e.g., isolated neighborhoods), WMNs are now envisioned to provide citywide access and beyond [15]. Modern mesh networks are expected to handle mobile applications with diverse QoS requirements like VoIP, VoD, and gaming [58].

WMN users access the Internet through a multihop backbone of fixed wireless routers. Each external user associates at all times with a single router that provides it with access to the mesh, which is called the users access point, or AP. Some of the routers, called gateways, are connected to the wired infrastructure. A common practice in small-scale WMNs is always assigning each user to the nearest gateway (e.g., [18]). In this approach, gateway handoffs (macro-mobility) are tightly coupled with link-layer AP handoffs (micro-mobility). This solution cannot adapt to load peaks within the mesh, thus limiting its capacity.

This shortcoming can be resolved by assigning *some* users from congested areas to distant gateways, hence avoiding congested paths, providing an improved quality of service (QoS), and eventually increasing the WMN's capacity. Intelligent gateway assignment policies must balance between the impact of link loads and network distances – in other words, perform *load-distance balancing* [38]. Note that gateway selection is a traffic engineering policy, rather than a routing extension. It can work on top of any routing protocol within the WMN.

We designed and implemented QMesh (Section 7.1) – a prototype QoS mesh network that features seamless mobility support and load-distance balancing. QMesh's external users perform a minimum of standard configurations, without installing additional software at their side. The QMesh infrastructure is based on inexpensive Windows XP desktops equipped with wireless cards, which makes it an attractive choice for office environments. The routing software deployed on the infrastructure nodes is a small-footprint device driver (to the best of our knowledge, this is the first WMN solution implemented in the Win32 kernel space). QMesh is managed by a centralized controller, which intelligently associates wireless users with access points and gateways. The QMesh code (driver and management software) and documentation are available for download at [8].

QMesh was deployed on a testbed of 7 mesh nodes, including two gateways. It supports a variety of real-life applications, including VoIP and video streaming. Performance measurements

(Section 7.2) validate our approach to mobility and user assignment.

## 7.1 QMesh Architecture

The QMesh routing software is implemented on top of the Mesh Connectivity Layer (MCL) – an ad-hoc routing and link quality measurement software package developed at Microsoft Research that features the LQSR routing protocol [5, 51]. Architecturally, the MCL code is a Win32 NDIS driver that elegantly plugs into the host networking stack between the network and link layers. It abstracts the WMN's multihop nature from upper-layer software, which handles the entire mesh as a single L2 segment. MCL requires installing its code on all network nodes. QMesh extends it with an access infrastructure functionality, namely, with MAC address resolution and unicast/broadcast traffic forwarding for non-LQSR users.

The QMesh controller is a user-space software that runs on a selected mesh router, and communicates with the other routers through LQSR extensions. It collects the wireless user location information from the access points, and associates every WMN user with a single AP and a single gateway. The controller can be instantiated with multiple assignment policies, encompassing nearest-neighbor assignment, perfect load-balancing, and more sophisticated algorithms that consider distance and load together (e.g., [38]). Fig. 7.1 illustrates the QMesh architecture.

### 7.1.1 Seamless Mobility

In QMesh, the mobile user's current AP functions as its default IP router. The user is forced to route all its traffic via this AP (a sandbox subnet) by setting the subnet mask to 255.255.255.255. The two nodes communicate directly, through a 802.11 ad-hoc link. (The alternative of implementing APs as transparent bridges operating in the 802.11 infrastructure mode was infeasible, due to a shortcoming of most Win32 wireless card drivers that do not support the promiscuous mode – the same problem was reported in [5]).

The assignment mechanism works as follows. As a mobile user initially associates with the mesh or moves away from its original AP, it gets discovered by one or more APs that intercept the user's broadcast control traffic - e.g., periodical DHCP requests. These APs enter the user's MAC address into their *local user cache*, or LUC, which they periodically send to the controller. The latter computes the (possibly new) assignment, and disseminates it in the network. All WMN nodes store the user-AP associations in a *global user cache*, or GUC, to maintain address resolution within the mesh infrastructure segment. We explored 3 methods of communicating the AP association back to the mobile node, seamlessly to the user:

**Gratuitous ARP:** originally suggested in [18]. All mobile users perceive the WMN as an omnipresent virtual access point. Its IP address is pre-configured by the user. Upon the initial association or handoff, the prospective access point manipulates the mapping of this virtual IP address to a MAC address, through publishing its own link-layer address in an unsolicited address resolution (ARP) reply (Figure 7.2(a)). The downside of this approach is that ARP is a low-level protocol that cannot be secured (e.g., encrypted).

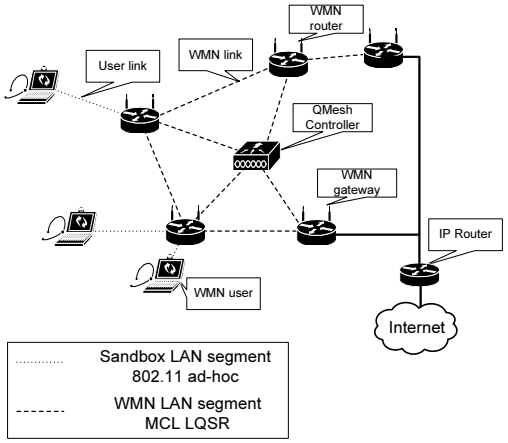



Figure 7.1: **The QMesh network architecture: users, mesh routers, and a centralized controller.**

**ICMP Router Discovery Protocol (IRDP):** manipulating the default router's IP address itself [50]. The mesh AP assigned to the user publishes its own network address as the user's default gateway, using a specific ICMP packet. IRDP can be enabled at a Windows computer through a dedicated DHCP request.

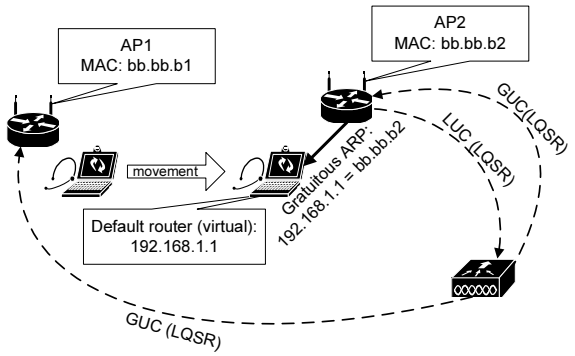
**DHCP Reconfigure:** manipulation of the default gateway's IP address through a dynamic update triggered by the DHCP server [95]. This option is not supported by the Windows XP host networking stack, and we chose not to implement it.

Unlike the previous implementations (e.g., [18]), QMesh does not employ any reliable messaging infrastructure for forwarding in-flight packets during the AP transition. Instead, we opt for a simple and lower-latency kernel-level implementation. Our performance measurements validate this approach.

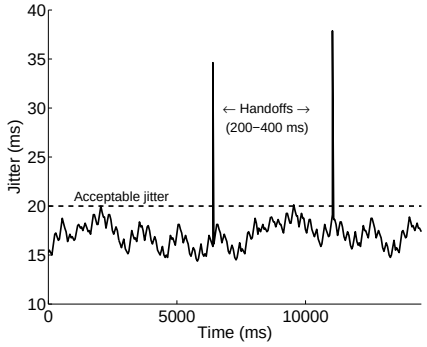
## 7.2 Performance Evaluation

We first study the performance impact of access point handoffs, as follows. We measure the fluctuations of jitter in a G.711 VoIP stream emerging from a mobile node upon two AP transitions. The jitter values stabilize in the acceptable range (below 20 ms) within 200-400 ms (Figure 7.2(b)), thus supporting the findings in previous WMN implementations [18, 58].

The next experiment demonstrates the importance of balancing loads and distances in user assignment. We measure the Mean Opinion Score (MOS) – the standard VoIP quality metric that combines the loss rate, jitter and delay experienced by the flow's packets [7]. MOS values range from 0 to 5; values above 4.0 are assumed good. We consider a setting in which up to five wireless users are closer to one access point, which is also a gateway, than to any other mesh node. Therefore, assigning them to this nearest neighbor (the setup is depicted in Figure 7.3(a)) results in overloading the access link, and hence, in a degraded MOS. On the other hand, routing some flows through a more distant AP/gateway pair reduces the congestion, at the expense of an increased

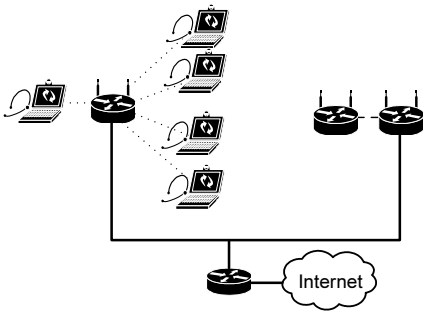



(a) Handoff via Gratuitous ARP

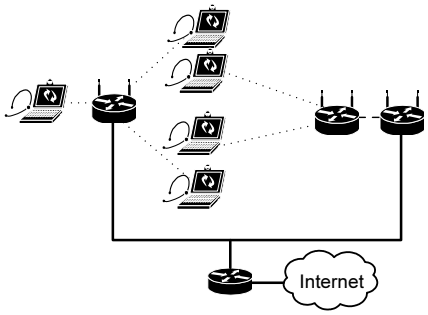



(b) Impact of handoff (VoIP jitter)

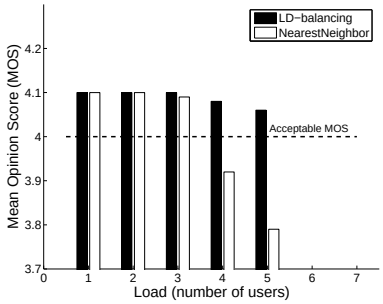
Figure 7.2: **AP handoff management in QMesh: (a) Gratuitous ARP-based handoff mechanism. (b) Fluctuations of VoIP jitter caused by AP handoffs.**




(a) Nearest-Neighbor assignment




(b) Load-distance balanced assignment




(c) Impact of load-distance balancing

Figure 7.3: **Comparison of the (a) nearest-neighbor and (b) load-distance balancing assignment policies, for the VoIP application, in terms of the Mean Opinion Score (MOS) metric.**

number of hops (Figure 7.3(b)). The measurements depicted in Figure 7.3(c) show that the second option can sustain all five flows within an acceptable quality, while the first one can handle only three.

# Chapter 8

# Conclusions and Future Work

We explored a distributed infrastructure for QoS provisioning to mobile users and groups thereof, through multiple geographically dispersed service points. This Mobility and Group Management Architecture (aka MAGMA) can prove highly valuable to many next-generation mobile networking technologies, ranging from wireless mesh networks (WMNs) for broadband Internet access to IP multimedia subsystems (IMS) for beyond-3G converged cellular/IP networks. In this context, the key challenges were *adaptiveness* (i.e., handling dynamic phenomena such as user mobility, flash crowds etc), and *scalability* (i.e., coping with millions of users through thousands of service points). We focused on vital algorithmic aspects of this framework, as well as on prototype implementations of our algorithms in real systems.

## 8.1 Conclusions

Chapter 3 studied a problem of service point assignment to mobile users or user groups in a distributed infrastructure with multiple service points. This problem will naturally arise in several emerging practical environments, in which the cost of stateful application handoffs is significant. We have provided a rigorous theoretical study, which includes competitive online algorithms and a lower bound on the competitive ratio of deterministic algorithms. Following this, we studied the performance of the proposed algorithms when applied in an urban WMN and in a wide-area chatroom service. We gave practical algorithms that approximate the optimal performance more closely, and scale well with the network size. Finally, we demonstrated that a very limited, and even noisy, prediction of the user's future motion allows to construct algorithms with near-optimal performance.

Chapter 5 introduced a novel load-distance balancing (LDB) problem, which is important for delay-sensitive service access networks with multiple servers. In such settings, the service delay consists of a network delay, which depends on network distance, and a congestion delay, which arises from server load. The problem seeks to minimize the maximum service delay among all users. The $\alpha-$LDB extension of this problem is achieve a desired $\alpha$-approximation of the optimal solution. We presented two scalable distributed algorithms for $\alpha-$LDB, Tree and Ripple, which compute a load-distance-balanced assignment with local information. We studied Tree's

and `Ripple`'s practical performance in a large-scale WMN, and showed that the convergence times and communication requirements of these algorithms are both scalable and workload-adaptive, i.e., they depend on the skew of congestion within the network and the size of congested areas, rather than the network size. Both algorithms are greatly superior to previously known solutions. `Tree` employs a fixed hierarchy among the servers, whereas `Ripple` requires no pre-defined infrastructure, scales better, and consistently achieves a lower cost.

Chapter 6 introduced QMesh, a novel scalable solution for dynamic assignment of mobile users to gateways in a large-scale WMN, which jointly takes into consideration factors like load peaks, mobility, and application-specific handoff costs. QMesh can be instantiated with application-specific handoff policies. We studied QMesh through extensive simulation in different settings of a wide-area urban WMN. Our results show that QMesh scales well (constant to logarithmic overhead) and adapts to network loads. It satisfies application QoS requirements for service capacities significantly exceeding those of traditional policies.

Finally, Chapter 7 presented a prototype implementation of QMesh within the Win32 kernel that features (1) native support of standard 802.11 clients, (2) transparent mobility, and (3) platform for intelligent user-to-gateway assignment. Performance evaluation conducted over a real testbed demonstrates the feasibility of QMesh's approach to handoffs, as well as the importance of balancing distances and loads in assigning users to WMN gateways.

## 8.2 Future Work

The work on MAGMA can be extended to a variety of new research directions.

### 8.2.1 Dynamic Infrastructure Deployment

MAGMA explored a fixed service infrastructure, e.g., static WMN gateways. However, many new applications require extending this perception. For example, rescue force applications require rapidly deploying an infrastructure for supporting mobile teams. In this context, the speed of self-configuration (including the services) is critical. The deployed infrastructure can be either externally deployed, or use part of the users as supernodes. This new model implies a host of novel service placement and selection problems, which we briefly sketch below.

**LD-balanced Clustering Problems:** Consider, e.g., the following extension of the load-distance balancing problem (Chapter 5). In the previously studied model, server locations were known a-priori – e.g., located on a grid, or selected via clustering based on distances between users (Chapter 6). This model can be extended to *jointly* perform service placement and assignment – a novel variation of the well-studied K-center, K-median and K-means clustering problems [78]. The key algorithmic issue is whether a joint service placement and assignment can improve upon the applying these steps separately (the latter approach was adopted, e.g., by [47]).

For example, the LD-balanced $K$-center problem is defined as follows. Consider $n$ users $U = \{u_1, \ldots, u_n\}$ in a (metric) space, in which $D(u_i, u_j)$ stands for a network distance between $u_i$ and $u_j$. A *service k-partition* is defined as a set of pairs $\{(U_1, s_1), \ldots, (U_k, s_k)\}$, such that (1) $s_i \in U_i$

for all $i$, (2) $U_i \bigcap U_j = \emptyset$ for all $i \neq j$, and (3) $\bigcup_{1 \leq i \leq k} U_i = U$. In this context, user $s_i$ is selected to serve all users in $U_i$ (including itself).

Consider a non-decreasing congestion function $\delta : \mathbb{N} \rightarrow \mathbb{R}^+$, which is uniform among all users. A partition is *optimal* if it minimizes

$$\max_{1 \leq i \leq k} [\delta(|U_i|) + \max_{u \in U_i} D(u, s_i)]$$

(note that the use of $\delta$ differentiates between this problem and the traditional $K$-center problem). Using the $L_1$ and $L_2$ norms instead of $L_\infty$ leads to similar definitions of LD-balanced $K$-median and $K$-means problems. An initial result indicates that an optimal clustering is not necessarily spatially convex [35].

**Local and Mobile LD-balanced Clustering:** Similarly to Chapter 5, we are looking for scalable local distributed solutions for LD-balanced clustering problems. Recently, multiple works addressed distributed mobile clustering without considering the impact of load, e.g., [60, 71]).

A particularly interesting direction is proposing *mobility-adaptive* solutions, which trade clustering perturbation for the amount of user motion. Intuitively, a system designer could expect that a small variation in user locations will result in small changes incurred to clustering. However, this expectation cannot be fulfilled if the system is close to its maximal capacity, even when the server locations are known in advance.

Relaxing the requirement for selecting exactly $k$ centers (i.e., allowing up to $(1 + \varepsilon)k$ servers) can be critical for maintaining a smooth LD-balanced clustering even in the presence of mobility. In this context, we plan to capitalize on recent results from the computational geometry community. For example, Har-Peled's work on clustering motion [64] demonstrates how to achieve a relaxed $k$-clustering of points whose movement is described by polynomial functions, under traditional definitions. This paper, as well as other works (e.g., [85]), uses a *coreset* technique to improve the scalability of computation-intensive clustering algorithms. Instead of running an algorithm on the entire input, it selects its small representative subset, called coreset, and computes an approximate solution of the original problem using it. We propose to explore the applicability of coresets to LD-balanced clustering. Yet another technique that can be borrowed from computational geometry for efficient mobile center management is kinetic data structures (KDS) [60]. KDS allow maintaining the mobile center's trajectory and speed (also termed as flight plan) when the mobile nodes' flight plans are either known in advance, or change at discrete times.

**Churn-Resilient Service Placement:** Spontaneous node joins and crashes, also called *churn*, are a commonplace phenomenon in dynamic distributed systems. Contrast to mobility, which leads to predictable changes in workload, churn (i.e., flash crowds) can be very hectic. *Gossip* protocols [53, 84], an efficient mechanism for disseminating data in a dynamic network, can be applied to spread the locally monitored churn indications. The gossip approach can be of independent value even for one-shot problems like distributed load-distance balancing, in which it can be used instead of the clustering approach adopted in Chapter 5.

### 8.2.2 New Approaches to Old Problems

**Analysis of Greedy LD-Balanced Assignment:** Chapter 5 presented a 2-approximation algorithm, `BFlow`, for the load-distance balancing problem. `BFlow`, which is based on multiple maximum flow computations in a bipartite user-server graph, has a large (although polynomial) time complexity. We examined the following alternative heuristic, which is both simple and fast: traverse a random permutation of users, and greedily assign each user to the server that minimizes this user's delay. Empirical experimentation showed that this heuristic consistently produced better results than `BFlow`, for all the studied workloads. A rigorous analysis is important for understanding this behavior. Currently, we can only analyze two extreme cases – a uniform workload, and a peaky workload (but not the combination of the two).

**Analysis of the QMesh Assignment Policy:** We would like to to theoretically justify the heuristic distributed probing policy employed by QMesh (Chapter 6), which combines randomized probing at growing distances with greedy server choices. It would interesting to analyze this policy from the game-theoretic view, demonstrate its price of anarchy (PoA) and price of stability (PoS) metrics [89], and confirm the empirical observation that QMesh always converges to a stable assignment.

# Bibliography


[1] Cisco Airespace Wireless Control System. http://www.cisco.com/univercd/cc/td/doc/product/wireless/wcs/index.htm.

[2] CRAWDAD: a Community Resource for Archiving Wireless Data at Dartmouth. http://www.crawdad.cs.dartmouth.edu.

[3] Dartmouth Maps. http://www.dartmouth.edu/~maps.

[4] JiST - Java in Simulation Time/SWANS - Scalable Ad hoc Network Simulator. http://jist.ece.cornell.edu.

[5] Mesh Networking: Software Artifacts and Support. http://research.microsoft.com/mesh.

[6] Minimum Exact Cover. http://www.nada.kth.se/~viggo/wwwcompendium/node147.html

[7] MOS Calculator. http://davidwall.com/MOSCalc.htm.

[8] QMesh - a Mesh Network with QoS Support. http://comnet.technion.ac.il/magma/software/qmesh/index.htm.

[9] Riverbed Technology. http://www.riverbed.com.

[10] Strix Systems. http://www.strixsystems.com.

[11] The MAGMA Research Project. http://comnet.technion.ac.il/magma.

[12] The MeshSim Simulator and Traces. http://comnet.technion.ac.il/~ebortnik/software/meshsim.tar.gz.

[13] The Network Simulator – ns-2. http://www.isi.edu/nsnam/ns.

[14] The WiMax Forum. http://wimaxforum.org.

[15] Tropos Networks. http://www.tropos.com.

[16] I. Akylidiz, X. Wang, and W. Wang. Wireless Mesh Networks: a Survey. *Computer Networks Journal (Elsevier)*, Mar. 2005.

[17] M. Alicherry, R. Bhatia, and L. E. Li. Joint Channel Assignment and Routing for Throughput Optimization in Multi-Radio Wireless Mesh Networks. *ACM MobiCom*, 2005.

[18] Y. Amir, C. Danilov, M. Hilsdale, R. Musaloiu-Elefteri, and N. Rivera. Fast Handoff for Seamless Wireless Mesh Networks. *ACM MobiSys*, Oct. 2006.

[19] Y. Amir, C. Danilov, M. Hilsdale, R. Musaloiu-Elefteri, and N. Rivera. Fast Handoff for Seamless Wireless Mesh Networks. *MobiSys*, 2006.

[20] Y. Amir, C. Danilov, R. Musaloiu-Elefteri, and N. Rivera. An Inter-domain Routing Protocol for Multi-homed Wireless Mesh Networks. *IEEE WoWMoM*, 2007.

[21] B. Awerbuch. On the Complexity of Network Synchronization. *JACM*, 32:804–823, Oct. 1985.

[22] A. Bar-Noy and I. Kessler. Tracking Mobile Users in Wireless Communication Networks. *IEEE INFOCOM*, pages 1232–1239, 1993.

[23] A. Bar-Noy, I. Kessler, and M. Sidi. Mobile Users: to Update or not to Update? *IEEE INFOCOM*, pages 570–576, 1994.

[24] A. Bar-Noy and I. Mansour. Competitive On-line Paging Strategies for Mobile Users under Delay Constraints. *ACM PODC*, pages 256–265, 2004.

[25] A. Barak, S. Guday, and R. Wheeler. The MOSIX Distributed Operating System, Load Balancing for UNIX. *Lecture Notes in Computer Science, Springer Verlag, vol 672*, 1993.

[26] Y. Bartal, A. Blum, C. Burch, and A. Tomkins. A polylog(n)-Competitive Algorithm for Metrical Task Systems. *ACM STOC*, pages 711–719, 1997.

[27] Y. Bejerano and I. Cidon. An Anchor Chain Scheme for IP Mobility Management. *Wireless Networks*, pages 409–420, 2003.

[28] Y. Bejerano, I. Cidon, and J. Naor. Dynamic Session Management for Static and Mobile Users: a Competitive On-Line Algorithmic Approach (Part II). *ACM DIAL-M*, 2000.

[29] Y. Bejerano and S.-J. Han. Cell Breathing Techniques for Balancing the Access Point Load in Wireless LANs. *IEEE Infocom*, 2006.

[30] S. Bespamyatnikh, B.Bhattacharya, D.Kirkpatrick, and M.Segal. Mobile Facility Location. *ACM Workshop on Foundations of Mobile Computing (DIALM)*, 2000.

[31] Y. Birk, I. Keidar, L. Liss, A. Schuster, and R. Wolff. Veracity Radius – Capturing the Locality of Distributed Computations. *ACM PODC*, 2006.

[32] S. Biswas and R. Morris. ExOR: Opportunistic Multi-Hop Routing for Wireless Networks. *ACM SIGCOMM*, 2005.

[33] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[34] A. Borodin, N. Linial, and M. Saks. An Optimal On-Line Algorithms for Metrical Task System. *JACM*, 39:745–763, 1992.

[35] E. Bortnikov. Dynamic Service Management in Infrastructure-Based Mobile Networks. Technical report, Technion, EE Faculty, Jan. 2006.

[36] E. Bortnikov, I. Cidon, and I. Keidar. Nomadic Service Points. *IEEE Infocom*, 2006.

[37] E. Bortnikov, I. Cidon, and I. Keidar. Nomadic Service Assignment. *IEEE TMC*, 6(8), Aug. 2007.

[38] E. Bortnikov, I. Cidon, and I. Keidar. Scalable Load-Distance Balancing. In *International Symposium on Distributed Computing (DISC)*, 2007.

[39] E. Bortnikov, I. Cidon, and I. Keidar. Scalable Real-time Gateway Assignment in Mobile Mesh Networks. *ACM CoNEXT*, Dec. 2007.

[40] E. Bortnikov, I. Cidon, I. Keidar, T. Kol, and A. Vaisman. Poster Abstract: A QoS Mesh Network with Mobility Support. *ACM SIGMOBILE MC2R*, Jan. 2008.

[41] J. Camp, J. Robinson, C. Steger, and E. Knightly. Measurement Driven Deployment of a Two-Tier Urban Mesh Access Network. *ACM MobiSys*, Oct. 2006.

[42] J. Chen, B. Knutsson, B. Wu, H. Lu, M. Delap, and C. Amza. Locality Aware Dynamic Load Management form Massively Multiplayer Games. *PPoPP*, 2005.

[43] M.-H. Chiu and M. A. Bassiouni. Predictive Schemes for Handoff Prioritization in Cellular Networks Based on Mobile Positioning. *IEEE JSAC*, 18(3):510–522, 2000.

[44] C. Chung, K., L. K. Pruhs, and A. Roth. The Price of Stochastic Anarchy. *SAGT*, 2008.

[45] W. Chung and S. Lee. Improving Performance of HMIPv6 Networks with Adaptive MAP Selection Scheme. *IEICE Transactions on Communications*, E90-B(4), 2007.

[46] R. Cohen and G. Nakibli. On the Computational Complexity and Effectiveness of N-hub Shortest-Path Routing. *IEEE Infocom*, 2004.

[47] R. Cohen and G. Nakibli. A Traffic Engineering Approach for Placement and Selection of Network Services. *IEEE Infocom*, 2007.

[48] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 2001.

[49] S. K. Das, R. Jayaram, and S. K. Sen. An Optimistic Quality-of-Service Provisioning Scheme for Cellular Networks. *IEEE ICDCS*, 1997.

[50] S. Deering. ICMP Router Discovery Messages. Internet-draft, IETF, June 1991. rfc1256.txt http://www.ietf.org/rfc/rfc1256.txt.

[51] R. Draves, J. Padhye, and B. Zill. Routing in Multi-radio, Multi-hop Wireless Mesh Networks. *ACM MobiCom*, Sept. 2004.

[52] L. Du, J.Bigham, and L. Cuthbert. A Bubble Oscillation Algorithm for Distributed Geographic Load Balancing in Mobile Networks. *IEEE Infocom*, 2004.

[53] A. D. et al. Epidemic algorithms for replicated database management. *ACM PODC*, 1987.

[54] E. Even-Dar, A. Kesselman, and Y. Mansour. Convergence Time to Nash Equilibrium in Load Balancing. *ACM Transactions on Algorithms*, 3(3), Aug. 2007.

[55] A. Fiat and M. Mendel. Better Algorithms for Unfair Metrical Task Systems and Applications. *SIAM J. Computing*, (6):1403–1422, 2003.

[56] M. J. Freedman, K. Lakshminaraynan, and D. Maizieres. OASIS: Anycast for Any Service. *ACM NSDI*, 2006. To appear.

[57] R. G. Gallagher. A Minimum Delay Routing Algorithm Using Distributed Computation. *IEEE ToC*, 25:73–84, 1977.

[58] S. Ganguly, V. Navda, K. Kim, A. Kashyap, D. Niculescu, R. Izmailov, S. Hong, and S. Das. Performance Optimizations for VoIP Services in Mesh Networks. *JSAC*, 24:2147–2158, Nov. 2006.

[59] J. Gao, L. Gubias, J. Hershberger, L. Zhang, and A. Zhu. Discrete Mobile Centers. *ACM Symp. on Computational Geometry*, 2001.

[60] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete mobile centers. In *Symposium on Computational Geometry*, pages 188–196, 2001.

[61] B. Ghosh, F.T.Leighton, B.Maggs, S.Muthukrishnan, G. Plaxton, R. Rajaraman, A. Richa, R. Tarjan, and D. Zuckerman. Tight Analyses of Two Local Load Balancing Algorithms. *ACM STOC*, 1995.

[62] R. A. Guerin and A. Orda. QoS Routing in Networks with Inaccurate Information: Theory and Algorithms. *IEEE/ACM ToN*, 1999.

[63] K. M. Hanna, N. N. Nandini, and B. N. Levine. Evaluation of a Novel Two-Step Server Selection Metric. *IEEE ICNP*, 2001.

[64] S. Har-Peled. Clustering Motion. *Discrete and Computational Geometry*, 31(4):545–565, 2004.

[65] W. Hsu, K. Merchant, H. Shu, C. Hsu, and A. Helmy. Weighted Waypoint Mobility Model and its Impact on Ad Hoc Networks. *ACM SIGMOBILE MC2R*, 9:59–63, 2005.

[66] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[67] S. Jamin, C. Jin, D. Raz, and Y. Shavitt. Constrained Mirror Placement on the Internet. *IEEE Infocom*, 2001.

[68] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tight Rope: Responsive yet Stable Traffic Engineering. *ACM SIGCOMM*, 2005.

[69] R. Karrer, A. Sabharwal, and E. Knightly. Enabling Large-scale Wireless Broadband: The Case for TAPs. *Proceedings of HotNets*, 2003.

[70] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhoffer. Local Approximation Schemes for Ad Hoc and Sensor Networks. *ACM DIALM-POMC*, 2005.

[71] F. Kuhn, T. Moscibroda, and R. Wattenhofer. Fault-Tolerant Clustering in Ad Hoc and Sensor Networks. (68), 2006.

[72] S. Kutten and D. Peleg. Fault-Local Distributed Mending. *J. Algorithms*, 1999.

[73] N. Lavi, I. Cidon, and I.Keidar. MaGMA: Mobility and Group Management Architecture for Real-Time Collaborative Applications. *Wiley J. on Wireless Communication and Mobile Computing (WCMC)*, 5:749–772, Nov. 2005.

[74] K.-W. Lee, B.-J. Ko, and S. Calo. Adaptive Server Selection for Large Scale Interactive Online Games. *ACM Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2004.

[75] C. R. Lin and J.-S. Liu. QoS Routing in Ad Hoc Wireless Networks. *IEEE JSAC*, 17, 1999.

[76] H. Luo, R. Ramjee, P. Sinha, L. Li, and S. Lu. UCAN: a Unified Cellular and Ad-Hoc Network Architecture. *ACM MobiCom*, Oct. 2001.

[77] A. Meyerson. Online Facility Location. *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2001.

[78] P. B. Mirchandani and R. L. Francis. *Discrete Location Theory*. John Wiley & Sons Inc., 1990.

[79] T. Moscibroda and R. Wattenhoffer. Facility Location: Distributed Approximation. *ACM PODC*, 2005.

[80] J. Moy. OSPF Version 2. Internet-draft, IETF, Apr. 1998. rfc2328.txt `http://www.ietf.org/rfc/rfc2328.txt`.

[81] M. Naor and L. Stockmeyer. What can be Computed Locally? *ACM Symp. on Theory of Computing*, 1993.

[82] R. Niedermeyer, K. Reinhardt, and P. Sanders. Towards Optimal Locality in Mesh Indexings. *Fudamentals of Computation Theory, LNCS Springer-Verlag*, 1279:364–375, 1997.

[83] J. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *ACM STOC*, 1988.

[84] P. Eugster and R. Guerraoui and S. Handurukande and P. Kouznetsov and A.-M. Kermarrec. Lightweight Probabilistic Broadcast. *ACM TOCS*, (21), 2003.

[85] PK Agarwal and S. Har-Peled and KR Varadarajan. Geometric Approximation via Coresets – Survey. In *Combinatorial and Computational Geometry*. MSRI publication, 2005.

[86] G. P. Pollini. Trends in Handover Design. *IEEE Communications Magazine*, 34, 1996.

[87] L. Qiu, V.N.Padmanabham, and G.M.Voelker. On Placement of Web Server Replicas. *IEEE Infocom*, 2001.

[88] R. Ramjee, D. Towsley, and R. Nagarajan. On Optimal Call Admission Control in Cellular Networks. *Wireless Networks*, 3(1):29–41, Jan. 1997.

[89] T. Roughgarden and E. Tardos. How Bad is Selfish Routing? *Journal of the ACM*, 2002.

[90] L. Song, D. Kotz, R. Jain, and X. He. Evaluating Location Predictors with Extensive Wi-Fi Mobility Data. *IEEE INFOCOM*, 2004.

[91] G. T. SSA. IP Multimedia Subsystem (IMS). Stage 2 (Release 5), 3GPP, 2005.

[92] L. Tasiulas. Linear Complexity Algorithm for Maximum Throughput in Radio Networks and Input Queued Switches. *IEEE Infocom*, 1998.

[93] The Open Group. Application Response Management - ARM. http://www.opengroup.org/tech/management/arm.

[94] O. Tickoo and B. Sikdar. Queueing Analysis and Delay Mitigation in IEEE 802.11 Random Access MAC Based Wireless Networks. *IEEE Infocom*, 2004.

[95] Y. T'Joens, C. Hublet, and P. D. Shrijver. DHCP reconfigure extension. Internet-draft, IETF, Dec. 2001. rfc3203.txt http://www.ietf.org/rfc/rfc3203.txt.

[96] J. Yoon, M. Liu, and B. Noble. Sound Mobility Models. *ACM MobiCom*, 2003.

[97] S. Yun, H. Kim, and I. Kang. Squeezing 100+ VoIP Calls out of 802.11b WLANs. *IEEE WoWMoM*, 2006.

[98] W. T. Zaumen, S. Vutukury, and J. Garcia-Luna-Aceves. Load-Balanced Anycast Routing in Computer Networks. *ISCC*, 2000.